TI Spins Motors

InstaSPIN Projects and Labs User's Guide

InstaSPIN-FOC & InstaSPIN-MOTION for F2802xF, F2805xF, F2805xM, F2806xF, F2806xM

Version 1.0.16 *Motor Solutions*

Product Overview

Both InstaSPIN-FOC and InstaSPIN-MOTION are sensorless or sensored FOC solutions that identify, tune and control your motor in minutes. Both solutions feature:

- 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 option for user adjustments
- Automatic or manual field weakening and field boosting
- Bus voltage compensation

InstaSPIN-MOTION combines this functionality with SpinTAC™ components from LineStream Technologies. SpinTAC features:

- A disturbance-rejecting speed and position controller, which proactively estimates and compensates for system errors. The controller offers single-parameter tuning that typically works over the entire operating range.
- Trajectory planning for easy design and execution of complex motion sequences (Note: this feature will not be exercised through the GUI. See the InstaSPIN-FOC and InstaSPIN-MOTION User Guide and MotorWare projects to exercise SpinTAC Velocity Plan).
- A motion engine that ensures that your motor transitions from one speed to another as smoothly as possible.

Additional information about the features and functionality of InstaSPIN-FOC and InstaSPIN-MOTION 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 reference for your design. These projects will help you develop a successful product. InstaSPIN-FOC and InstaSPIN-MOTION motor control solutions, as well as the lab projects, are delivered within MotorWare. For a latest complete listing of the API functions, MotorWare's Software Coding Standards, and Architecture, run MotorWare.exe from the latest version install.

Ex: C:\ti\motorware\motorware_1_01_00_18\MotorWare.exe

In the lab projects, you will learn how to modify user.h, the 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 also what projects apply to which boards and target device.

		drv8301kit drv8312kit hvkit			boost8301 boost8305		
Solution	**Name**	**2x**	**5x**	**6x**	**2x**	**6x**	**Brief Description**
FOC	Lab01	✓	✓	✓	✓	✓	HAL, Hello world
FOC	Lab01b	✓	✓	✓	✓	✓	Open loop control for hardware integrity verification
FOC	Lab01c	✓	✓	✓	✓	✓	Closed current loop for signal chain verification
FOC	Lab02a		✓	✓		✓	Motor ID from ROM
FOC	Lab02b	✓	✓	✓	✓	✓	Motor ID from RAM/FLASH
FOC	Lab02c	✓	✓	✓	✓	✓	Motor ID for low inductance PMSM
FOC	Lab02d			✓		✓	Lab02b with fpu32
FOC	Lab03a	✓	✓	✓	✓	✓	Using motor parameters w/o offsets
FOC	Lab03b	✓	✓	✓	✓	✓	Using motor parameters with offsets
FOC	Lab03c			✓		✓	Lab03b with fpu32
FOC	Lab04	✓	✓	✓	✓	✓	Torque mode
FOC	Lab04a			✓		✓	Lab04 with fpu32
FOC	Lab05a	✓	✓	✓	✓	✓	Torque mode and tuning Id/Iq PI
FOC	Lab05b	✓	✓	✓	✓	✓	Speed mode and tuning speed PI
MOTION	Lab05c		✓	✓		✓	Inertia ID
MOTION	Lab05d		✓	✓		✓	Running SpinTAC Speed Controller
MOTION	Lab05e		✓	✓		✓	Tuning SpinTAC Speed Controller
MOTION	Lab05f		✓	✓		✓	PI vs. SpinTAC Speed Controllers
FOC	Lab05g			✓		✓	Lab05b with fpu32
FOC	Lab05h	✓	✓	✓	✓	✓	Step Response Generation & Graphing for Controller Tuning

MOTION	Lab06a		✓	✓		✓	Running SpinTAC Profile Generator
MOTION	Lab06b		✓	✓		✓	Running SpinTAC Simple Plan
MOTION	Lab06c		✓	✓		✓	Running SpinTAC Washer Plan
MOTION	Lab06d		✓	✓		✓	Design your own SpinTAC Velocity Plan
MOTION	Lab06e					✓	Dual Motor Sensorless
FOC	Lab07	✓	✓	✓	✓	✓	Rs Online
FOC	Lab07a			✓		✓	Lab07 with fpu32
FOC	Lab09	✓	✓	✓	✓	✓	Field Weakening
FOC	Lab09a			✓		✓	Lab09 with fpu32
FOC	Lab10a	✓	✓	✓	✓	✓	Overmodulation
MOTION	Lab10b		✓	✓		✓	Overmodulation with MOTION
FOC	Lab10c			✓		✓	Lab10a with fpu32
FOC	Lab10d					✓	Dual Motor Sensorless Velocity Control
FOC	Lab10e	✓	✓	✓	✓	✓	Flying Start (Rotor already moving)
FOC	Lab11	✓		✓	✓	✓	Simplified Project without CTRL Object
FOC	Lab11a	✓		✓	✓	✓	Lab11 plus InstaSPIN features
FOC	Lab11b	✓		✓	✓	✓	Lab11a plus Vibration Compensation
FOC	Lab11d					✓	Dual Motor Sensorless Velocity Control
FOC	Lab11e					✓	Hall sensor start-up with transition to FAST
MOTION	Lab12a		✓	✓		✓	Sensored Inertia ID
MOTION	Lab12b		✓	✓		✓	SpinTAC Sensored Speed Control
MOTION	Lab12c					✓	Dual Motor Sensored Velocity Control
MOTION	Lab13a		✓	✓		✓	Tuning SpinTAC Position Control
MOTION	Lab13b		✓	✓		✓	Position Transitions with SpinTAC Move
MOTION	Lab13c		✓	✓		✓	Motion Sequence Position Example
MOTION	Lab13d		✓	✓		✓	Motion Sequence Example – Vending Machine
MOTION	Lab13e		✓	✓		✓	Smooth Velocity Transitions in Position Control
MOTION	Lab13f					✓	Dual Motor Sensored Position Control
FOC	Lab20	✓		✓	✓	✓	New ctrl Structure
FOC	Lab21			✓		✓	Initial Position Detection and High Frequency Injection

Version: 1.0.16

Revision History:

1.0.16	March, 2017	Update for MotorWare v18 New lab & write-up on lab01b (Open loop control for hardware integrity verification) New lab & write-up on lab01c (Closed current loop for signal chain verification) Changed updateRunRsOnLine() functionality to have initial value of USER_MOTOR_Rs instead of _IQ(0.0), which will allow quicker settling time in lab07 and lab011a Relocated CLK_disableTbClockSync() in all hal.c source files to fix issue where PWM sync could sometimes be lost when hard-resetting MCU in CCS debug mode Remove OVM functions in lab5h and remove svgen_current.c file in project Offsets removed from user_2motors.h to avoid issue where current/voltage offsets were defined in two separate header files Added SCI/UART tutorial to motorware_hal_tutorial.pdf under “InstaSPIN Projects Additional Tutorials” Various bug and typo fixes
1.0.15	October, 2016	Update for MotorWare v17 New lab & write-up on lab05h (Step Response Generation & Graphing) New lab & write-up on lab10d (Dual Motor) New lab & write-up on lab10e (Flying Start) New lab & write-up on lab11e (Hall Start) Changed overmodulation technique in lab10a and lab10c to improve robustness Fixed typo on lab06e (noted as lab06f in 1.0.14) Fixed typo on lab11a (field weakening instead of

		overmodulation)
1.0.14	**February, 2016**	**Restored previously truncated write-ups for labs 12b and 13a** **Updated tables with Labs 11d, 11e, 6e, 12c, 13f**
1.0.13	**August, 2015**	**Changed SVM range, instead of 0 to 4/3, now it is from 0 to 2/3, which matches the way SVM outputs are translated into duty cycles, updated in pwm.c/h of MotorWare drivers.** **Fixed page 32/33 use of USER_MOTOR_IDENT_FREQUENCY_Hz instead of USER_MOTOR_FLUX_EST_FREQ_Hz** **Added write-ups for labs 9 and 10** **Added project 11 “Simplified Project without CTRL Object”** **Added project 11a “Lab 11 plus InstaSPIN features”** **Added project 11b “Lab 11a plus Vibration Compensation”**
1.0.12	**Jan 22, 2015**	**Added project 20 “New ctrl Structure”** **Added project 21 “Initial Position Detection and High Frequency Injection”**
1.0.11	**June 19, 2014**	**Modified current controller gains calculation for lab 5a** **Modified speed controller gains calculation for lab 5b** **Added 5xF and 5xM variants to the labs doc** **Added graphing into lab 13a** **New Labs Added:** **12a, 12b (replaces lab 12)**
1.0.10	**March 4, 2014**	**Updated description to lab 2a to include ACIM motors**

		Updated description of lab 3a to include ACIM motors **Added description to lab 7** **Clarified the following in InstaSPIN-MOTION labs** **- Proper use and additions for migrating user.h in –FOC to –MOTION in lab5c** **- Updates for ACI motors in lab12**
1.0.9	**November 12, 2013**	**Added POSITION updates** **New Labs added:** **13a, 13b, 13c, 13d, 13e**
1.0.8	**November 07, 2013**	**Added note for CCS to NOT copy project into workspace (page 11)** **Moved to compiler v6.2.3. If using compiler 6.2.x, use 6.2.3 and greater. An IQmath function prototype malfunction resides in compilers 6.2.0 – 6.2.2.** **Three build configurations are available: Debug, Release, and Flash. Switch between build configurations if a Flash project is required.**
1.0.7	**August 22, 2013**	**First release as standalone document.** **New Labs added:** **2b, 2d, 3c, 4a, 5g, 7a, 9, 10c**
1.0.6	**April 2013**	**Released in the Appendix of the InstaSPIN-FOC & InstaSPIN-MOTION User's Guide.** **New Labs added:** 5c, 5d, 5e, 5f, 6a, 6b, 6c, 6d, 7, 10b, 12
1.0.5	**February 26, 2013**	**First released in the Appendix of the InstaSPIN-FOC User's**

		Guide. **Labs Supported:** 1, 2a, 2c, 3a, 3b, 4, 5a, 5b, 9, 10a

Contents

Lab Descriptions

Lab 1 – CPU and Inverter Setup

This application note covers how to use the HAL object to setup the 2806xF/M, 2805xF/M or 2802xF and inverter hardware. MotorWare API function calls will be used to simplify the microprocessor setup.

Lab 1b – Open loop control to verify integrity of user hardware

This lab implements a scalar volts/frequency control to test the integrity of the hardware, namely 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.

Lab 1c – Closed current, open speed loop control to verify intergrity of user signal chain

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

Lab 2a – Using InstaSPIN for the First Time out of ROM

InstaSPIN implements a FAST enabled self-sensored field oriented variable speed controller. The ROM library contains the FAST angle observer plus all code needed to implement the FOC controller. For this lab we start by using the full ROM based code to create a FOC motor control.

Lab 2b – Using InstaSPIN out of User Memory

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 angle 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.

Lab 2c – Using InstaSPIN to Identify Low Inductance PMSM Motors

This particular lab provides an example to identify difficult PMSM motors, especially the low inductance PMSM motors.

Lab 2d – Using InstaSPIN out of User Memory, with fpu32

This lab is the same as lab 2b, but with fpu32 enabled. This lab only applies to 2806xF device variants.

Lab 3a – Using your own Motor Parameters

By default, InstaSPIN starts by identifying the motor that is attached to the inverter. When identifying the motor, the parameters Rs, Ls, and air gap flux are estimated. This lab will take the motor parameter estimates from the previous lab and place them into the file user.h. User.h is used to hold scaling factors, motor parameters, and inverter parameters for customizing InstaSPIN to any motor control system.

Lab 3b – Using your own Board Parameters: Current and Voltage Offsets (user.h)

Skipping auto-calibration and using your own current and voltage offsets continues to reduce the start-up time. If the board offsets are known, then auto-calibration at start-up is not needed. Also introduced is the option to bypass the Rs Fine Re-estimation.

Lab 3c – Using your own Board Parameters: Current and Voltage Offsets, with fpu32

This lab is the same as lab 3b, but with fpu32 enabled. This lab only applies to 2806xF device variants.

Lab 4 – Current Loop Control (Create a Torque Controller)

The speed loop is disabled and a reference is sent directly to the Iq current controller. Bypassing the speed loop and directly controlling Iq current makes the FOC control torque.

Lab 4a – Current Loop Control (Create a Torque Controller), with fpu32

This lab is the same as lab 4, but with fpu32 enabled. This lab only applies to 2806xF device variants.

Lab 5a – Adjusting the FOC PI Current Controller

For the current loop PIs, InstaSPIN calculates starting Kp and Ki gains for both Id and Iq. During start-up, InstaSPIN identifies the time constant of the motor to determine the Ki and Kp. Sometimes the Id and Iq Kp and Ki gains need to be manually adjusted for an optimal setting. This lab will keep the torque controller from Lab 4 and will show how to manually adjust the current PI controller.

Lab 5b – Adjusting the InstaSPIN-FOC 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 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.

The InstaSPIN-MOTION disturbance-rejecting speed controller replaces the standard PI controller. The InstaSPIN-MOTION controller offers several advantages: 1) it proactively estimates and compensates for system errors; 2) the controller offers single-parameter tuning that typically works over the entire operating range. If you would like to use the InstaSPIN-MOTION controller, you may skip Lab 5b and proceed to Lab 5c.

Lab 5c – InstaSPIN-MOTION Inertia Identification

Inertia identification is the first step in enabling the InstaSPIN-MOTION speed controller. The inertia value is automatically identified by the controller, and is used to determine how strongly to respond to disturbances in the system. In this lab, you will learn how to run the inertia identification process from within your MotorWare project.

Lab 5d – InstaSPIN-MOTION Speed Controller

The InstaSPIN-MOTION speed controller features Active Disturbance Rejection Control (ADRC), which actively estimates and compensates for system disturbances in real time. The InstaSPIN-MOTION speed controller also features a single parameter, bandwidth, which determines the stiffness of the system and dictates how aggressively the system will respond to disturbances. Once tuned, the controller the typically works over a wide range of speeds and loads.

In this lab, you will learn how to replace the InstaSPIN-FOC PI controller with the InstaSPIN-MOTION speed controller in your MotorWare project.

Lab 5e – Tuning the InstaSPIN-MOTION Speed Controller

With single coefficient tuning, InstaSPIN-MOTION allows you to quickly test and tune your velocity control from soft to stiff response. The single gain (bandwidth) typically works across the entire variable speed and load range, reducing complexity and system tuning. In this lab, you will tune the InstaSPIN-MOTION speed controller to obtain the best possible system performance.

Lab 5f – Comparing Speed Controllers

The InstaSPIN-MOTION speed controller shows remarkable performance when compared against a traditional PI controller. This lab will lead you through a comparison of these two controllers.

Lab 5g – Adjusting the InstaSPIN-FOC Speed PI Controller, with fpu32

This lab is the same as lab 5b, but with fpu32 enabled. This lab only applies to 2806xF device variants.

Lab 5h – Step Response Generation & Graphing for Controller Tuning

This lab is to generate step responses for the current and the speed controller. With these step responses it is possible to configure the speed and current PI controller to fit the customer system requirements.

Lab 6a – Smooth System Movement with SpinTAC™ Move

InstaSPIN-MOTION includes SpinTAC Move, a motion profile generator that generates constraint-based, time-optimal motion trajectory curves. It removes the need for lookup tables, and runs in real time to generate the desired motion profile. This lab will demonstrate the different configurations and their impact on the final speed change of the motor.

Lab 6b – Motion Sequence Example

InstaSPIN-MOTION includes SpinTAC Velocity Plan, a motion sequence planner that allows you to easily build complex motion sequences. You can use this functionality to quickly build your application's motion sequence and speed up development time. This lab provides a very simple example of a motion sequence.

Lab 6c – Motion Sequence Real World Example: Washing Machine

This lab builds off Lab 6b and provides a very complex example of a motion sequence.

Lab 6d – Designing your own Motion Sequence

Now that SpinTAC Velocity Plan has been introduced, this lab lets you create your own motion sequence. It is a chance to be creative and utilize the topics and skills that were learned in previous labs.

Lab 6e – Dual motor InstaSPIN-MOTION Sensorless Velocity Control

Based on Lab 11d, Sensorless InstaSPIN-MOTION is implemented to control two inverters independently from one MCU.

Lab 7 – Using Rs Online Recalibration

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

Lab 7a – Using Rs Online Recalibration, with fpu32

This lab is the same as lab 7, but with fpu32 enabled. This lab only applies to 2806xF device variants.

Lab 9 – An Example in Automatic Field Weakening

A simple procedure in automatic field-weakening is explored. The current voltage space vector is always compared to the maximum space vector. A voltage “head room” is maintained by controlling the negative Id current of the FOC controller.

Lab 9a – An Example in Automatic Field Weakening, with fpu32

This lab is the same as lab 9, but with fpu32 enabled. This lab only applies to 2806xF device variants.

Lab 10a – An Example in 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 greater than 86.6%, 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.

Lab 10b – An Example in Space Vector Over-Modulation using InstaSPIN-MOTION

The SVM that is used by InstaSPIN is capable of saturating to a pre-specified duty cycle. When using a duty cycle greater than 86.6%, 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. This example adds the InstaSPIN-MOTION speed controller and profile generator.

Lab 10c – An Example in Space Vector Over-Modulation, with fpu32

This lab is the same as lab 10a, but with fpu32 enabled. This lab only applies to 2806xF device variants.

Lab 10d – Dual motor InstaSPIN-FOC Sensorless Velocity Control

Sensorless InstaSPIN-FOC is implemented to control two inverters independently from one MCU. Only support LaunchXL-F28069M + BoostXL-DRV8301 or BoostXL-DRV8305.

Lab 10e – Flying Start (Rotor already moving)

The lab uses the flying start function in InstaSPIN-FOC, the flying start feature is used to start a rotating motor, as quick as possible and resume normal operation with a minimal impact on load or speed.

Lab 11 – A Simplified Example without Controller Module

This lab utilizes a simplified approach, so that users can see the entire field oriented control system, spelled out in the interrupt service routine. This can be thought of an approach that will be combined with user's code to create a production type of project.

Lab 11a – A Feature Rich Example without Controller Module

Since the inception of InstaSPIN, users have been looking for an example with the least amount of ROM function calls, very straight forward ISR, and with all the features that InstaSPIN provides. Also, users are interested in not having a high level controller module, so that users have the flexibility to modify the project without too many levels of abstraction. This lab provides users both benefits of not having a highly integrated controller module, and at the same time having all the features InstaSPIN brings to sensorless motor control.

Lab 11b – Vibration Compensation Example

In applications where the load is dependent on the mechanical angle such as air conditioning compressors, it is desirable to have a control loop that compensates for the known load. TI created a new library that implements an algorithm that compensates load that causes vibration. This lab shows an example on how to use the vibration compensation library.

Lab 11d – Dual motor InstaSPIN-FOC Sensorless Velocity Control

Sensorless InstaSPIN-FOC is implemented to control two inverters independently from one MCU. Only support LaunchXL-F28069M + BoostXL-DRV8301 or BoostXL-DRV8305.

Lab 11e– Hall Sensor Start with Transition to Sensorless (FAST)

To improve start-up smoothness, an example is shown which uses Hall sensors as the rotor feedback for zero speed start-up until a user adjustable transition point to Sensorless (FAST).

Lab 12a – InstaSPIN-MOTION Inertia Identification with Sensor

This lab demonstrates how to replace the FAST estimator with a quadrature encoder. It will demonstrate identifying the system inertia using a quadrature encoder. This lab also discusses how to identify the system inertia in less than one revolution.

Lab 12b – Using InstaSPIN-MOTION with Sensored Systems

For applications where a sensor is required, InstaSPIN-MOTION can provide the same advantages that it provides for sensorless applications. Currently, InstaSPIN-MOTION supports a quadrature encoder. Hall Effect sensors are not supported at this time. This lab demonstrates how to replace the FAST estimator with a quadrature encoder.

Lab 12c – Dual motor InstaSPIN-MOTION Sensored Velocity Control

Building on Lab 6b, Sensored InstaSPIN-MOTION Velocity Control is implemented to control two inverters independently from one MCU.

Lab 13a – Tuning the InstaSPIN-MOTION Position Controller

Tuning position control applications can be very difficult and time consuming. InstaSPIN-MOTION provides a position-velocity controller that can be tuned using a single coefficient. This single gain (bandwidth) typically works across the entire range of loads and transitions in applications, reducing their complexity. This lab demonstrates how to connect the InstaSPIN-MOTION position controller and tune it for your application.

Lab 13b – Smooth Position Transitions with SpinTAC™ Move

InstaSPIN-MOTION includes SpinTAC Move, a motion profile generator that generates constraint-based, time-optimal position trajectory curves. It removes the need for lookup tables, and runs in real time to generate the desired motion profile. This lab will demonstrate the different configurations and their impact on the final position transition of the motor.

Lab 13c – Motion Sequence Position Example

InstaSPIN-MOTION includes SpinTAC Velocity Plan, a motion sequence planner that allows you to easily build complex motion sequences. You can use this functionality to quickly build your application's motion sequence and speed up development time. This lab provides a very simple example of a motion sequence.

Lab 13d – Motion Sequence Real World Example: Vending Machine

This lab builds off Lab 13c and provides a very complex example of a motion sequence.

Lab 13e – Smooth Velocity Transitions in Position Control

In addition to providing smooth position transitions, SpinTAC Move can also provide smooth speed transitions while still operating in a position control system. This lab demonstrates how to configure SpinTAC Move to generate speed transitions in position mode.

Lab 13f – Dual motor InstaSPIN-MOTION Sensored Position Control

Building on Lab 12c, Sensored InstaSPIN-MOTION Position Control is implemented to control two inverters independently from one MCU.

Lab 20 – New ctrl Structure

To provide easier access to the FOC elements of the control, a new ctrl structure has been created. The previous MotorWare ctrl included all of the modules used to implement FOC. The new ctrl only contains the PI controllers, i.e. the speed, Id, and Iq controllers. The rest of the FOC modules are located in the mainISR.

Lab 21 – Initial Position Detection and High Frequency Injection

Signals are injected into the motor to find the d-axis initial position when the power is first applied to the motor control. After initial position detection, a frequency much higher than the motor's operating frequency range is injected to allow for zero speed control of the motor.

Lab 1 - CPU and Inverter Setup

Abstract

This application note covers how to use the HAL object to setup the 2802xF, 2805xF, 2805xM, 2806xF, 2806xM and inverter hardware. MotorWare API function calls will be used to simplify the microprocessor setup.

Introduction

The first lab is an introduction in using the MotorWare software. The 2802xF, 2805xF, 2805xM, 2806xF and 2806xM processor Clock, GPIOs, Watchdog, and other Peripherals are setup using the HAL object and APIs. The HAL or "Hardware Abstraction Layer" object is the MotorWare interface to controlling micro-controller peripherals and inverter setup. All labs in the InstaSPIN series of motor labs build upon this lab, so it is recommended to perform this lab first before moving on.

Objectives Learned

- Use the HAL object to setup the 2802xF, 2805xF, 2805xM, 2806xF or 2806xM processor.
- Use the HAL object to setup and initialize the inverter.
- Use the enumerations to select settings for peripherals.

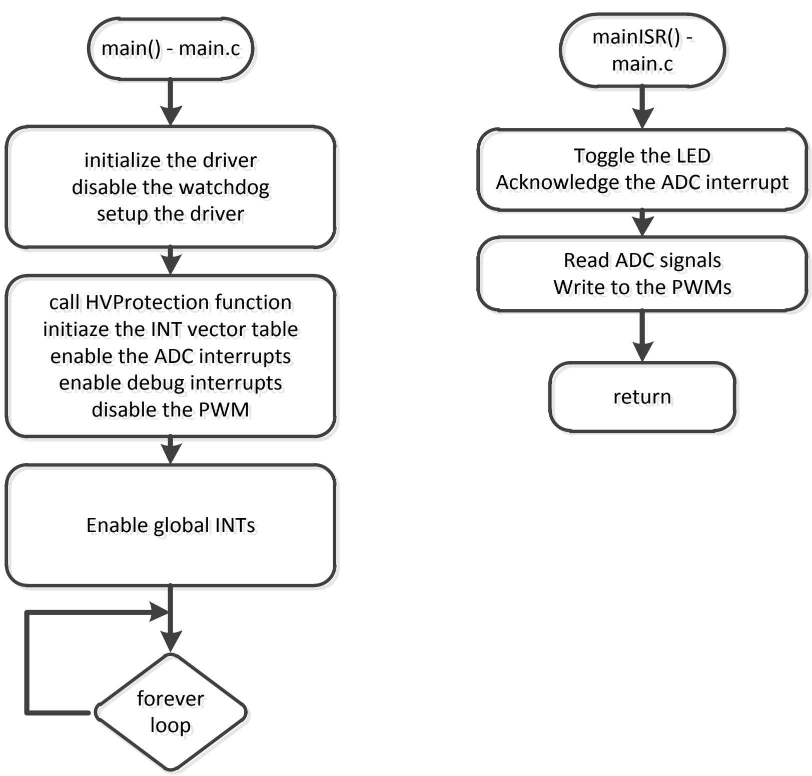


Figure 1: Lab 1 Software Flowchart

Background

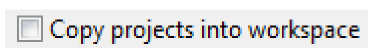
The MotorWare package is used for this lab. The setup of peripherals and even the inverter will be taken care of by MotorWare APIs, specifically the HAL object and APIs. Important commands to initialize the drive are listed in the flowchart of Figure 1. The files and project are located in the MotorWare directory as shown in Figure 2 depending on which processor the user is working with, below.

For lab 1 and all InstaSPIN-FOC lab projects, the MotorWare path refers to a 2802xF, 2805xF or 2806xF device series. All TMS320F2802x**F**, TMS320F2805x**F** and TMS320F2806x**F** devices include the appropriate ROM capability to run these projects. All of the projects are based on an inverter board (DRV8312, DRV8301, HVMTR, etc.) paired with the TMDSCNCD28027F, TMDSCNCD28054MISO or TMDSCNCD28069MISO controlCARD. The TMDSCNCD28069MISO controlCARD uses the TMS320F28069**M** device. The TMDSCNCD28054MISO controlCARD uses the TMS320F28054**M** device. The "**M**" devices are identical to "**F**" devices but include *additional* ROM capability for InstaSPIN-MOTION. "**M**" and "**F**" devices are identical in the software and behavior for all InstaSPIN-FOC projects.

To open lab 1 use the following menu selection: Project -> Import Existing CCS/CCE Project ->

Select the "ccs" folder at the appropriate MotorWare revision, board and MCU target combination to import all projects for these labs.: Ex: c:\ti\motorware\MotorWare_1_01_00_18\sw\solutions\instaspin_foc\boards\drv8312kit_revD\f28x\f2806xF \projects\ccs

Do NOT select Copy the projects into the workspace
work out of the parent ti\motorware\motorware_#_##_##_## directory



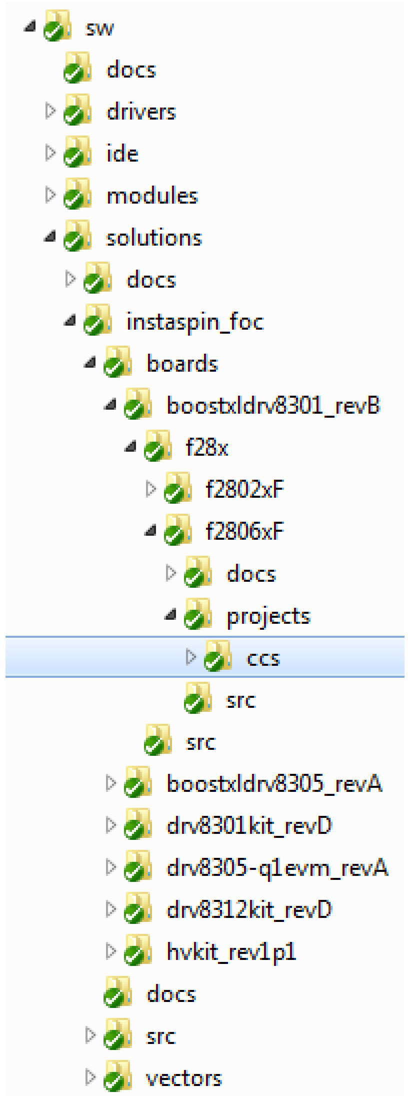


Figure 2: Lab 1 Location

Includes

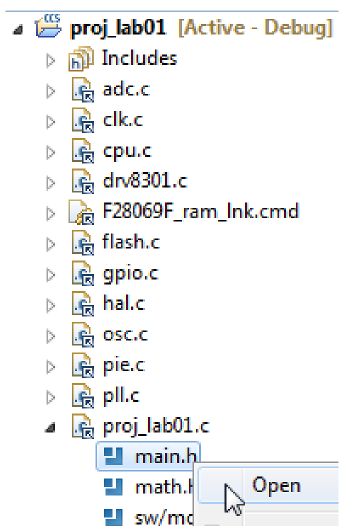
A description of the included files for Lab 1 is shown in the below tables. Note that main.h is common across all labs so there will be more includes than are needed for this lab.

Table 1: Important header files needed for the setup.

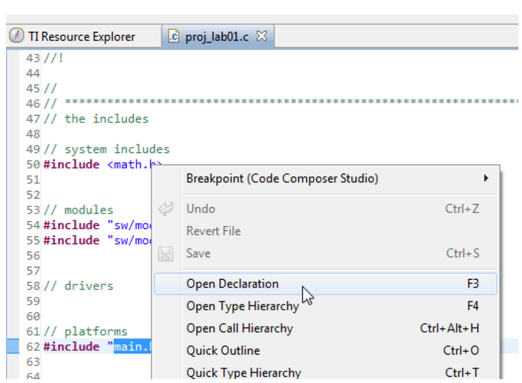
main.h	Header file containing all included files used in main.c	
	modules	
	math.h	Common math conversions, defines, and shifts
	est.h	Function definitions for the FAST ROM library
	platforms	
	hal.h	Device setup and peripheral drivers. Contains the HAL object.
	user.h	User file for configuration of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

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

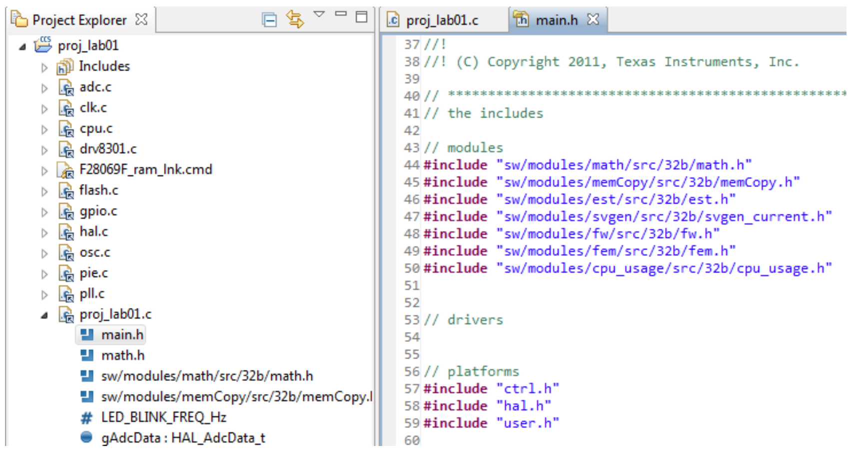
1. Select the arrow for the file "proj_lab01.c" to view the include files included within this file
2. Right-mouse click on "main.h", this will open "proj_lab01.c" with the reference to "main.h" highlighted



3. Right-mouse click on "main.h" and select "Open Declaration"



4. main.h is now open for review



Global Object and Variable Declarations

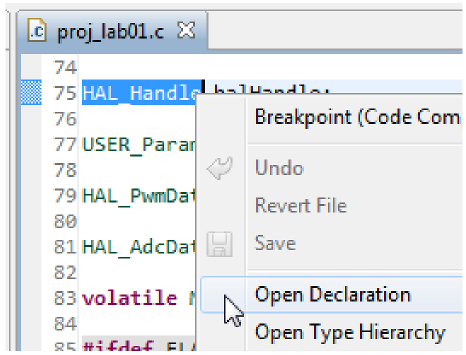
Global objects and declarations that are listed in the table below are only the objects that are absolutely needed for the drive setup. Other object and variable declarations are used for display or information for the purpose of this lab.

Table 2: Global object and variable declarations important for the setup

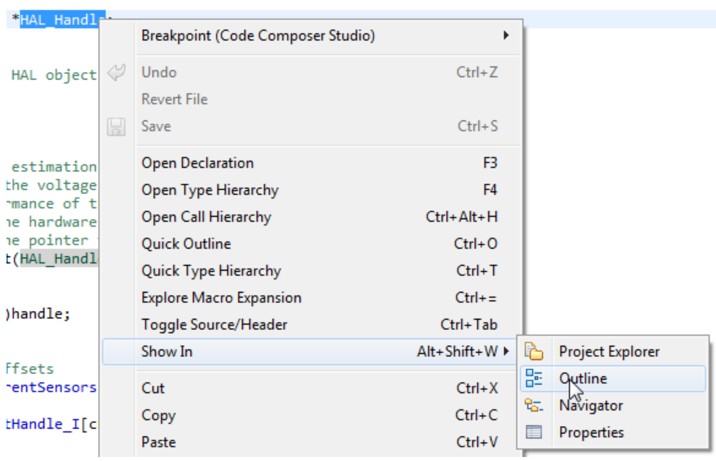
globals		
	CTRL	
	HAL_Handle	The handle to the hardware abstraction layer object (HAL). The driver object contains handles to all microprocessor peripherals and is used when setting up and controlling the peripherals.
	USER_Params	Holds the scale factor information that is in user.h. Allows for scale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

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

1. In the file "proj_lab01.c" right-mouse click on HAL_Handle and select "Open Declaration"

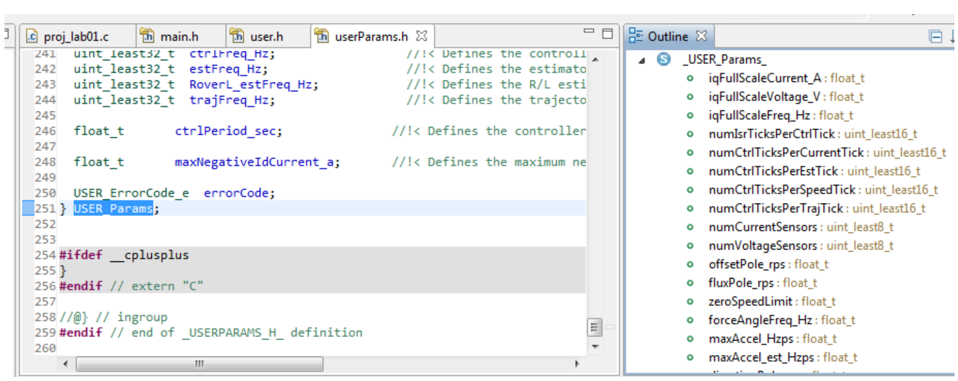


2. With the file “hal_obj.h” now open, right- mouse click on HAL_Handle and select “Show In Outline”



3. With the Outline View open, expand “_HAL_Obj_” to see each member of the HAL object

4. In the file “proj_lab01.c” right-mouse click on USER_Params and select “Open Declaration”
5. From the Outline view, expand _User_Params_ to display each member of the object



Initialization and Setup

This section covers functions needed to setup 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 3: Important setup functions needed for the motor control are in the project for enhanced capability of the laboratory and not fundamentally needed to setup the drive. For a more in depth explanation for definitions of the parameters and return values go to the MotorWare section of this document (InstaSPIN-FOC and InstaSPIN-MOTION User's Guide, SPRUHJ1).

Table 3: Important setup functions needed for the motor control

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.

mainISR

The methods used inside of the mainISR() are time critical and are used run-time. 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 with HAL_writePwmData() resulting in a 50% duty cycle since the gPwmData{} values are initialized to zero when defined. Table 4 explains the functions used in the mainISR.

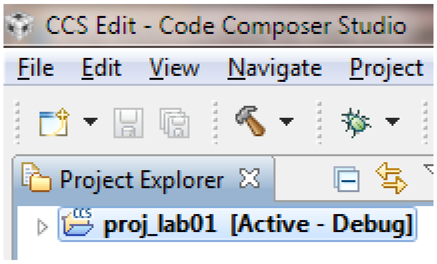
Table 4: The mainISR API functions that are used for Lab 1.

mainISR		
	HAL_toggleLed	Toggles the LED on the motor inverter.
	HAL_acqAdcInt	Acknowledges the ADC interrupt so that another ADC interrupt can happen again.
	HAL_readAdcData	Reads in the Adc result registers, adjusts for offsets, and scales the values according to the settings in user.h. The structure gAdcData holds three phase voltages, three line currents, and one DC bus voltage.
	HAL_writePwmData	Converts the Q pwm values in gPwmData to Q0 and writes these values to the EPWM compare registers.

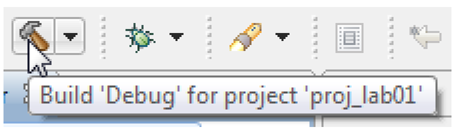
Lab Procedure

Lab1 is a quick lab similar to "Hello World" type of programs. The corresponding embedded programming code is to blink a LED #2 on the ControlCARD. The goal is to review the MCU and inverter setup functions, specifically the HAL object and make sure the LED blinks.

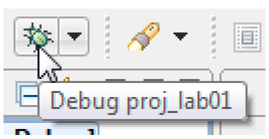
1. Insert the MCU control card, connect the USB cable to the control card, and finally apply power to the kit.
2. Click proj_lab01 at the top of the Project Explorer tab (near the top left of your screen).



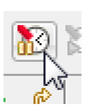
3. Click the hammer that symbolizes “Build”.

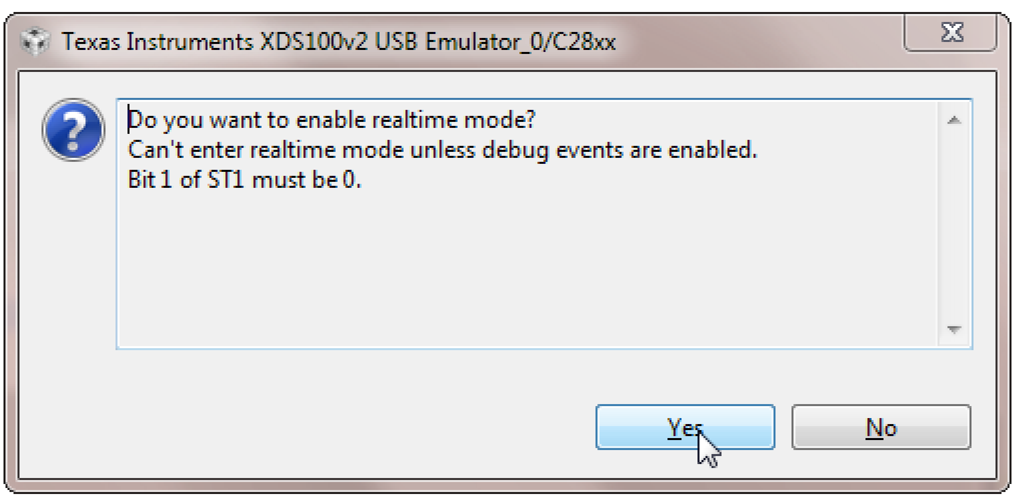


4. Click the green bug that symbolizes “Debug”. This button should automatically change the CCS perspective to “CCS Debug” as well as load the .out file to the target.



5. Click Real-Time Silicon Mode which looks like a clock and press “Yes” if a small window pops up.

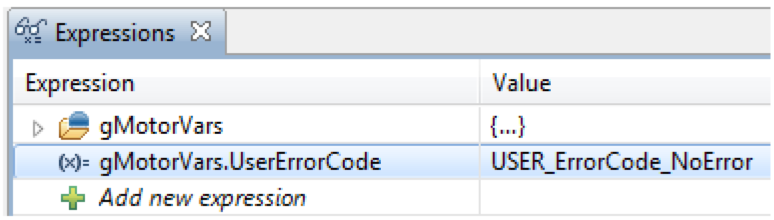




6. Click Resume which looks like a yellow vertical line with a green triangle besides it.

7. One of the LED labeled “LD2” on the inverter board will blink.

8. If LED labeled “LD2” doesn’t blink, there might be a user configuration error. In order to see if this is the case, take a look at the following variable in the watch window:



If this variable is different than “USER_ErrorCode_NoError”, edit user.h file to address the error shown by variable gMotorVars.UserErrorCode.

9. Lab 1 is complete.

Conclusion

The HAL object was created to ease the setup of the MCU and inverter. Lab 1 taught us how to use the HAL object to setup and initialize the MCU and inverter. We will build on this functionality to see how to enable InstaSPIN.

Lab 1b – Open loop control for hardware integrity

Abstract

This lab 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.

Introduction

In lab 1b, we show an example without FAST estimator. Instead, we utilize an angle generator module to generate the angle based on motor target frequency, and use a Volt/Hertz profile to generate an output command voltage to drive a motor. The intention is to test the InstaSPIN-FOC modules through this scalar Volt/Hertz control, of which modules include PWM, ADC, CLARK, PARK and SVGEN.

Prerequisites

Assumes knowledge of this guide up to proj_lab01.

Objectives Learned

- How to implement a simple scalar volt/Hertz control of motor.
- How to test InstaSPIN-FOC modules

Background

In order to achieve better dynamic performance, a more complex control scheme needs to be applied to control the ACI or PM motor. In V/Hz control, the stator voltage to frequency ratio is usually based on the rated values of these variables. The typical V/Hz profile can be shown in Figure 3. 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 the Vs. So, 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, it follows the constant V/Hz relationship.

• At higher frated Hz, the constant Vs/f ratio cannot be satisfied because the stator voltages would be limited at the rated value in order to avoid insulation breakdown at stator windings. Therefore, the resulting air gap flux would be reduced, and this will unavoidably cause the decreasing developed torque correspondingly. This region is usually called the "field weakening region". To avoid this, constant V/Hz principle is also violated at such frequencies.

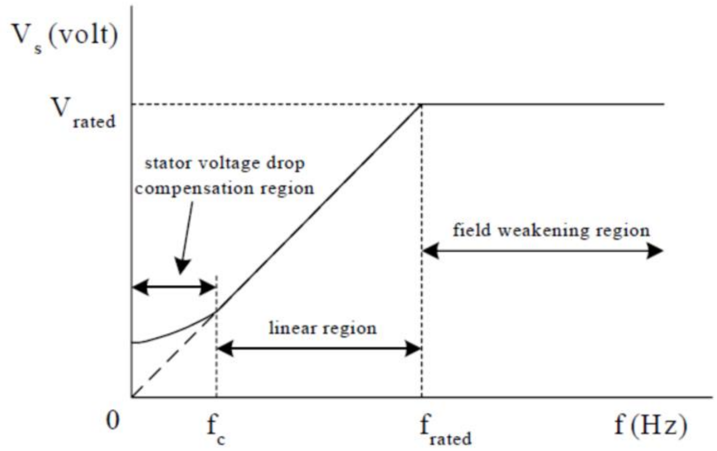


Figure 3 Stator voltage versus frequency profile under V/Hz control

In this lab, the profile is modified as Figure 4 by imposing a lower limit on frequency. This approach is acceptable to applications such as fan and blower drives where the speed response at low end 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 is needed to implement the 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}.

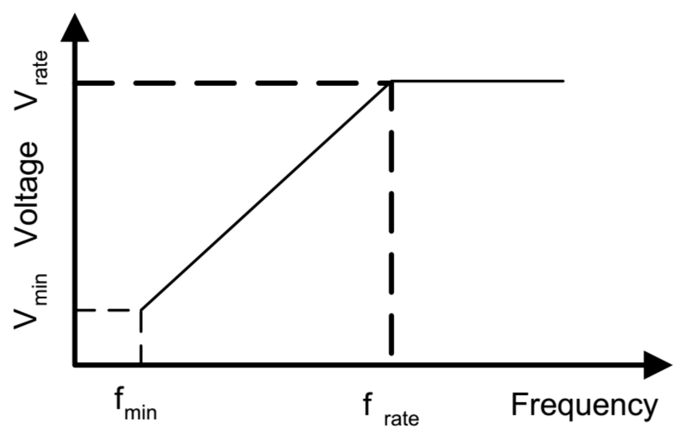


Figure 4 Modified V/Hz profile

In this lab, a ramp generator module is used to generate the angle based on motor target frequency. One such variable, called *StepAngleMax*, is used to determine the minimum period (1/frequency) of the ramp signal. Adding a fixed step value to the *Angle_pu* variable causes the value in *Angle_pu* to cycle at a constant rate.

*Angle_pu = Angle_pu + StepAngleMax*Freq*

At the end limit, the value in *Angle_pu* 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:

$$f = \frac{StepAngle * f_s}{2^N}$$

where f_s is the sampling loop frequency in Hz and N = Global Q value in the auto wrapper variable *Angle_pu.*

For IQmath implementation, the maximum step size in per-unit, StepAngleMax, for a given base frequency, f_b and a defined GLOBAL_Q number is computed as follows:

$$StepAngleMax = f_b \times T_s \times 2^{GLOBAL\_Q}$$

Equivalently, by using _IQ() function for converting from a floating-point number to a _IQ number, the StepAngleMax can also be computed as

$$StepAngleMax = \_IQ(f_b \times T_s)$$

where T_s is the sampling period (sec).

PWMDAC

This module converts any s/w variables into the PWM signals in EPWMxA/B for C2000 MCU. Thus, it can be used to view the signal, represented by the variable, at the outputs of the PWMxA, PWMxB, pins through the external RC low-pass filters.

Step 1, declare object for this PWMDAC module in in proj_lab01b.c.

```
// the PWMDAC variable
HAL_DacData_t gDacData;
```

Follow the Step 2a to monitor the SVGEN for PWM output.

Step 2a, set the right offset and gain for each PWMDAC channel in proj_lab01b.c and hal.c

```
// set DAC parameters
 gDacData.ptrData[0] = &gPwmData.Tabc.value[0];
 gDacData.ptrData[1] = &gPwmData.Tabc.value[1];
 gDacData.ptrData[2] = &gPwmData.Tabc.value[2];
 gDacData.ptrData[3] = &gAdcData.V.value[0];

 HAL_setDacParameters(halHandle, &gDacData);

// set PWMDAC parameters for each channel to ensure the output waveform
void HAL_setDacParameters(HAL_Handle handle, HAL_DacData_t *pDacData)
{

   HAL_Obj *obj = (HAL_Obj *)handle;
   pDacData->PeriodMax = PWMDAC_getPeriod(obj->pwmDacHandle[PWMDAC_Number_1]);

   pDacData->offset[0] = _IQ(0.5);
   pDacData->offset[1] = _IQ(0.5);
   pDacData->offset[2] = _IQ(0.5);
   pDacData->offset[3] = _IQ(0.5);

   pDacData->gain[0] = _IQ(1.0);
   pDacData->gain[1] = _IQ(1.0);
   pDacData->gain[2] = _IQ(1.0);
   pDacData->gain[3] = _IQ(1.0);

} // end of HAL_setDacParameters() function
```

Step 3a, Connect inputs of the PWMDAC module.

```
 // connect inputs of the PWMDAC module.
 gDacData.value[0] = (*gDacData.ptrData[0]);       //
 gDacData.value[1] = (*gDacData.ptrData[1]);       //
 gDacData.value[2] = (*gDacData.ptrData[2]);       //
 gDacData.value[3] = (*gDacData.ptrData[3]);       //

 HAL_writeDacData(halHandle,&gDacData);
```

In lab01b, we connect project-related variables to the PWMDAC module; the output waveform of the DACs for the DRV8312_RevD kit + F28069M ControlCard combination is as shown Figure 5:

Ch1-> PWMDAC1: SVGEN output for phase A
Ch2-> PWMDAC2: SVGEN output for phase B
Ch3-> PWMDAC3: SVGEN output for phase C

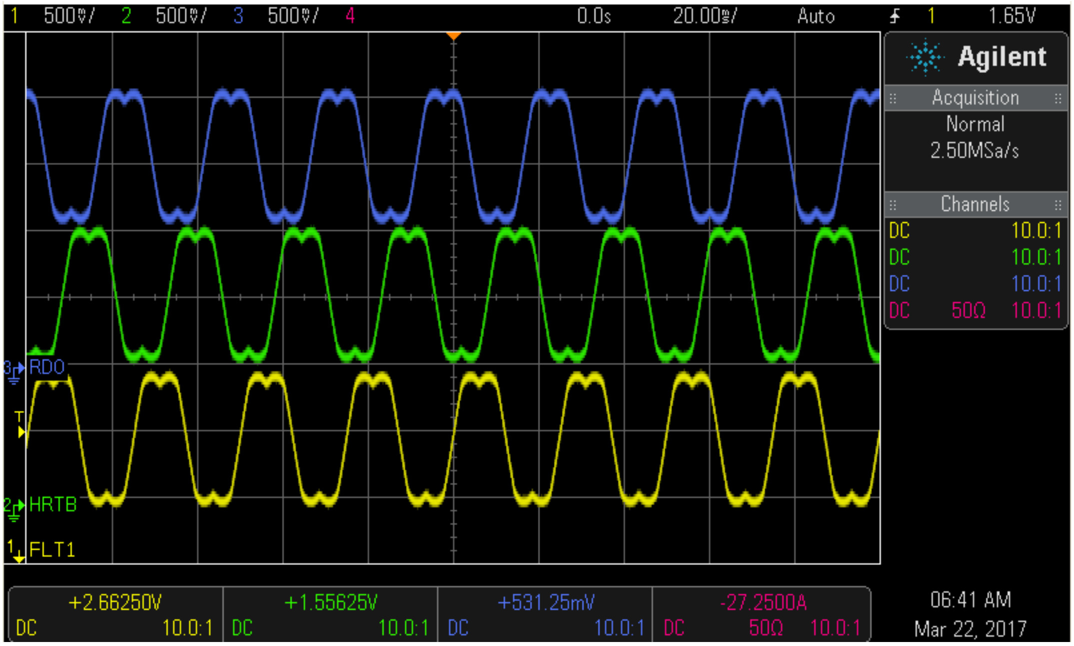


Figure 5 SVGEN output waveform

Follow the Step 2b to monitor the Angel_gen, Current and Voltage for ADC sampling.

Step 2b, set the right offset and gain for each PWMDAC channel in proj_lab01b.c and hal.c

```
// set DAC parameters
 gDacData.ptrData[0] = &angle_gen.Angle_pu;
 gDacData.ptrData[1] = &gAdcData.I.value[0];
 gDacData.ptrData[2] = &gPwmData.Tabc.value[0];
 gDacData.ptrData[3] = &gAdcData.V.value[0];

 HAL_setDacParameters(halHandle, &gDacData);

// set PWMDAC parameters for each channel to ensure the output waveform
void HAL_setDacParameters(HAL_Handle handle, HAL_DacData_t *pDacData)
{

   HAL_Obj *obj = (HAL_Obj *)handle;
   pDacData->PeriodMax = PWMDAC_getPeriod(obj->pwmDacHandle[PWMDAC_Number_1]);

   pDacData->offset[0] = _IQ(0.0);
   pDacData->offset[1] = _IQ(0.5);
   pDacData->offset[2] = _IQ(0.5);
   pDacData->offset[3] = _IQ(0.5);

   pDacData->gain[0] = _IQ(1.0);
   pDacData->gain[1] = _IQ(1.0);
   pDacData->gain[2] = _IQ(1.0);
   pDacData->gain[3] = _IQ(1.0);

} // end of HAL_setDacParameters() function
```

As above, we connect relevant variables to PWMDAC module, and monitor the output waveform as seen in Figure 6. (DRV8312_RevD kit + F28069M ControlCard)

Ch1-> PWMDAC1: Angle output of angel_gen module
Ch2-> PWMDAC2: Phase A current sampling from ADC
Ch3-> PWMDAC3: Phase A PWM output
Ch4->Sampling: Phase A current wavefrom with current probe

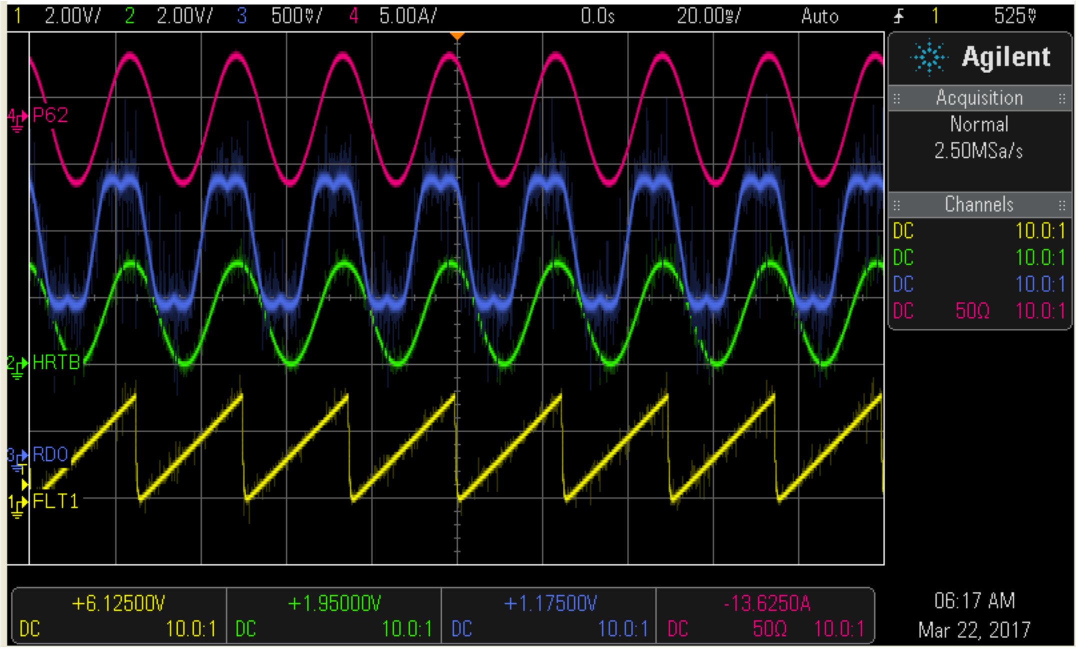


Figure 6 Angel, Current and Voltage

DATALOG

The datalog module can be utilized to store and graph the data. Using the graph tool in CCS, it is possible to implement a virtual oscilloscope to monitor the input/output waveform for datalog-enabled projects.

Setup and use the graph window

Setup the graph window as instructed in the following steps, as well as in the menu shown Figure 7

Step 1. Click Tools->Graph->Dual time in CCS menu. Click the Import button and select the appropriate single or dual time file from “..\sw\solutions\instaspin_foc\src\proj_lab01b_d1&d2_1.graphProp.”

Step 2. Click Tools->Graph->Single time in CCS menu. Click the Import button and select the appropriate single or dual time file from “..\sw\solutions\instaspin_foc\src\proj_lab01b_d3_1.graphProp.”

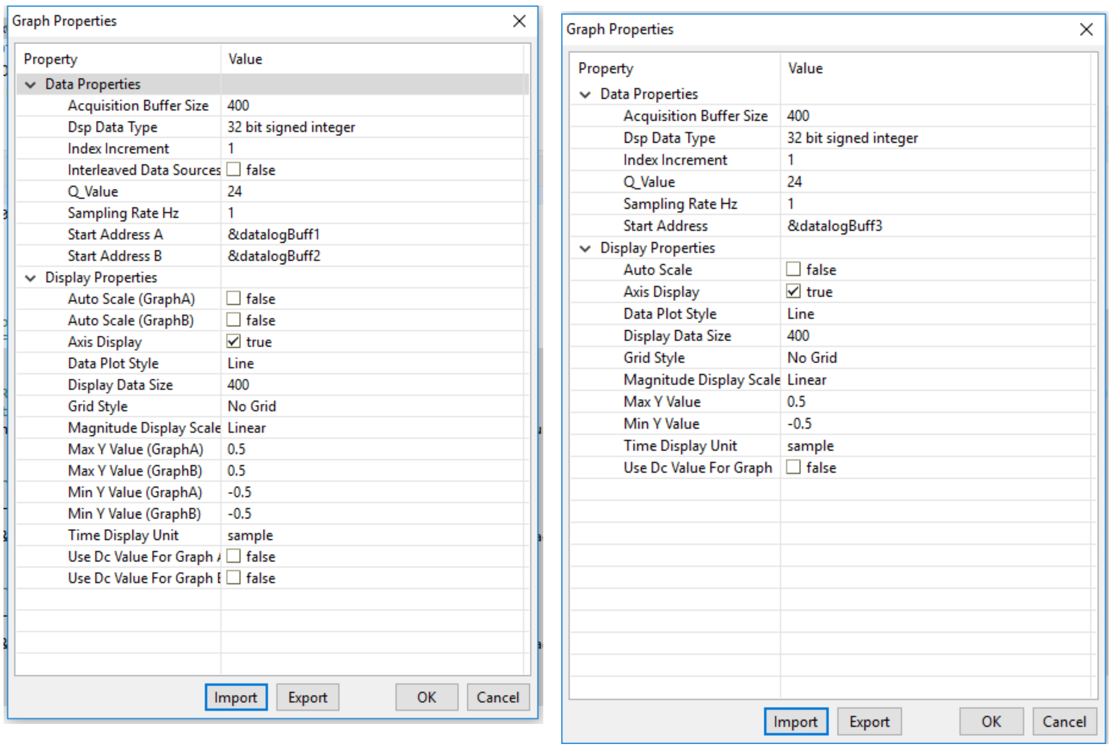
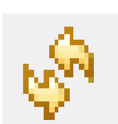


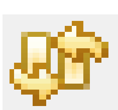
Figure 7 Graph Tool Properties Configuration

Please note that the Auto Scale (GraphA) and Auto Scale (GraphB) values in the following figure are set to false. This enables the option of Max Y value and Min Y value of GraphA and GraphB. Then select OK. The graphs will import into CCS. Arrange the windows as you like in your workspace.

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

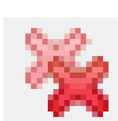


Press this button to refresh the current graph window with the current values written into the data array. This button needs to be pressed every time you have received new data, to show the new graph.



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

Refresh the graph every time you halt the debug session.



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

Step 3a. Connect inputs of the datalog module; the datalog graph for SVGEN output is shown in Figure 7

```
// Connect inputs of the datalog module
datalog.iptr[0] = &gPwmData.Tabc.value[0];        // datalogBuff[0]
datalog.iptr[1] = &gPwmData.Tabc.value[1];        // datalogBuff[1]
datalog.iptr[2] = &gPwmData.Tabc.value[2];        // datalogBuff[2]

datalog.Flag_EnableLogData = true;
datalog.Flag_EnableLogOneShot = false;
```

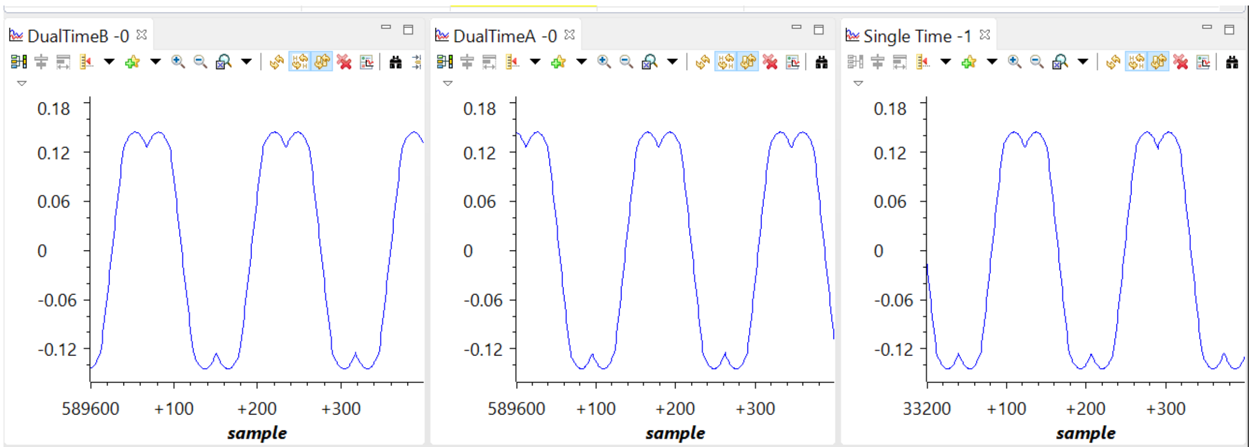


Figure 8 datalog graph for SVGEN output

Step 3a. Connect inputs of the datalog module; view the datalog graph for angle and sampling current/voltage of ADC as shown in Figure 9.

```
datalog.iptr[0] = &angle_gen.Angle_pu;            // datalogBuff[0]
datalog.iptr[1] = &gAdcData.I.value[0];           // datalogBuff[1]
datalog.iptr[2] = &gAdcData.V.value[0];           // datalogBuff[2]
```

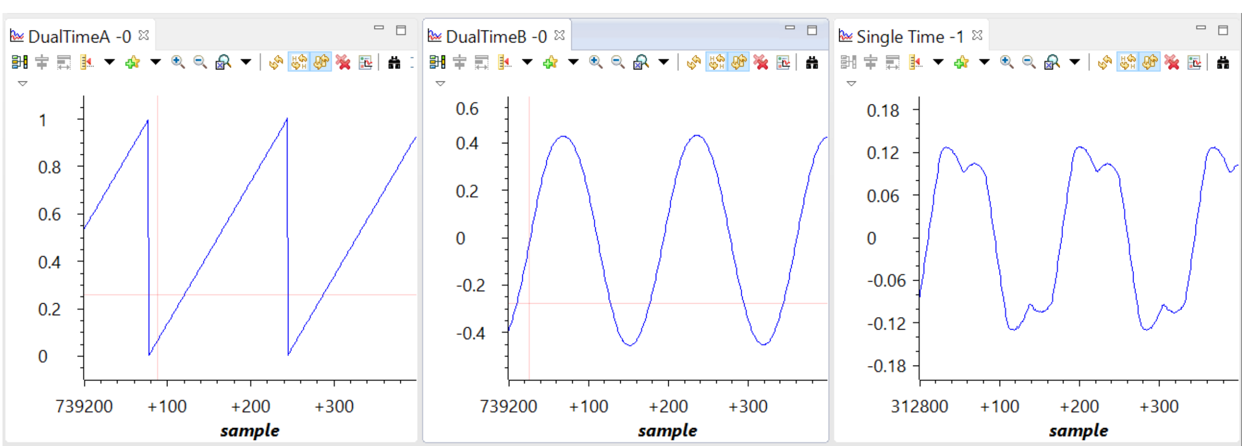


Figure 9 datalog graph for angle, sampling current and volatge

Project Files

Compared to proj_lab01, new files have been added for proj_lab01b as shown below:

Table 5: New files that must be included in the project for Lab1b.

Proj_lab01b		
	angle_gen.c	Define the angle generator module routines
	vs_freq.c	Define the volts/Hertz profile module routines

Includes

A description of the included files for proj_lab01b is shown in the below Table 6. Note that main.h is common across all Motorware projects, so there will be more included header files than needed for this lab.

Table 6: Important header files needed for the Lab01b.

main.h	Header file containing all included files used in proj_lab01b.c, brief Defines the structures, global initialization, and functions used in 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.
	ctrl.h	Contains the public interface, object and function definitions for various functions related to the CTRL object.
	ctrl_obj.h	Defines the structures for the CTRL object
	user.h	Contains the motor control initialization data for the CTRL, HAL, and EST modules

Global Object and Variable Declarations

Global objects and declarations listed in Table 7 below are the objects that are absolutely needed for the drive setup. Other object and variable declarations are used for display or information only for the purpose of this lab.

Table 7: Global object and variable declarations are important for the setup.

globals			
	CTRL_Handle	ctrlHandle	The handle to a controller object (CTRL). The controller object implements all of the FOC algorithms and calls the FAST observer functions.
	MOTOR_Vars_t	gMotorVars	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.
	ANGLE_GEN_Handle ANGLE_GEN_Obj	angle_genHandle angle_gen	The object and handle of a Angle Generator struct.
	VS_FREQ_Handle VS_FREQ_Obj	vs_freqHandle vs_freq	The object and handle of a Volts/Hertz Profile struct.

Initialization and Setup

This section covers functions needed to setup 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 11 are in the project for enhanced capability of the laboratory and not fundamentally needed to setup the motor control. For a more in depth explanation for definitions of the parameters and return values go to the document MotorWare section of this document (InstaSPIN-FOC and InstaSPIN-MOTION User's Guide, SPRUHJ1G).

Table 8: Important setup functions needed for the motor control.

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.

Background

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

mainISR

The main ISR calls very critical, time dependent functions that run the FOC. A block diagram for Lab01b is shown in Figure 5.

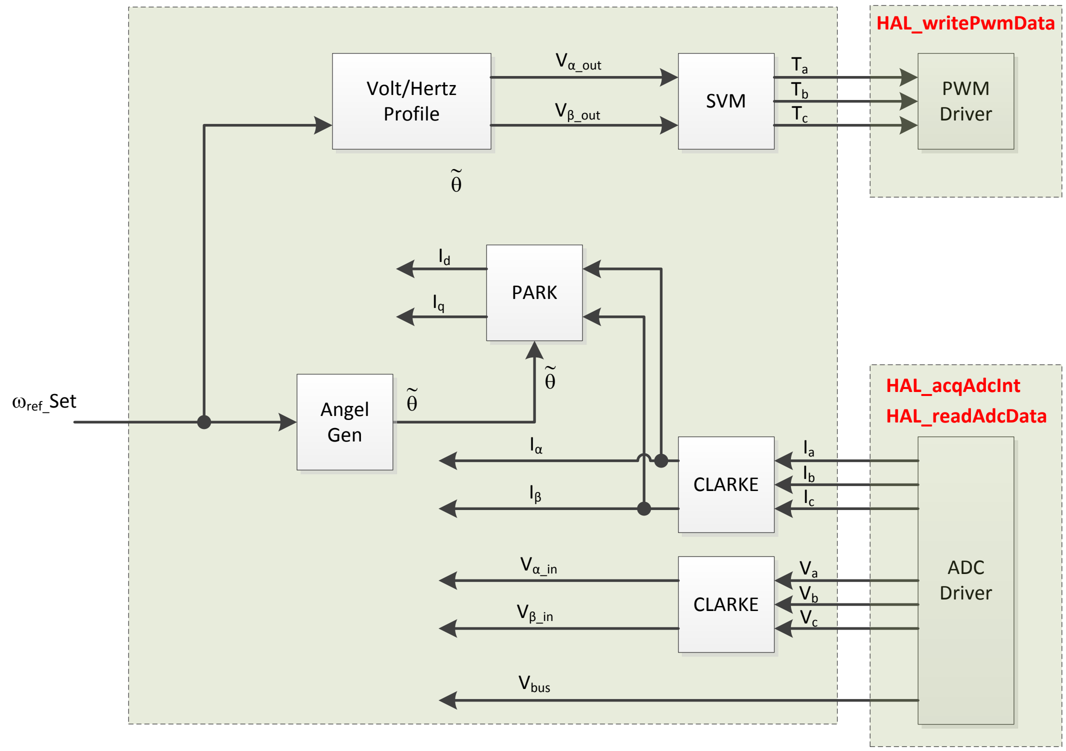


Figure 10 Block diagram of Volt/Hertz control for motor

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.

The first topic that needs to be covered before running any motor with Lab01b is changes to the file "user.h".

Open user.h following these steps:

1. Expand user.c from the 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 Outline View will provide an outline of the user.h contents

In proj_lab01b, open user.h from the directory in MotorWare as shown below in Figure 11.

It is advised to change the below parameters to reflect the specifications of the motor being tested, as instructed in the user.h comments

```
#define USER_MOTOR_FREQ_LOW          (10.0)         // Hz - suggested to set to 10% of rated motor frequency
#define USER_MOTOR_FREQ_HIGH         (100.0)        // Hz - suggested to set to 100% of rated motor frequency
#define USER_MOTOR_FREQ_MAX          (120.0)        // Hz - suggested to set to 120% of rated motor frequency
#define USER_MOTOR_VOLT_MIN          (3.0)          // Volt - suggested to set to 15% of rated motor voltage
#define USER_MOTOR_VOLT_MAX          (18.0)         // Volt - suggested to set to 100% of rated motor voltage
```

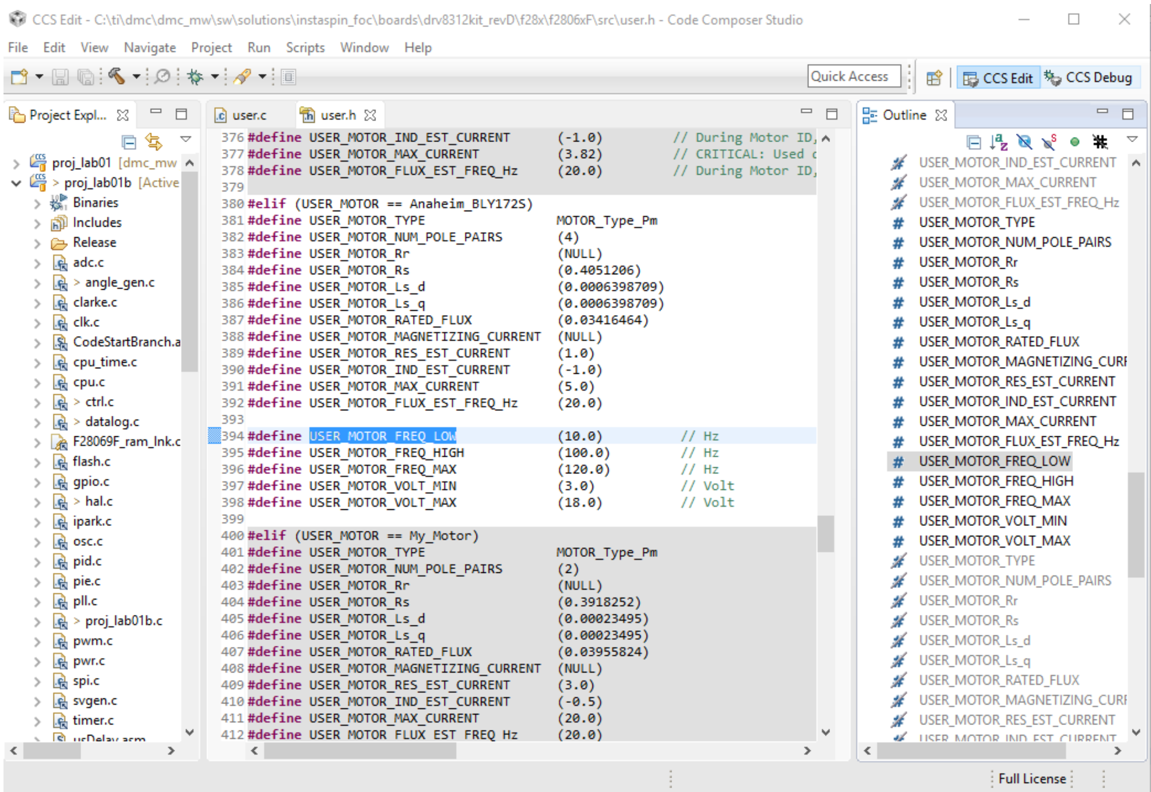


Figure 11: “user.h” in lab project

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

- 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 “sw\solutions\instaspin_foc\src\proj_lab01b.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 12
  - Note the number format.
  - For example, if “gMotorVars.Flag_enableSys” is displayed as a character, right-mouse click on it and select “Number Format -> Decimal”
  - For the “Q-Value(24)” format, after right-mouse clicking on the value select “Q-Values->Q-Value(24)” from the pop-up menu

Expressions

Expression	Type	Value
gMotorVars	struct _MOTOR_Vars_t_	{...}
gMotorVars.UserErrorCode	enum unknown	USER_ErrorCode_NoError
gMotorVars.CtrlVersion	struct _CTRL_Version_	{...}
gMotorVars.Flag_enableSys	unsigned char	0 (Decimal)
gMotorVars.Flag_Run_Identify	unsigned char	0 (Decimal)
gMotorVars.Flag_enableUserParams	unsigned char	1 (Decimal)
gMotorVars.Flag_enableRsRecalc	unsigned char	0 (Decimal)
gMotorVars.Flag_enableForceAngle	unsigned char	1 (Decimal)
gMotorVars.Flag_enableOffsetcalc	unsigned char	1 (Decimal)
gMotorVars.Flag_enablePowerWarp	unsigned char	0 (Decimal)
gMotorVars.CtrlState	enum unknown	CTRL_State_Idle
gMotorVars.EstState	enum unknown	EST_State_Idle
gMotorVars.SpeedRef_krpm	long	0.09999996424 (Q-Value(24))
gMotorVars.MaxAccel_krpmps	long	0.1999999881 (Q-Value(24))
gMotorVars.Speed_krpm	long	0.0 (Q-Value(24))
gMotorVars.Torque_Nm	long	0.0 (Q-Value(24))
gMotorVars.MagnCurr_A	float	0.0
gMotorVars.Rr_Ohm	float	0.0
gMotorVars.Rs_Ohm	float	0.0
gMotorVars.Lsd_H	float	0.0
gMotorVars.Lsq_H	float	0.0
gMotorVars.Flux_VpHz	float	0.0
gMotorVars.Kp_spd	long	0.0 (Q-Value(24))
gMotorVars.Ki_spd	long	0.0 (Q-Value(24))
gMotorVars.Kp_Idq	long	0.0 (Q-Value(24))
gMotorVars.Ki_Idq	long	0.0 (Q-Value(24))
gMotorVars.VdcBus_kV	long	0.0 (Q-Value(24))
vs_freq	struct _VS_FREQ_Obj_	{...}
vs_freq.VfSlope	long	-0.8084905148 (Q-Value(24))
vs_freq.VoltMin	long	6.328594446 (Q-Value(24))
angle_gen	struct _ANGLE_GEN_Obj_	{...}
gDacData	struct _HAL_DacData_t_	{...}
datalog.Flag_EnableLogOneShot	unsigned char	0 (Decimal)

Figure 12 Watch Window in Lab01b

Step 1. Disconnect the motor, power on the EVM kit, start lab, and follow PWMDAC and DATALOG modules instruction to monitor some signals from SVGEN.

- 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 "gMotorVars.Flag_enableSys" equal to 1.
- Set the variable "gMotorVars.Flag_Run_Identify" equal to 1.

Step 2. Power off the EVM kit, and follow PWMDAC and DATALOG modules instruction to change configuration settings to monitor some signals from ANGEL_GEN and ADC.

- Set the variable "gMotorVars.Flag_Run_Identify" to 0 to turn off the PWMs to the motor.
- Turn off real-time control and stop the debugger.

- Power off EVM kit

Step 3. Connect motor, power on EVM kit,
- Enable the real-time debugger again.
- Set the variable “gMotorVars.Flag_enableSys” and “gMotorVars.Flag_Run_Identify” equal to 1 again.
- Set the variable “gMotorVars.SpeedRef_krpm” to a different value to ensure the motor still runs.

Step 4. Refer to Step 2 above to power off the EVM kit if done experimenting.

Conclusion

Lab 01b has demonstrated how to implement a scalar volts/frequency control to test the integrity of the hardware, like the PWM and ADC modules for hardware abstraction layer (HAL) setup. In addition, this lab demonstrated use of the PWMDAC and Datalog functions to verify the SVGEN, PARK, and PWM modules.

Lab 1c – Closed current loop control for signal chain integrity

Abstract

This lab implements a scalar volts/frequency closed current loop, open speed loop control to test the signal chain integrity, such as the hardware current/voltage sensing and controller ADC module. While compatible with Texas Instruments' hardware, this lab is intended for custom hardware verification.

Introduction

In lab 1c, we show an example without using the FAST estimator. Instead, lab 1c uses an angle generator module to generate the angle based on target frequency of motor, as well as employs a closed current loop to control Id and Iq to run a motor. The objective is to test the InstaSPIN-FOC modules through current close loop control without rotor position information; for this objective, software modules include: PI, PWM, ADC, CLARK, PARK I-PARK and SVGEN.

Prerequisites

Assumes knowledge of this guide up to proj_lab01b.

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 lab01b, use PWMDAC to view FOC critical signals from ADC or PI modules.

Step 1, same as lab01b

Step 2a, set the right offset and gain for each PWMDAC channel in proj_lab01c.c and hal.c

```
 // set DAC parameters
  gDacData.ptrData[0] = &angle_gen.Angle_pu;
  gDacData.ptrData[1] = &controller_obj->pid_Iq.refValue;
  gDacData.ptrData[2] = &controller_obj->pid_Iq.fbackValue;
  gDacData.ptrData[3] = &gPwmData.Tabc.value[0];

  HAL_setDacParameters(halHandle, &gDacData);

// set PWMDAC parameters for each channel to ensure the output waveform
void HAL_setDacParameters(HAL_Handle handle, HAL_DacData_t *pDacData)
{

   HAL_Obj *obj = (HAL_Obj *)handle;
   pDacData->PeriodMax = PWMDAC_getPeriod(obj->pwmDacHandle[PWMDAC_Number_1]);

   pDacData->offset[0] = _IQ(0.0);
   pDacData->offset[1] = _IQ(0.5);
   pDacData->offset[2] = _IQ(0.5);
   pDacData->offset[3] = _IQ(0.5);

   pDacData->gain[0] = _IQ(1.0);
   pDacData->gain[1] = _IQ(1.0);
   pDacData->gain[2] = _IQ(1.0);
   pDacData->gain[3] = _IQ(1.0);

} // end of HAL_setDacParameters() function
```

Step 3, same as lab01b

In lab01c, we connect relevant variables to the PWMDAC module, and monitor the output waveform from the DACs of DRV8312_RevD kit + F28069M ControlCard, as shown in Figure 13.

Ch1-> PWMDAC1: Angle output of angle_gen
Ch2-> PWMDAC2: Iq reference
Ch3-> PWMDAC3: Iq feedback
Ch4-> Phase current from current probe

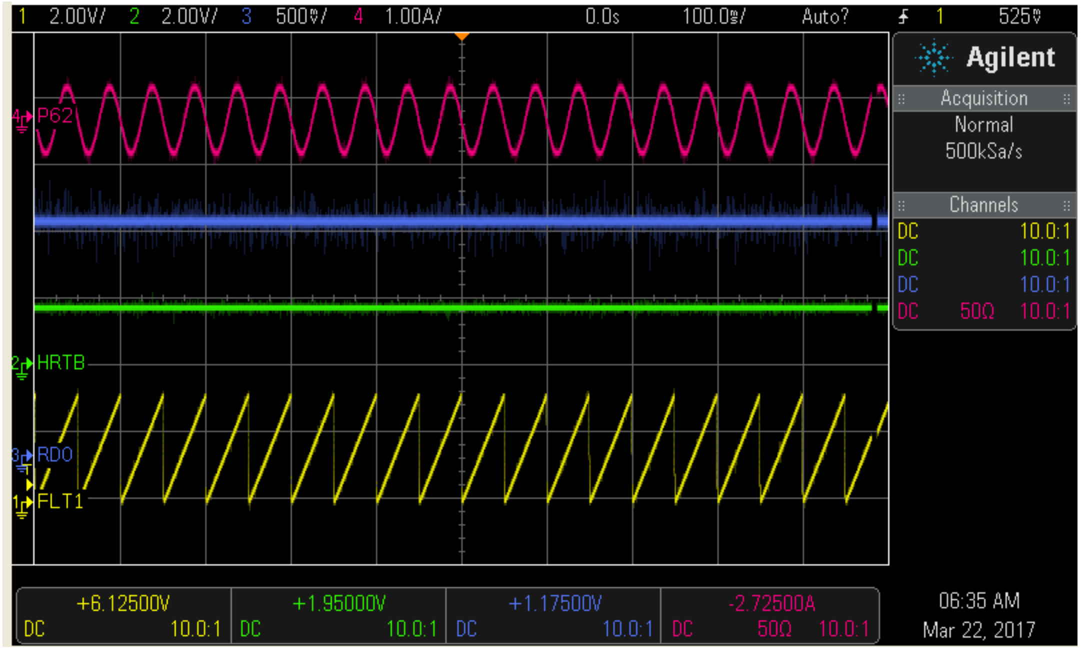


Figure 13 Iq reference, Iq feedback phase current and angle output waveform

DATALOG

As in lab01b, use the datalog and graph tools to monitor the input/output wavefrom in this project.

Step 1. Click Tools->Graph->Dual time in CCS menu. Click the Import button and select the appropriate single or dual time file from "..\sw\solutions\instaspin_foc\src\proj_lab01c_d1&d2_1.graphProp."

Step 2. Click Tools->Graph->Single time in CCS menu. Click the Import button and select the appropriate single or dual time file from "..\sw\solutions\instaspin_foc\src\proj_lab01c_d3_1.graphProp."

Step 3. Connect inputs of the datalog module; the datalog graph for SVGEN output is shown in Figure 14

```
// Connect inputs of the datalog module
datalog.iptr[0] = &angle_gen.Angle_pu;                  // datalogBuff[0]
datalog.iptr[1] = &controller_obj->pid_Iq.refValue;     // datalogBuff[1]
datalog.iptr[2] = &controller_obj->pid_Iq.fbackValue;   // datalogBuff[2]

datalog.Flag_EnableLogData = true;
datalog.Flag_EnableLogOneShot = false;
```

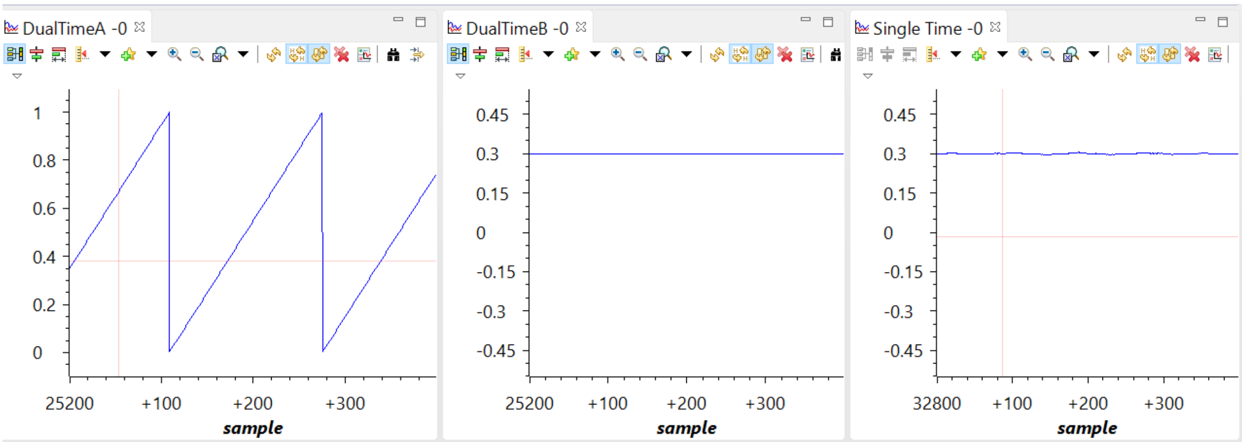


Figure 14 Iq reference, Iq feedback and angle output waveform

Project Files

No new project files in lab01c compared to lab 01b.

Includes

There are no new includes.

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 lab01b.

Background

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

mainISR

The main ISR calls very critical, time dependent functions that run the FOC. A block diagram for Lab01c is shown in Figure 15.

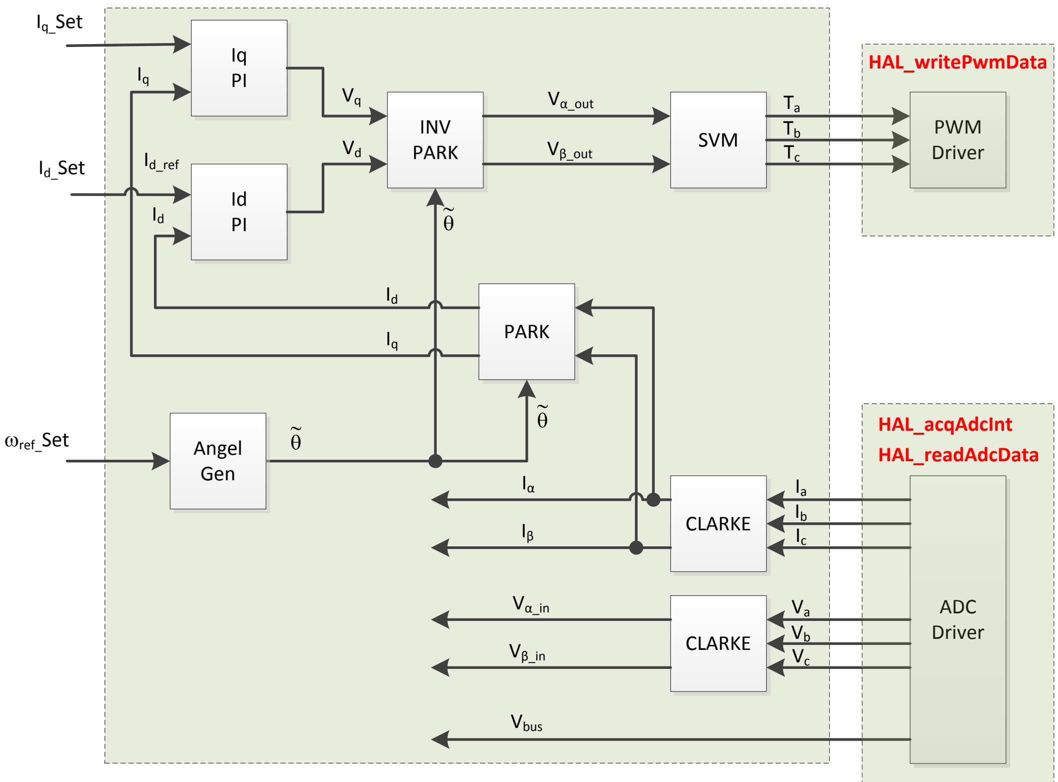


Figure 15 Block diagram of Volt/Hertz control for motor with added current closed loop

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 in this lab.

Step 1. Connect the motor, power on the EVM kit, start lab, and follow PWMDAC and DATALOG modules instruction to monitor some signals from SVGEN.

- 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 "gMotorVars.Flag_enableSys" equal to 1.
- Set the variable "gMotorVars.Flag_Run_Identify" equal to 1.
- Set the variable "gMotorVars.SpeedRef_krpm", "gMotorVars.IdSet_A" and "gMotorVars.IqSet_A" to a different value to ensure the motor runs.

Step 2. Power off the EVM kit, and follow PWMDAC and DATALOG modules instruction to change the configuration settings to monitor some signals from ANGEL_GEN and ADC.

- Set the variable "gMotorVars.Flag_Run_Identify" to 0 to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.
- Power off EVM kit

Conclusion

Lab 01c has demonstrated how to implement a current closed loop, open speed loop control to test the integrity of the signal chain hardware, with a new addition of current closed loop PI controllers. The intended consequence is verification of the signal chain integrity of the hardware abstraction layer (HAL) setup, specifically the ADC module for current sensing. In addition, this lab demonstrated use of the PWMDAC and Datalog functions to verify the PI, PWM, ADC, CLARK, PARK, I-PARK and SVGEN modules.

Lab 2a - Using InstaSPIN for the First Time out of ROM

Abstract

InstaSPIN-FOC and InstaSPIN-MOTION are FAST enabled self-sensored field oriented controllers. InstaSPIN-FOC offers cascaded speed control, while the InstaSPIN-MOTION speed controller features Active Disturbance Rejection Control (ADRC), which estimates and compensates for system disturbances in real time.

The ROM library contains the FAST observer plus all code needed to implement the FOC controller and speed loop. For this lab we start by using the full ROM based code to create an FOC motor control.

Introduction

Labs have been developed to cover the various uses of the on-chip motor control ROM. The library is very robust and can be customized to many different applications. The library can perform all of the sensorless estimation and the full cascaded FOC and speed control loop with very minimal external software or it can just provide the rotor flux angle estimation with the InstaSPIN-MOTION advanced speed controller. This lab implements the full InstaSPIN-FOC solution from ROM with the fewest number of library function calls. We will learn what enables the motor parameter identification and then how to start the motor.

Objectives Learned

- 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.

Background

Lab 2a adds the critical function calls for identifying and running a motor. The block diagram of Figure 16 shows the default closed loop functionality of the ROM library. This lab will create a full sensorless FOC drive with a minimum number of InstaSPIN function calls.

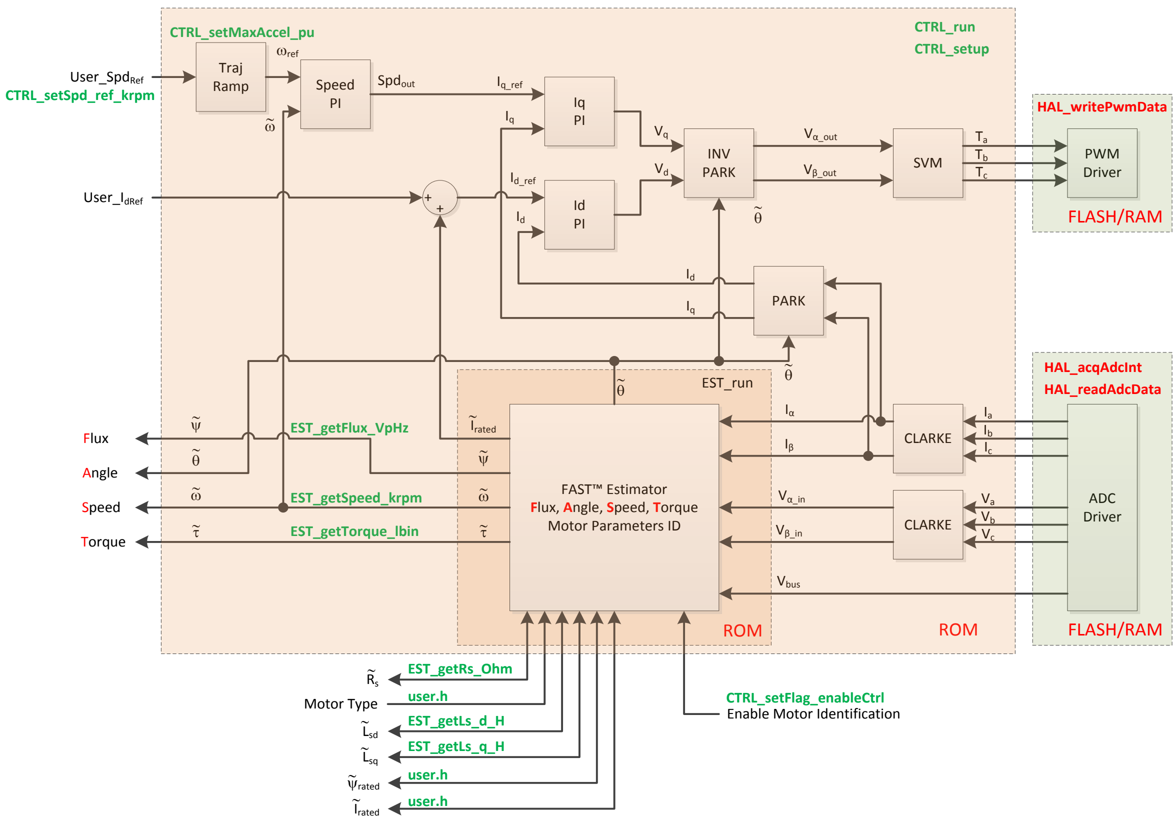


Figure 16: Block diagram for Lab 2a showing what blocks will be used to run an FOC, speed controlled, PMSM, motor. API functions that are in red are functions that were used in previous labs.

Includes

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

Table 9: Important header files needed for the motor control.

main.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.

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.

Table 10: Global object and variable declarations important for the motor control

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.

Initialization and Setup

This section covers functions needed to setup 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 11 are in the project for enhanced capability of the laboratory and not fundamentally needed to setup the motor control. For a more in depth explanation for definitions of the parameters and return values go to the document MotorWare section of this document (InstaSPIN-FOC and InstaSPIN-MOTION User's Guide, SPRUHJ1).

Table 11: Important setup functions needed for the motor control.

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.

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 12 lists the important functions used. The flowchart of Figure 17 shows the logic of the forever loop. The block called "Update Global Variables" is used to update variables such as speed, Stator Resistance, Inductance, etc.

Table 12: Functions used in the forever loop that check for errors and setup the InstaSPIN controller and FAST observer in real-time.

forever loop		
	CTRL	
	CTRL_isError	Check for errors throughout the state machine of the InstaSPIN controller and FAST observer.
	CTRL_setFlag_enableCtrl	Enables/Disables the sensorless controller. The first time this function is called and the function CTRL_setFlag_enableUserMotorParams() is not set TRUE, motor commissioning is performed. Then subsequent times it is called the motor will start to run.
	CTRL_updateState	Returns "true" if the controller's state machine has changed.
	CTRL_getState	Returns the state of the controller.
	CTRL_setMaxAccel_pu	Sets the maximum acceleration rate of the speed reference.
	CTRL_setSpd_ref_krpm	Sets the output speed reference value in the controller.
	HAL	
	HAL_updateAdcBias	It sets the voltage and current measurement offsets, before the motor is started.
	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_setMaxCurrentSlope_pu	Sets the maximum current slope for the Iq and Id reference change. If current isn't ramping as fast as needed, increase the value with this function call.

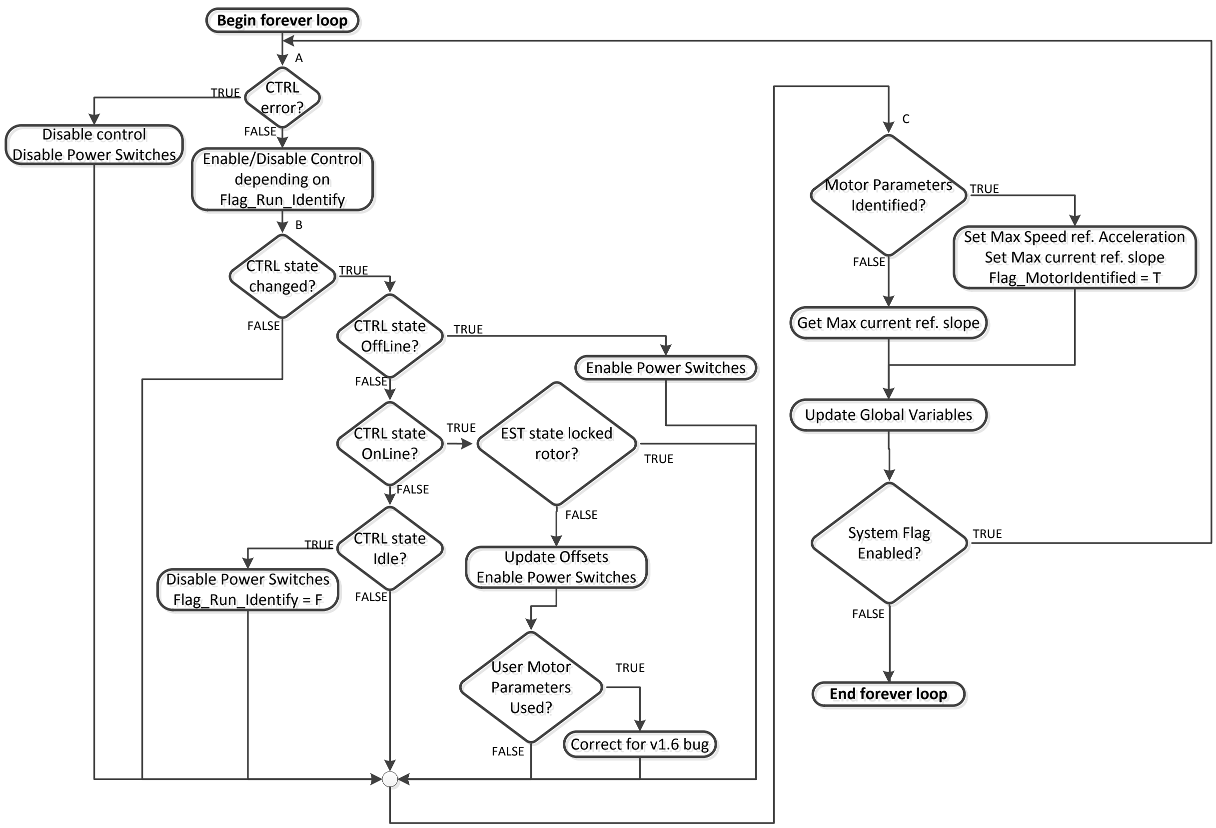


Figure 17: Forever loop flowchart.

Main ISR

The main ISR calls very critical, time dependent functions that run the FOC and FAST observer. The new functions that are required for this lab are listed in the Table below.

Table 13: InstaSPIN functions used in the main ISR.

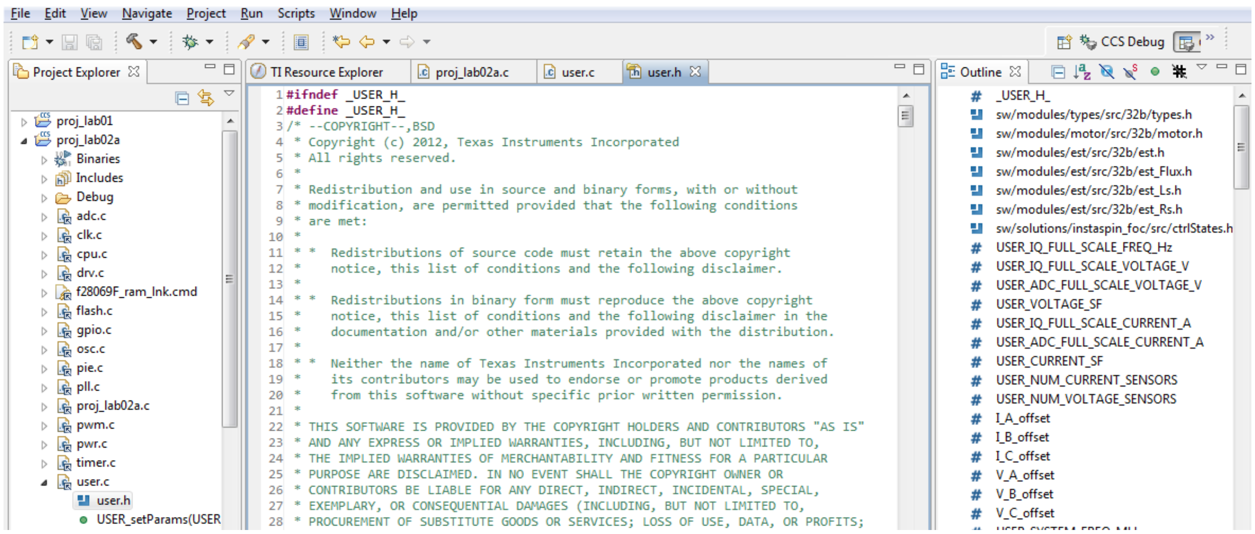
mainISR		
	CTRL	
	CTRL_run	The CTRL_run function implements the field oriented control. There are three parts to CTRL_run: CTRL_runOffline, CTRL_runOnline, and CTRL_runOnlineUser.
	CTRL_setup	Is responsible for updating the CTRL state machine and must be called in the same timing sequence as CTRL_run().

Lab Procedure

The code for Lab 2a is setup according to the flowchart shown in Figure 17: Forever loop flowchart. The HAL calls from Lab 1 will be used as a starting point and are shown in red. Extra function calls are added into the code to interface with the FAST library and are shown in green. The first step when running a motor with InstaSPIN is to fill the library with nameplate data from the motor. The first topic that needs to be covered before running any motor with InstaSPIN is the file “user.h”.

Open user.h following these steps:

5. Expand user.c from the Project Explorer window
6. Right-mouse click on user.h and select open, this opens the file user.c
7. Right-mouse click on the highlighted “user.h” and select “Open Declaration”, this opens user.h
8. Opening the Outline View will provide an outline of the user.h contents



In proj_lab02a, open user.h from the directory in MotorWare shown the figure above.

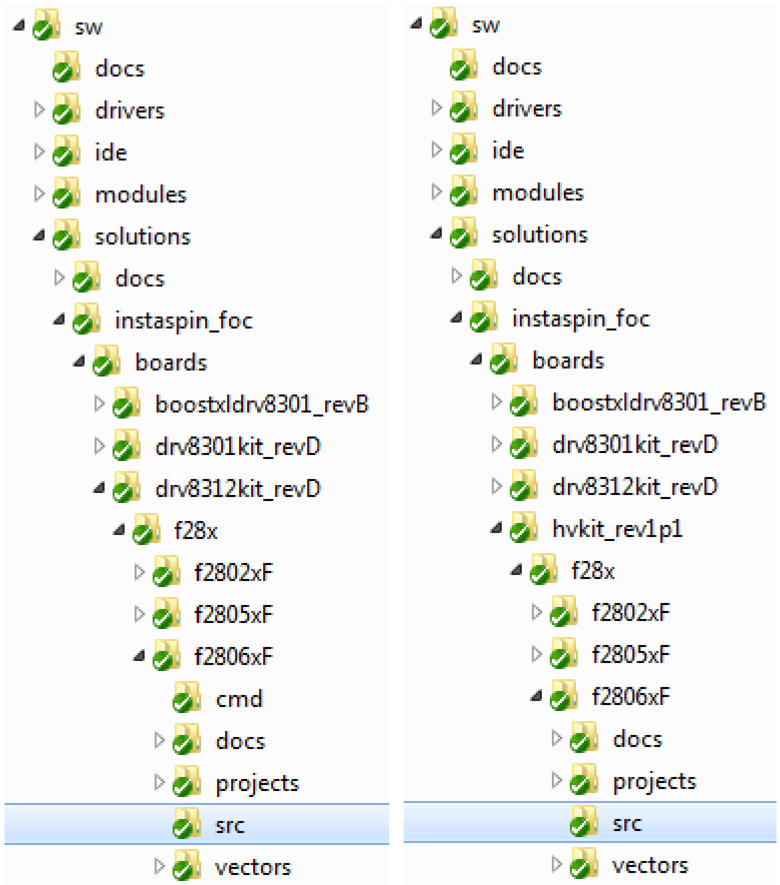


Figure 18: Directory to the file user.h for PMSM (left) and ACIM Motors (right).

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

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

```
#define MY_MOTOR              113
```

Comment out // the line that defines the current motor, which will look like the following:

```
#define USER_MOTOR HighCurrent_LowInductance
```

and add a line as shown below:

```
#define USER_MOTOR MY_MOTOR
```

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

```
#elif (USER_MOTOR == MY_MOTOR)
#define USER_MOTOR_TYPE                  MOTOR_Type_Pm
#define USER_MOTOR_NUM_POLE_PAIRS        (4)
#define USER_MOTOR_Rr                    (NULL)
#define USER_MOTOR_Rs                    (NULL)
#define USER_MOTOR_Ls_d                  (NULL)
#define USER_MOTOR_Ls_q                  (NULL)
#define USER_MOTOR_RATED_FLUX            (NULL)
#define USER_MOTOR_MAGNETIZING_CURRENT   (NULL)
#define USER_MOTOR_RES_EST_CURRENT       (1.0)
#define USER_MOTOR_IND_EST_CURRENT       (-1.0)
#define USER_MOTOR_MAX_CURRENT           (3.0)
#define USER_MOTOR_FLUX_EST_FREQ_Hz      (20.0)
```

And as below if it is an ACIM motor:

```
#elif (USER_MOTOR == MY_MOTOR)
#define USER_MOTOR_TYPE                  MOTOR_Type_Induction
#define USER_MOTOR_NUM_POLE_PAIRS        (4)
#define USER_MOTOR_Rr                    (NULL)
#define USER_MOTOR_Rs                    (NULL)
#define USER_MOTOR_Ls_d                  (NULL)
#define USER_MOTOR_Ls_q                  (NULL)
#define USER_MOTOR_RATED_FLUX            (0.8165*220.0/60.0)
#define USER_MOTOR_MAGNETIZING_CURRENT   (NULL)
#define USER_MOTOR_RES_EST_CURRENT       (1.0)
#define USER_MOTOR_IND_EST_CURRENT       (NULL)
#define USER_MOTOR_MAX_CURRENT           (3.0)
#define USER_MOTOR_FLUX_EST_FREQ_Hz      (5.0)
```

A few values can already be put into the user.h motor parameters.

- `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 name plate 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.

A spreadsheet was created to help setup user.h parameters based on motor parameters, control frequencies, filter poles, etc. The spreadsheet can be found in this folder:

C:\ti\motorware\motorware_1_01_00_18\docs\labs\motorware_selecting_user_variables.xlsx

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 this motor is defined to be a permanent magnet motor. The terms "Magnetizing Current" and "Rr" are not needed for a PM motor model 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 setup current, voltage scaling and filter parameters for the library.

Now, connect the motor that will be run with InstaSPIN to the kit. Insert the MCU control card. Connect the USB cable to the control card. Finally apply power to the kit. In Code Composer, build proj_lab02a. Start a Debug session and download the proj_lab02a.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 "gMotorVars" and is defined in main.h. A script has been written to easily add these variables to the watch window.

- 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 "sw\solutions\instaspin_foc\src\proj_lab02a.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 "gMotorVars.Flag_enableSys" is displayed as a character, right-mouse click on it and select "Number Format -> Decimal"
    - For the "Q-Value(24)" format, after right-mouse clicking on the value select "Q-Values->Q-Value(24)" from the pop-up menu

gMotorVars	{...}	struct _MOTOR_Vars_t_
gMotorVars.UserErrorCode	USER_ErrorCode_NoError	enum unknown
gMotorVars.CtrlVersion	{...}	struct _CTRL_Version_
gMotorVars.Flag_enableSys	0 (Decimal)	unsigned int
gMotorVars.Flag_Run_Identify	0 (Decimal)	unsigned int
gMotorVars.Flag_enableForceAngle	1 (Decimal)	unsigned int
gMotorVars.Flag_enablePowerWarp	0 (Decimal)	unsigned int
gMotorVars.CtrlState	CTRL_State_Idle	enum unknown
gMotorVars.EstState	EST_State_Idle	enum unknown
gMotorVars.SpeedRef_krpm	0.09999996424 (Q-Value(24))	long
gMotorVars.MaxAccel_krpmps	0.1999999881 (Q-Value(24))	long
gMotorVars.Speed_krpm	0.0 (Q-Value(24))	long
gMotorVars.Torque_lbin	0.0	float
gMotorVars.MagnCurr_A	0.0	float
gMotorVars.Rr_Ohm	0.0	float
gMotorVars.Rs_Ohm	0.0	float
gMotorVars.Lsd_H	0.0	float
gMotorVars.Lsq_H	0.0	float
gMotorVars.Flux_VpHz	0.0	float
gMotorVars.I_bias	{...}	struct _MATH_vec3_
gMotorVars.V_bias	{...}	struct _MATH_vec3_
gMotorVars.VdcBus_kV	0.0 (Q-Value(24))	long
gDrvSpi8301Vars	Error: identifier not found: ...	unknown

- 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 “gMotorVars.Flag_enableSys” equal to 1.
- Set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

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 “gMotorVars.Flag_Run_Identify” is equal to 0, and we are identifying a PMSM motor, then the motor parameters have been identified. If we are identifying an ACIM motor, the controller and estimator states will show the following states:

gMotorVars.CtrlState	CTRL_State_Idle
gMotorVars.EstState	EST_State_LockRotor

If this is the case, then lock the rotor, and enable the controller again by setting “gMotorVars.Flag_Run_Identify” to 1. Once the “gMotorVars.Flag_Run_Identify” is equal to 0, 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 = ` gMotorVars.Rr_Ohm's value (ACIM motors only)
  - `USER_MOTOR_Rs = ` gMotorVars.Rs_Ohm's value
  - `USER_MOTOR_Ls_d = ` gMotorVars.Lsd_H's value
  - `USER_MOTOR_Ls_q = ` gMotorVars.Lsq_H's value
  - `USER_MOTOR_RATED_FLUX = ` gMotorVars.Flux_VpHz's value
  - `USER_MOTOR_MAGNETIZING_CURRENT = ` gMotorVars.MagnCurr_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,

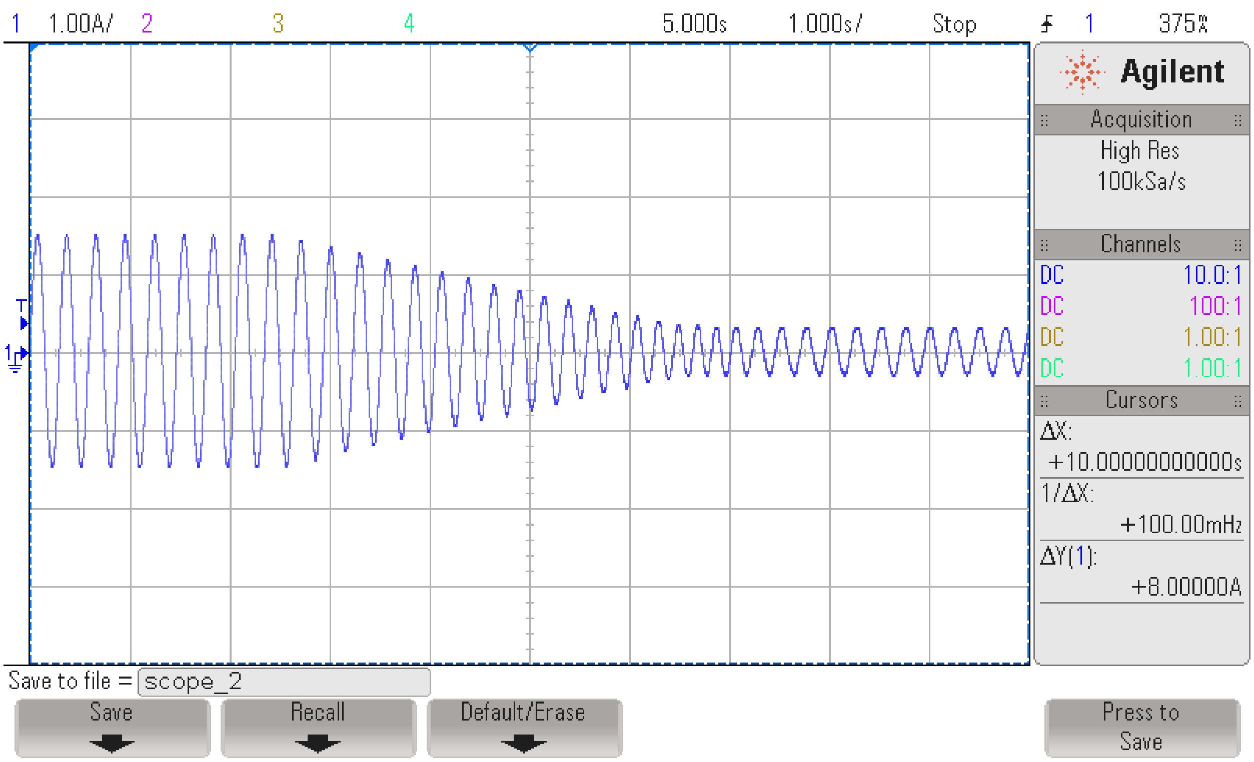
- Set the variable "gMotorVars.Flag_Run_Identify" equal to 1 again.

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 kilo-rpm) by the variable "gMotorVars.Speed_krpm". The target speed reference is (in kilo-rpm) set by the variable "gMotorVars.SpeedRef_krpm".

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

If this is an ACIM motor, you might want to experiment with PowerWarp. Once the motor is running in closed loop, enable power Warp by setting "gMotorVars.Flag_enablePowerWarp" flag to 1. The following oscilloscope plot shows how the current consumed by the motor goes from rated magnetizing current to a minimum calculated by the PowerWarp algorithm.

TI Spins Motors



Notice that when changing between speeds, the motor shaft speed does not change instantaneously. An acceleration trajectory is setup 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 "gMotorVars.MaxAccel_krpmps".

When done experimenting with the motor,

- Set the variable "gMotorVars.Flag_Run_Identify" to 0 to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

API functions used in the watch window during this lab are shown in the table below.

Table 14: Functions used to interface with the watch window during this lab.

updateGlobalVariables		
	CTRL	
	CTRL_setSpd_ref_krpm	Sets the output speed reference value in the controller in kilo-rpm.
	CTRL_setMaxAccel_pu	Sets the maximum acceleration rate of the speed reference.
	CTRL_getState	Gets the controller state.
	EST	
	EST_getSpeed_krpm	Gets the speed value in kilo-rpm.
	EST_getTorque_lbin	Gets the torque value in lb-in.
	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_VpHz	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.

Conclusion

Lab 2a has demonstrated the basics of using the InstaSPIN library and running it out of ROM. 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.

Lab 2b – Using InstaSPIN out of User RAM and/or FLASH

Abstract

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

Introduction

In the previous lab (lab 2a) the full InstaSPIN-FOC motor control software was executed out of ROM. This lab will also implement the InstaSPIN motor control software but it will be done with open source code that is executed out of user RAM. The only code that has to remain closed is the FAST observer.

Objectives Learned

- Include open source code for the sensorless FOC.
- How the FAST observer is initialized and setup.
- How to run the FAST observer.

Background

A block diagram is shown below of the minimal ROM implementation of InstaSPIN-FOC. InstaSPIN-MOTION can use the same implementation. All of the same functions that were used in lab 2a are also used in this lab. If any functions are re-used from a previous lab, they will be highlighted in red. The new file that is included in lab 2b's project is "ctrl.c". The ctrl.c file contains the same control code that is in the ROM of the 2802xF, 2805xF/M and 2806xF/M.

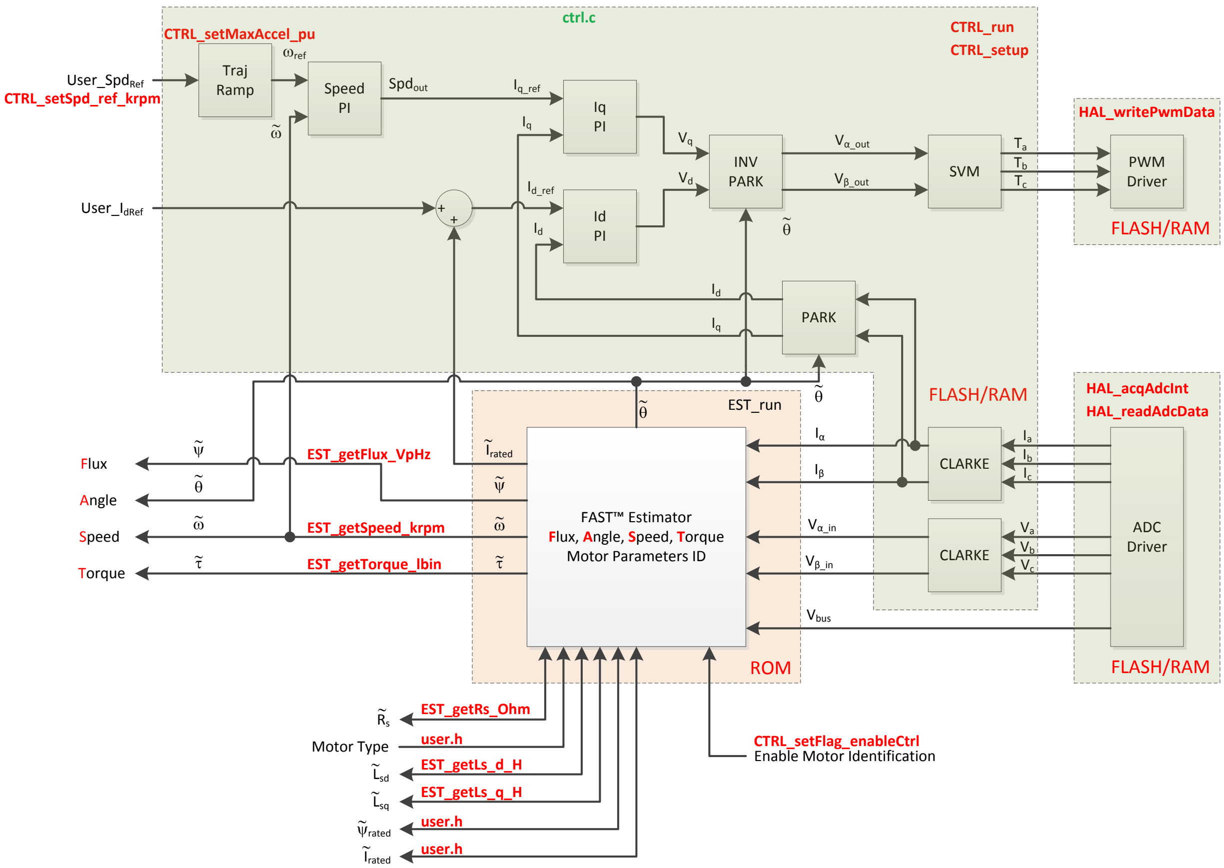


Figure 19: Block diagram of only FAST in ROM with the rest of InstaSPIN-FOC in user memory

Project Files

Because most of the InstaSPIN-FOC code has moved from ROM to user memory, more files must be added to the project as compared to lab 2a. Table 15 lists the new files in the project. Note that many of the functions are located in the header file associated with the C-file, such as: clark.h, ipark.h, park.h, svgen.h and traj.h.

Table 15: New files that must be included in the project for user's RAM.

proj_lab02b		
	clarke.c	Contains the Clarke transform code.
	ctrl.c	Contains code for CTRL_run and CTRL_setup, which is the code that runs the FOC.
	filter_fo.c	Contains code for a first order filter used for offset calibration.
	ipark.c	Contains the inverse Park transform code.
	offset.c	Contains the offset code used to find voltage and current feedback offsets.
	park.c	Contains the Park transform code.
	svgen.c	Contains the space vector generator code.
	traj.c	Contains code for creating ramp functions.

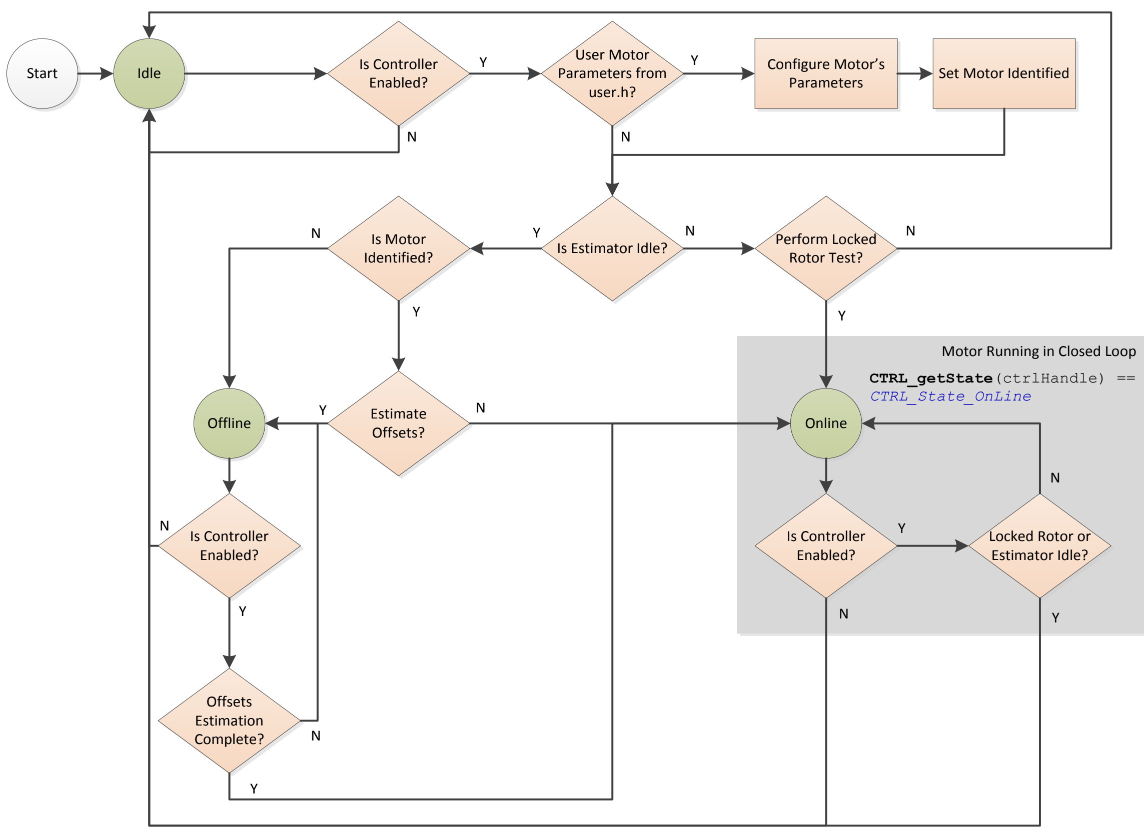


Figure 20: State diagram showing the three states, Idle, Offline, and Online, for CTRL_run.

Now is a good time to talk about the CTRL_run state machine shown in the figure above. There are three states in CTRL_run: Idle, Offline, and Online. The Idle state normally happens when the controller is shutdown. CTRL is in the Offline state during offset calibration and motor parameter identification. The Online state happens when the control and motor are running and all identification has finished.

The most important part of the code happens during the Online state, where the function `CTRL_run` is called from the mainISR(). As seen below, the CTRL and HAL handles, and the ADC and PWM data are passed to the CTRL_run() function.

proj_lab02c.c

```
interrupt void mainISR(void)
{
  // toggle status LED
  if(gLEDcnt++ > (uint_least32_t)(USER_ISR_FREQ_Hz / LED_BLINK_FREQ_Hz))
  {
    HAL_toggleLed(halHandle,(GPIO_Number_e)HAL_Gpio_LED2);
    gLEDcnt = 0;
  }


  // acknowledge the ADC interrupt
  HAL_acqAdcInt(halHandle,ADC_IntNumber_1);


  // convert the ADC data
  HAL_readAdcData(halHandle,&gAdcData);


  // run the controller
  CTRL_run(ctrlHandle,halHandle,&gAdcData,&gPwmData);


  // write the PWM compare values
  HAL_writePwmData(halHandle,&gPwmData);


  // setup the controller
  CTRL_setup(ctrlHandle);


  if(CTRL_getMotorType(ctrlHandle) == MOTOR_Type_Pm)
    {
      // reset Ls Q format to a higher value when Ls identification starts
      CTRL_resetLs_qFmt(ctrlHandle, gMax_Ls_qFmt);
    }

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

The code lines below are part of `CTRL_run`, all of the code for `CTRL_run` can be found in the file ctrl.c. The if-then statements below show that when the motor is being identified `CTRL_runOnLine` is executed from ROM. After the motor is identified, `CTRL_runOnLine_User` is run from user RAM. `CTRL_runOnLine_User` is an inlined function that is located in ctrl.h. It contains the entire FOC implementation with calls to the FAST observer for rotor flux angle values.

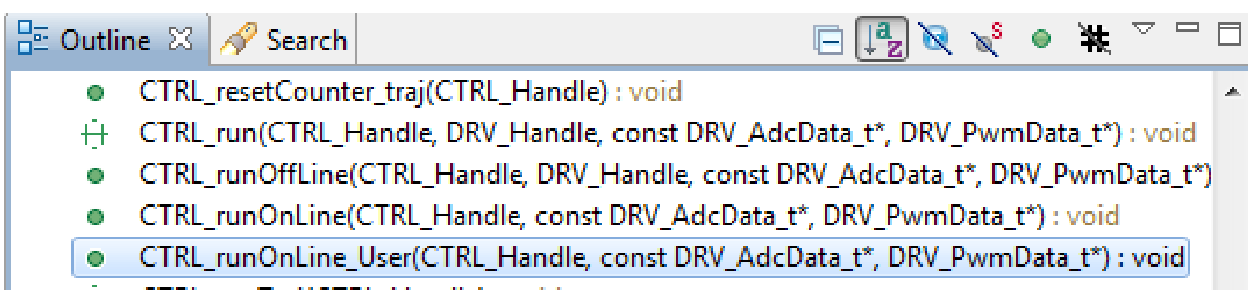
proj_lab02c.c | ctrl.c | ctrl.h

```c

    // run the appropriate controller
    if(ctrlState == CTRL_State_OnLine)
      {
        CTRL_Obj *obj = (CTRL_Obj *)handle;

        // increment the current count
        CTRL_incrCounter_current(handle);

        // increment the speed count
        CTRL_incrCounter_speed(handle);

        if(EST_getState(obj->estHandle) >= EST_State_MotorIdentifie
          {
            // run the online controller
            CTRL_runOnLine_User(handle,pAdcData,pPwmData);
          }
        else
          {
            // run the online controller
            CTRL_runOnLine(handle,pAdcData,pPwmData);
          }
      }
```

Executes from user memory (points to `CTRL_runOnLine_User(handle,pAdcData,pPwmData);`)

Executes from ROM memory during Motor ID (points to `CTRL_runOnLine(handle,pAdcData,pPwmData);`)

With the file ctrl.h open and selecting Outline View within CCS (View->Outline), you can select `CTRL_runOnLine_User` from the Outline window to review the code in the source window.



proj_lab02c.c | ctrl.c | ctrl.h

```
inline void CTRL_runOnLine_User(CTRL_Handle handle,
                            const DRV_AdcData_t *pAdcData,DRV_PwmData_t *pPwmData)
{
  CTRL_Obj *obj = (CTRL_Obj *)handle;

  _iq angle_pu;

  MATH_vec2 phasor;


 // run Clarke transform on current
 CLARKE_run(obj->clarkeHandle_I,&pAdcData->I,CTRL_getIab_in_addr(handle));


 // run Clarke transform on voltage
 CLARKE_run(obj->clarkeHandle_V,&pAdcData->V,CTRL_getVab_in_addr(handle));


 // run the estimator
 EST_run(obj->estHandle,CTRL_getIab_in_addr(handle),CTRL_getVab_in_addr(handle),
         pAdcData->dcBus,TRAJ_getIntValue(obj->trajHandle_spd));


 // generate the motor electrical angle
 angle_pu = EST_getAngle_pu(obj->estHandle);


 // compute the sin/cos phasor
 CTRL_computePhasor(angle_pu,&phasor);

```

Includes

There are no new includes after lab 2a.

Global Object and Variable Declarations

There are no new global object and variable declarations after lab 2a.

Initialization and Setup

Nothing has changed between lab 2a and this lab for initialization and setup.

Main Run-Time loop (forever loop)

The forever loop remains the same as lab 2a.

Main ISR

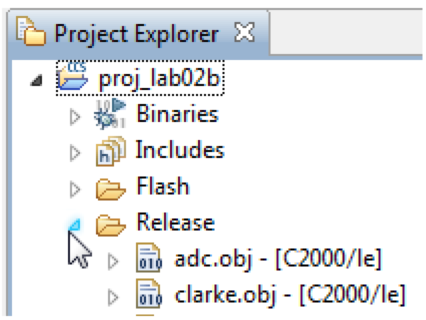
Nothing has changed from lab 2a, only now the code is run out of RAM.

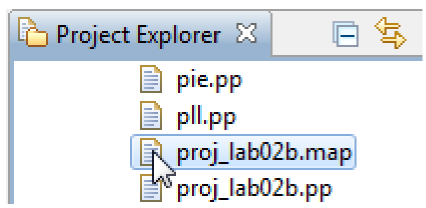
Lab Procedure

The code for Lab 2b is setup according to the diagram as shown in the figure below. Notice that all of the function calls are shown in red. As the code appears from main.c, there is no difference between running the code from ROM (lab 2a) or RAM (lab 2b).

For this lab procedure, it will be good to look through the map file and see what functions are still in ROM and the others that are run from RAM. First we must generate the MAP file by building the code.

- In Code Composer, build proj_lab02b.
- In the project explorer window, expand proj_lab02b→Flash or proj_lab02b→Release depending on the build configuration selected.
  - Double click to open the file proj_lab02b.map.





Looking at the proj_lab02b.map file section "GLOBAL SYMBOLS: SORTED BY Symbol Address", there are many CTRL functions that are located in the .text section of RAM or between addresses 0x08000 – 0x010000. If the map listing file for proj_lab02a is opened, there are no CTRL symbols located in user memory and hence are all called out of ROM. However, the map file for proj_lab02b shows the CTRL function calls being loaded and executed in user's memory as shown below:

```
00008948   _HAL_enableAdcInts
0000895a   _HAL_disableGlobalInts
0000895e   _CTRL_updateState
00008a10   _CTRL_setupClarke_V
00008a26   _CTRL_setupClarke_I
00008a42   _CTRL_setup
00008a4c   _CTRL_setWaitTimes
00008a55   _CTRL_setSpd_ref_pu
00008a58   _CTRL_setSpd_ref_krpm
00008a6b   _CTRL_setGains
00008a93   _CTRL_setParams
00008cd0   _CTRL_run
```

To run this lab, follow the same exact procedure from lab 2a.

Conclusion

Most of the InstaSPIN code is open source. Lab 2b has demonstrated how to run FOC out of user RAM with minimal function calls into ROM. The only part of InstaSPIN that remains in ROM is the FAST observer and functions used during motor commissioning (identification).

InstaSPIN-MOTION provides several components in ROM, including inertia identification, the speed controller, and the motion profile generator. These components will be covered in the upcoming labs.

Lab 2c – Using InstaSPIN to Identify Low Inductance PMSM

Abstract

This lab implements a few additional functions compared to previous lab 2b, so that PMSM motors with low inductance can be identified.

Introduction

When identifying low inductance PMSM motors, special considerations need to be taken. These considerations are related to how the control loops are setup, so that even with low inductance, the control loops are stable. This lab implements and utilizes new functions to allow identification of low inductance PMSM motors.

Objectives Learned

- Utilize additional functions to identify low inductance PMSM motors.
- Run the FAST with low inductance PMSM motors.
- Configure the current control loops when identifying and running low inductance PMSM motors.

Background

The same block diagram from lab 2b applies to lab 2c.

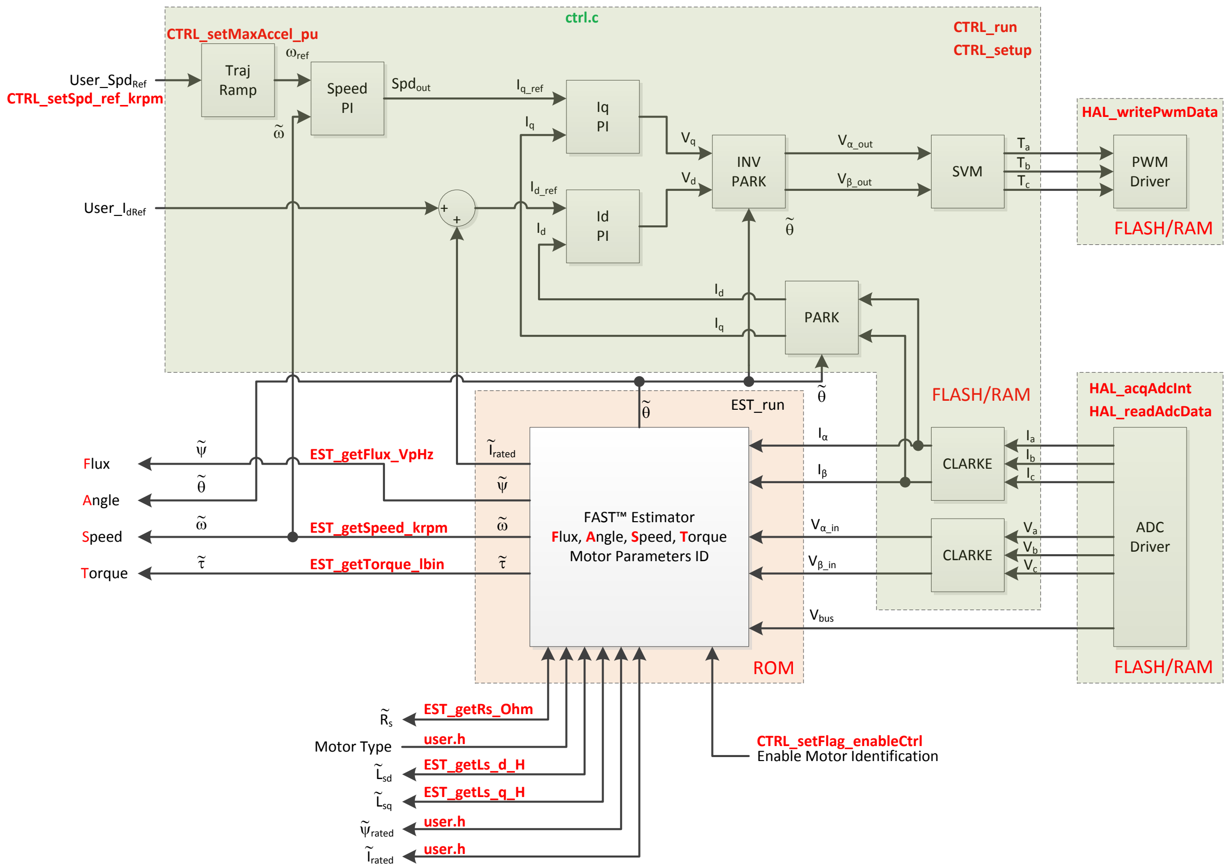


Figure 21: Block diagram of only FAST in ROM with the rest of InstaSPIN-FOC in user memory

Project Files

No new project files in lab 2c compared to lab 2b.

Includes

There are no new includes.

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 lab.

Main Run-Time loop (forever loop)

Table 16: New API function calls.

forever loop		
	proj_lab2c.c	
	CTRL_recalcKpKi	This function recalculates gains of the current control loops as well as the speed control loop during motor identification to allow low inductance motors to identify and run
	CTRL_calcMax_Ls_qFmt	This function calculates the correct Q format of the inductance to be identified. This is required for low inductance PMSM motors
main ISR		
	proj_lab2c.c	
	CTRL_resetLs_qFmt	This function uses the Q format calculated in function CTRL_calcMax_Ls_qFmt when inductance identification begins

Lab Procedure

To build, load and run the code, follow this procedure:

- Build and load program by clicking on the debug icon
- 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 "gMotorVars.Flag_enableSys" equal to 1.
- Set the variable "gMotorVars.Flag_Run_Identify" equal to 1.

After setting these two flags, the motor will start identifying, and motor parameters will be displayed in the watch window.

When identifying low inductance motors, it is possible that the PWM frequency needs to be increased to allow a better current control by minimizing current ripple. In general, 45 kHz should be high enough to identify any inductance:

```
#define USER_PWM_FREQ_kHz   (45.0)
```

If the current ripple is still too high, which can be checked with a current probe and an oscilloscope, consider increasing this frequency up to 60 kHz in 5 kHz increments.

```
#define USER_PWM_FREQ_kHz   (60.0)
```

Also, if the motor has too much cogging torque at low speeds, which is common in high speed motors, consider increasing the frequency at which the flux and inductance is identified. This frequency is set to 20 Hz by default, but can be increased.

```
#define USER_MOTOR_FLUX_EST_FREQ_Hz  (20.0)
```

If the motor is not running smoothly while identifying the motor, or if the inductance coming back from the identification process varies too much from run to run, increase this frequency in increments of 20, up to 100 Hz or so if needed.

```
#define USER_MOTOR_FLUX_EST_FREQ_Hz  (100.0)
```

Lastly, low inductance motors usually have low flux values also. Make sure that the full scale voltage is set to a value that allows the flux to be identified. Consider this formula when selecting full scale voltage:

Min Flux in V/Hz = USER_IQ_FULL_SCALE_VOLTAGE_V/USER_EST_FREQ_Hz/0.7

Lab 2d – Using InstaSPIN out of User Memory, with fpu32

Abstract

This lab runs Lab 2b with floating point unit enabled. This lab only applies to 6x devices, as it has a floating point unit.

Objectives Learned

Run motor identification lab with fpu32 enabled.

Lab Procedure

Follow the exact same procedure as in Lab 2b.

Conclusion

We conclude that the libraries in ROM also work when fpu32 is enabled in 6x devices.

Lab 3a – Using your own motor parameters

Abstract

By default, when InstaSPIN starts it first identifies the motor attached to the inverter if that motor has not been previously identified. When identifying the motor, the parameters Rs, Ls, and air gap flux are estimated. This lab will take the motor parameter estimates from the previous lab and place them into the file user.h. User.h is used to hold scaling factors, motor parameters, and inverter parameters for customizing InstaSPIN to any motor control system. The software will be modified to indicate that the motor has been previously identified such that the identification step can be skipped.

Introduction

The motor commissioning stage is very useful, but does take a long time to perform. Motor Parameters only have to be measured once. This lab shows how to use the motor parameters previously identified in lab 2 and then how to save those motor parameters in the file user.h.

Objectives Learned

- Bypassing motor identification and loading motor parameters from user.h.
- Storing the voltage and current offsets for future bypassing.

Background

Lab 3a adds the function call for setting the required flag to bypass motor identification. This lab also has function calls for saving the current and voltage offsets to the user gMotorVars structure, viewable from the Watch Window. Figure 22 shows the block diagram for running FAST with an external FOC with a listing of the new API functions used in this lab.

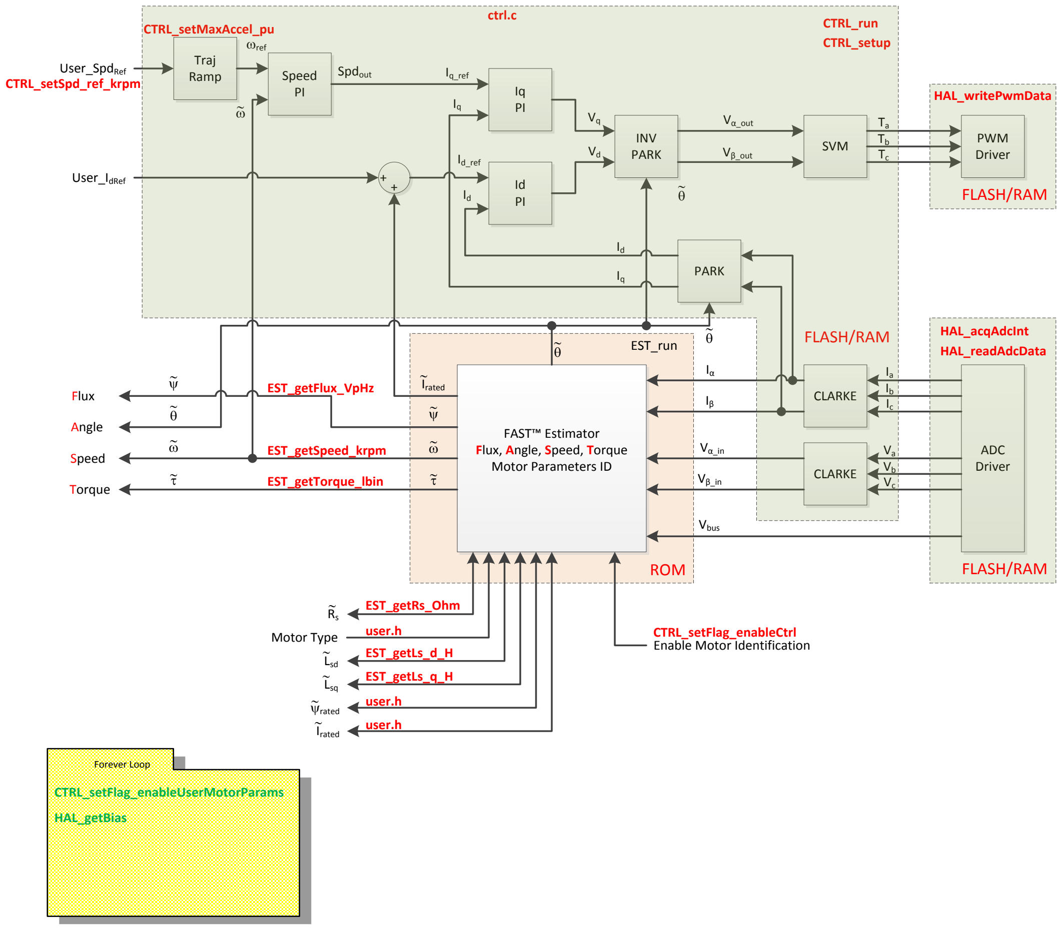


Figure 22: Block diagram of only FAST in ROM with the rest of InstaSPIN-FOC in user memory

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

Nothing has changed in initialization and setup from the previous lab.

Main Run-Time loop (forever loop)

Table 17: New API function calls.

forever loop		
	CTRL	
	CTRL_setFlag_enableUserMotorParams	Sending a true in this function's parameters will cause the controller to read motor parameters from user.h and not perform the motor commissioning.
	HAL	
	HAL_getBias	Returns the current and voltage offsets that are inherent in the current and voltage feedback circuits of the inverter.

Main ISR

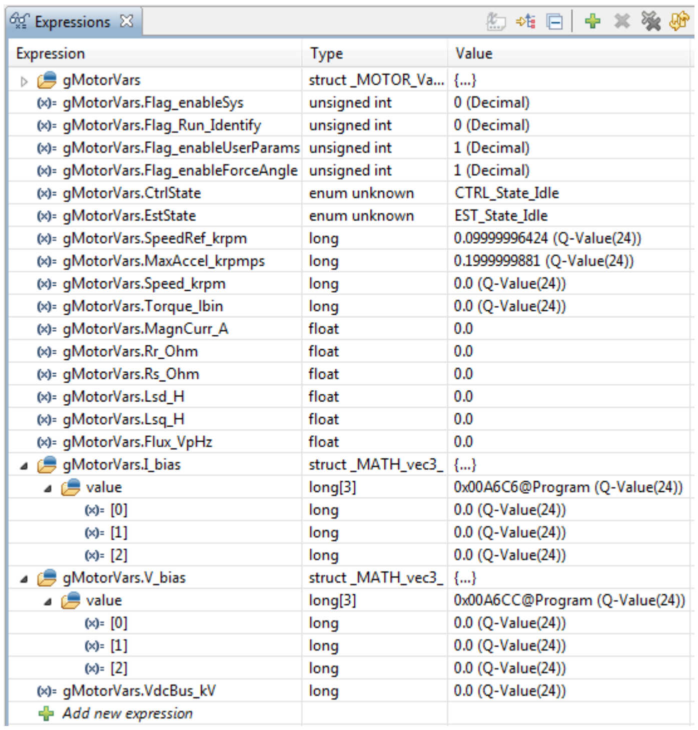
Nothing has changed in this section of the code from the previous lab.

Lab Procedure

After verifying that user.h has been properly updated with your identified motor parameters, build lab3a, connect to the target and load the .out file.

- Add the appropriate watch window variables by calling the script “proj_lab03a.js”.
- Enable the real-time debugger.
- Click the run button.
- Enable continuous refresh on the watch window.

The watch window should look like the figure below. Make sure that number formats are correct. For example, the gMotorVars.I_bias array values are in Q24 format.


Expressions

Expression	Type	Value
gMotorVars	struct _MOTOR_Va...	{...}
gMotorVars.Flag_enableSys	unsigned int	0 (Decimal)
gMotorVars.Flag_Run_Identify	unsigned int	0 (Decimal)
gMotorVars.Flag_enableUserParams	unsigned int	1 (Decimal)
gMotorVars.Flag_enableForceAngle	unsigned int	1 (Decimal)
gMotorVars.CtrlState	enum unknown	CTRL_State_Idle
gMotorVars.EstState	enum unknown	EST_State_Idle
gMotorVars.SpeedRef_krpm	long	0.09999996424 (Q-Value(24))
gMotorVars.MaxAccel_krpmps	long	0.1999999881 (Q-Value(24))
gMotorVars.Speed_krpm	long	0.0 (Q-Value(24))
gMotorVars.Torque_lbin	long	0.0 (Q-Value(24))
gMotorVars.MagnCurr_A	float	0.0
gMotorVars.Rr_Ohm	float	0.0
gMotorVars.Rs_Ohm	float	0.0
gMotorVars.Lsd_H	float	0.0
gMotorVars.Lsq_H	float	0.0
gMotorVars.Flux_VpHz	float	0.0
gMotorVars.I_bias	struct _MATH_vec3_	{...}
value	long[3]	0x00A6C6@Program (Q-Value(24))
[0]	long	0.0 (Q-Value(24))
[1]	long	0.0 (Q-Value(24))
[2]	long	0.0 (Q-Value(24))
gMotorVars.V_bias	struct _MATH_vec3_	{...}
value	long[3]	0x00A6CC@Program (Q-Value(24))
[0]	long	0.0 (Q-Value(24))
[1]	long	0.0 (Q-Value(24))
[2]	long	0.0 (Q-Value(24))
gMotorVars.VdcBus_kV	long	0.0 (Q-Value(24))
Add new expression		

Figure 23: Watch Window before startup.

Verify that the bias values are 0.0.

Before starting let’s cover what will occur in this lab and the variables to watch. After “Flag_enableSys” and “Flag_Run_Identify” are set true, the following motor events will happen:

- Voltage and current offsets (bias) are measured
- Rs value of the motor is re-estimated

In the next procedure, once “Flag_Run_Identify” is set true, watch how the bias values are updated and then immediately afterwards the Rs_Ohm value is re-estimated.

Procedure:

- Make sure gMotorVars.Flag_enableUserParams is set to 1, which should be the default value for this lab

(x)= gMotorVars.Flag_enableUserParams	unsigned int	1 (Decimal)

- In the Watch Window set “Flag_enableSys” to true (value of 1), and then set “Flag_Run_Identify” to true. Note the bias values are updated and then Rs_Ohm is re-estimated. The Watch Window will look something like the figure below.

Expressions

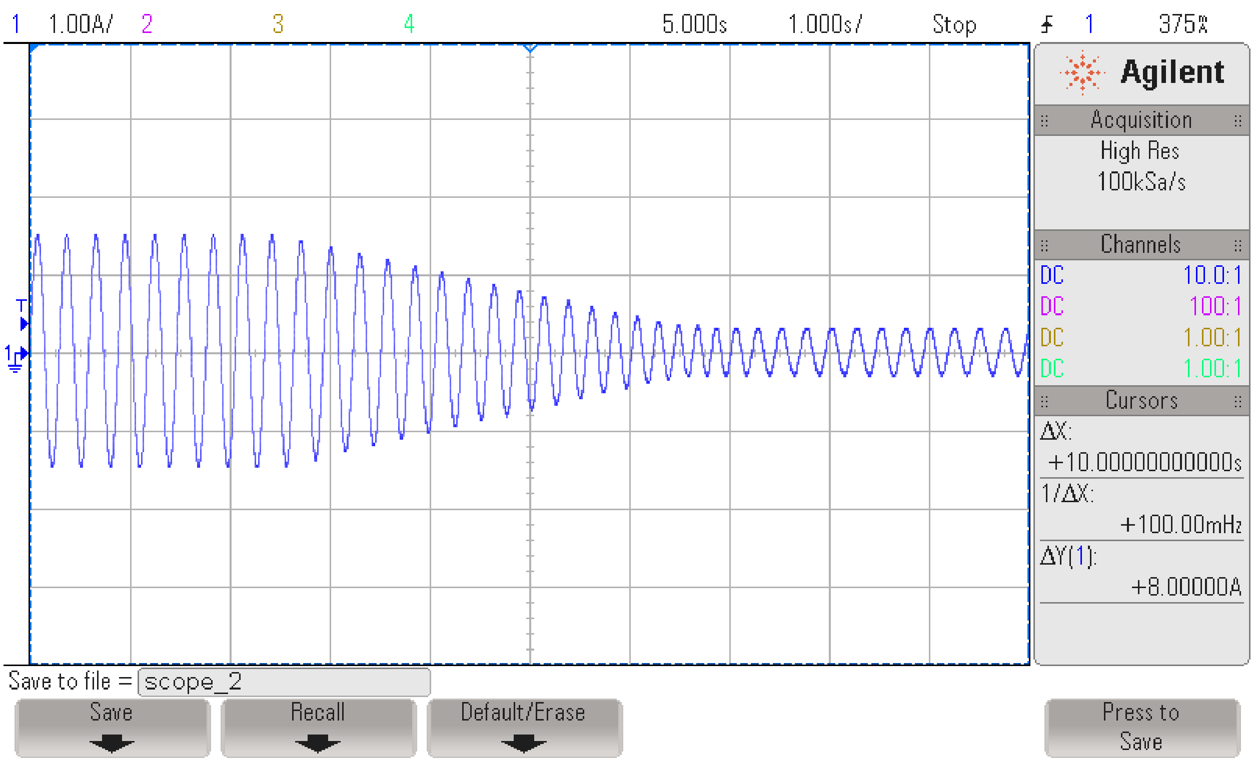
Expression	Type	Value
gMotorVars	struct _MOTOR_Va...	{...}
(x)= gMotorVars.Flag_enableSys	unsigned int	1 (Decimal)
(x)= gMotorVars.Flag_Run_Identify	unsigned int	1 (Decimal)
(x)= gMotorVars.Flag_enableUserParams	unsigned int	1 (Decimal)
(x)= gMotorVars.Flag_enableForceAngle	unsigned int	1 (Decimal)
(x)= gMotorVars.CtrlState	enum unknown	CTRL_State_OnLine
(x)= gMotorVars.EstState	enum unknown	EST_State_OnLine
(x)= gMotorVars.SpeedRef_krpm	long	0.09999996424 (Q-Value(24))
(x)= gMotorVars.MaxAccel_krpmps	long	0.1999999881 (Q-Value(24))
(x)= gMotorVars.Speed_krpm	long	0.09925693274 (Q-Value(24))
(x)= gMotorVars.Torque_lbin	long	0.01241630316 (Q-Value(24))
(x)= gMotorVars.MagnCurr_A	float	0.0
(x)= gMotorVars.Rr_Ohm	float	0.0
(x)= gMotorVars.Rs_Ohm	float	0.3987606
(x)= gMotorVars.Lsd_H	float	0.0006398709
(x)= gMotorVars.Lsq_H	float	0.0006398709
(x)= gMotorVars.Flux_VpHz	float	0.03465718
gMotorVars.I_bias	struct _MATH_vec3_	{...}
value	long[3]	0x00A6C6@Program (Q-Value(24))
(x)= [0]	long	0.8700211048 (Q-Value(24))
(x)= [1]	long	0.8719879985 (Q-Value(24))
(x)= [2]	long	0.8682925701 (Q-Value(24))
gMotorVars.V_bias	struct _MATH_vec3_	{...}
value	long[3]	0x00A6CC@Program (Q-Value(24))
(x)= [0]	long	0.4943121076 (Q-Value(24))
(x)= [1]	long	0.4900337458 (Q-Value(24))
(x)= [2]	long	0.4849975705 (Q-Value(24))
(x)= gMotorVars.VdcBus_kV	long	0.0236555934 (Q-Value(24))
Add new expression		

Figure 24: The Watch Window after bias calculation and Rs re-calibration.

- Next copy and paste the I_bias and V_bias values from the Watch Window into the corresponding #define statements in the file user.h, as shown below.

```
//! \brief ADC current offsets for A, B, and C phases
#define    I_A_offset    (0.8658943772)
#define    I_B_offset    (0.8676848412)
#define    I_C_offset    (0.8632109165)

//! \brief ADC voltage offsets for A, B, and C phases
#define    V_A_offset    (0.06220561266)
#define    V_B_offset    (0.06224888563)
#define    V_C_offset    (0.06216460466)
```

- Verify that the drive is functioning by changing the “gMotorVars.SpeedRef_krpm” value and making sure the motor is spinning smoothly.
- If this is an ACIM motor, you might want to experiment with PowerWarp. Once the motor is running in closed loop, enable power Warp by setting “gMotorVars.Flag_enablePowerWarp” flag to 1. The following oscilloscope plot shows how the current consumed by the motor goes from rated magnetizing current to a minimum calculated by the PowerWarp algorithm.



When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.

Turn off real-time control and stop the debugger.

Conclusion

This lab showed how to bypass the motor identification process and load motor parameters from user.h. The inverter current and voltage offsets were measured, read, and copied into user.h for use in the next lab where a complete customization of the InstaSPIN software is done to work with the inverter and completely eliminate any calibration before motor startup.

Lab 3b – Using your own Board Parameters: Current and Voltage Offsets (user.h)

Abstract

Skipping auto-calibration and using your own current and voltage offsets continues to reduce the start-up time. If the board offsets are known, then auto-calibration at start-up is not needed. Also introduced is the option to bypass the Rs Fine Re-estimation.

Introduction

At the very beginning immediately after InstaSPIN is enabled, an offset calibration takes place. The offsets for the current and voltage feedback circuits are measured and recorded. These offsets are subtracted from the ADC measurements during motor operation. During normal operation of the motor and drive, these offsets do not change much. In this lab, the offsets that were measured in lab 3a are stored in the user.h file, further reducing the amount of time needed before starting the motor.

Objectives Learned

- Write current and voltage offsets to the HAL object to reduce calibration time before motor startup.
- Bypass Rs fine re-estimation (also known as Rs recalibration) which also reduces calibration time before motor startup.

Background

Lab 3b adds function calls for using the current and voltage offsets that are recorded in the file user.h. It is also shown how to bypass Rs re-calibration. Figure 25 shows where the new additions have been added in the code.

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

During the initialization, the offset values for current and voltage from the file user.h are used to initialize the HAL object. Then the controller is configured to bypass offset calibration. The bias values are stored into the HAL object in the location hal.adcBias.I.value[0 – 2] and hal.adcBias.V.value[0 – 2]. Note that the bias values are not part of ROM and are therefore inside of user memory.

Table 18: New API function calls during initialization.

Initialization		
	CTRL	
	CTRL_setFlag_enableOffset	Sending a false in this function's parameters will cause the controller to bypass the offset calibration.
	HAL	
	HAL_setBias	Manually set the current and voltage offset values that are inherent in the current and voltage feedback circuits of the inverter.

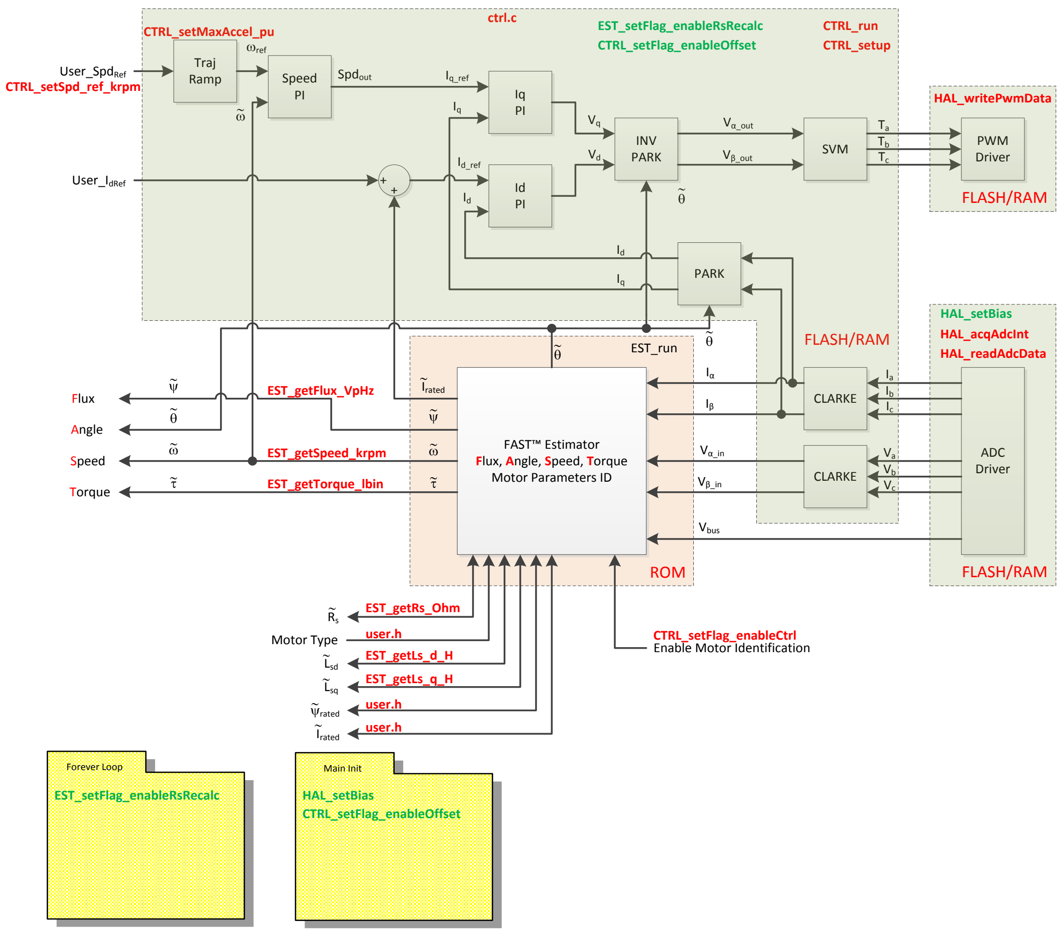


Figure 25: Block diagram of only FAST in ROM with the rest of InstaSPIN-FOC in user memory

Main Run-Time loop (forever loop)

Inside of the forever loop, the Rs re-calibration is disabled. The global variable gMotorVars.Flag_enableRsRecalc is used as a Boolean parameter into the function EST_setFlag_enableRsRecalc(), as see in the code below. This allows the user in real time, to turn Rs Recalc back on from the Watch Window.

```
      EST_setFlag_enableRsRecalc(obj-
>estHandle,gMotorVars.Flag_enableRsRecalc);
```

During initialization, this flag is set to FALSE by default to with the following code:

```
      volatile MOTOR_Vars_t gMotorVars = MOTOR_Vars_INIT;
```

Table 15 lists the function needed to turn off the Rs recalibration.

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

Forever Loop		
	CTRL	
	CTRL_setFlag_enableRsRecalc	Sending a false in this function's parameters will cause the controller to bypass the Rs calibration.

Main ISR

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

Lab Procedure

After verifying that user.h has been properly updated with your inverter's voltage and current offsets, build lab3b, connect to the target and load the .out file.

- Add the appropriate watch window variables by calling the script "proj_lab03b.js".
- Enable the real-time debugger.
- Click the run button.
- Enable continuous refresh on the watch window.

The bias values in the watch window should be zero before the motor is run. To see where the actual bias values are contained in the working code, add the "hal" object to the watch window and expand as shown in the figure below.

hal	{...}	struct _HAL_Obj_
adcHandle	0x00000B00	struct _ADC_Handle_ *
clkHandle	0x00007010	struct _CLK_Handle_ *
cpuHandle	0x0000A8BE	struct _CPU_Handle_ *
flashHandle	0x00000A80	struct _FLASH_Handle_ *
gpioHandle	0x00006F80	struct _GPIO_Handle_ *
offsetHandle_I	0x00AA8A@Program	struct _OFFSET_Handle_ *[3]
offset_I	0x00AA90@Program	struct _OFFSET_[3]
offsetHandle_V	0x00AABA@Program	struct _OFFSET_Handle_ *[3]
offset_V	0x00AAC0@Program	struct _OFFSET_[3]
oscHandle	0x00007014	struct _OSC_Handle_ *
pieHandle	0x00000CE0	struct _PIE_Handle_ *
pllHandle	0x00007011	struct _PLL_Handle_ *
pwmHandle	0x00AAF0@Program	struct _PWM_Handle_ *[3]
pwmDacHandle	0x00AAF6@Program	struct _PWM_Handle_ *[3]
pwrHandle	0x00000985	struct _PWR_Handle_ *
timerHandle	0x00AAFE@Program	struct _TIMER_Handle_ *[3]
wdogHandle	0x00007022	struct _WDOG_Handle_ *
adcBias	{...}	struct _HAL_AdcData_t_
I	{...}	struct _MATH_vec3_
value	0x00AB06@Program	long[3]
(x)= [0]	0.8649999499 (Q-Value(24))	long
(x)= [1]	0.8649999499 (Q-Value(24))	long
(x)= [2]	0.8649999499 (Q-Value(24))	long
V	{...}	struct _MATH_vec3_
value	0x00AB0C@Program	long[3]
(x)= [0]	0.0 (Q-Value(24))	long
(x)= [1]	0.0 (Q-Value(24))	long
(x)= [2]	0.0 (Q-Value(24))	long
(x)= dcBus	709235559	long
(x)= current_sf	29024582	long
(x)= voltage_sf	46361040	long
(x)= numCurrentSensors	3	unsigned short
(x)= numVoltageSensors	3	unsigned short

Figure 26: Watch window before startup.

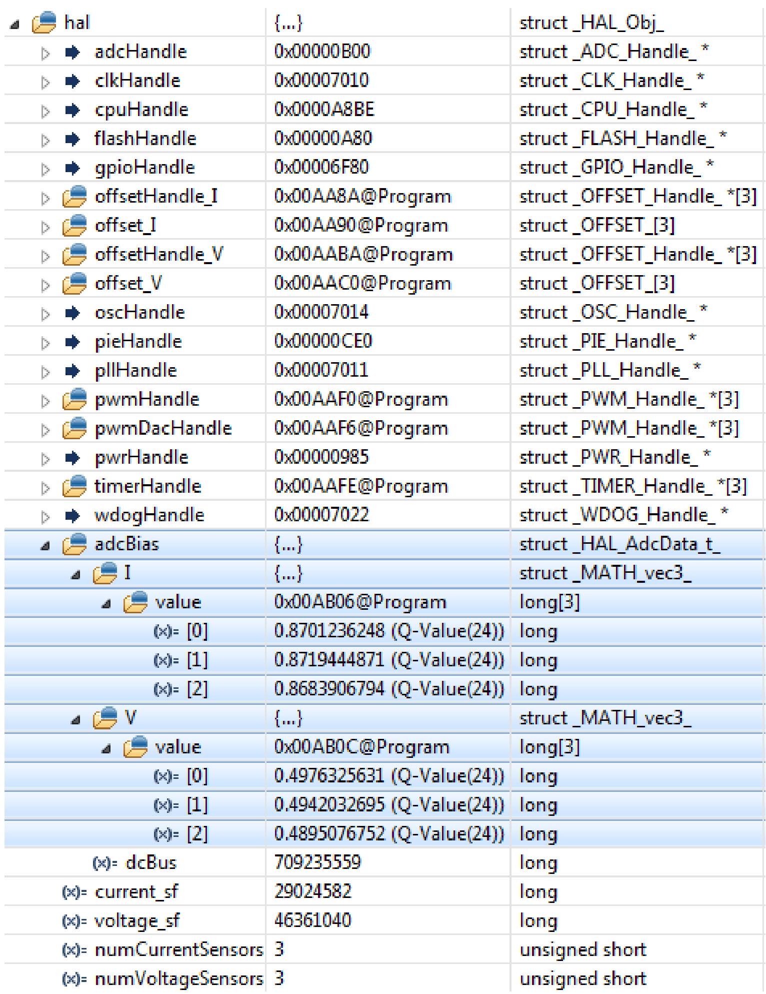
Note how the adcBias.I.value[0-2] for the HAL object are not 0.0 in the beginning. The current values are bi-polar, so after the current signal is converted to a voltage, it is biased up to ½ of the ADC range. That is what is shown in the adc bias values of the Watch Window in the , the bias is set such that the bi-polar current can be measured by the uni-polar ADC.

- Set “Flag_enableSys” to true and then set “Flag_Run_Identify” to true. Watch how the bias values for the HAL object are updated and should be the same as the offset values that are in user.h, as shown below. The watch window will look something like the figure below.
- The motor responds to speed reference changes the same as in previous labs

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

```
//! \brief ADC current offsets for A, B, and C phases
#define    I_A_offset    (0.8577747345)     // 0.7386139631 example value
#define    I_B_offset    (0.8596333265)     // 0.7363774180 example value
#define    I_C_offset    (0.8554588556)     // 0.7402219176 example value

//! \brief ADC voltage offsets for A, B, and C phases
#define    V_A_offset    (0.1735522151)     // 0.1301432252 example value
#define    V_B_offset    (0.1735241413)     // 0.1301434636 example value
#define    V_C_offset    (0.1729278564)     // 0.1303436756 example value
```



hal	{...}	struct _HAL_Obj_
adcHandle	0x00000B00	struct _ADC_Handle_ *
clkHandle	0x00007010	struct _CLK_Handle_ *
cpuHandle	0x0000A8BE	struct _CPU_Handle_ *
flashHandle	0x00000A80	struct _FLASH_Handle_ *
gpioHandle	0x00006F80	struct _GPIO_Handle_ *
offsetHandle_I	0x00AA8A@Program	struct _OFFSET_Handle_ *[3]
offset_I	0x00AA90@Program	struct _OFFSET_[3]
offsetHandle_V	0x00AABA@Program	struct _OFFSET_Handle_ *[3]
offset_V	0x00AAC0@Program	struct _OFFSET_[3]
oscHandle	0x00007014	struct _OSC_Handle_ *
pieHandle	0x00000CE0	struct _PIE_Handle_ *
pllHandle	0x00007011	struct _PLL_Handle_ *
pwmHandle	0x00AAF0@Program	struct _PWM_Handle_ *[3]
pwmDacHandle	0x00AAF6@Program	struct _PWM_Handle_ *[3]
pwrHandle	0x00000985	struct _PWR_Handle_ *
timerHandle	0x00AAFE@Program	struct _TIMER_Handle_ *[3]
wdogHandle	0x00007022	struct _WDOG_Handle_ *
adcBias	{...}	struct _HAL_AdcData_t_
I	{...}	struct _MATH_vec3_
value	0x00AB06@Program	long[3]
(x)= [0]	0.8701236248 (Q-Value(24))	long
(x)= [1]	0.8719444871 (Q-Value(24))	long
(x)= [2]	0.8683906794 (Q-Value(24))	long
V	{...}	struct _MATH_vec3_
value	0x00AB0C@Program	long[3]
(x)= [0]	0.4976325631 (Q-Value(24))	long
(x)= [1]	0.4942032695 (Q-Value(24))	long
(x)= [2]	0.4895076752 (Q-Value(24))	long
(x)= dcBus	709235559	long
(x)= current_sf	29024582	long
(x)= voltage_sf	46361040	long
(x)= numCurrentSensors	3	unsigned short
(x)= numVoltageSensors	3	unsigned short

Figure 27: The watch window after bias calculation and Rs re-calibration.

Conclusion

This lab showed how to read in your own inverter's offset values for voltage and current. Before the motor starts spinning, offset voltage calibration and Rs recalibration take time. If these values are previously measured, they can be stored and read into the control allowing us to bypass the time it takes to measure these values.

Lab 3c – Using your own Board Parameters: Current and Voltage Offsets, with fpu32

Abstract

This lab runs Lab 3b with floating point unit enabled. This lab only applies to 6x devices, as it has a floating point unit.

Objectives Learned

Bypassing motor identification and using motor parameters from user.h, with fpu32 enabled.

Lab Procedure

Follow the exact same procedure as in Lab 3b.

Conclusion

We conclude that the libraries in ROM also work when fpu32 is enabled in 6x devices.

Lab 4 – Current Loop Control (Create a Torque Controller)

Abstract

The speed loop is disabled and a reference is sent directly to the Iq current controller.

Introduction

This lab will explore using the InstaSPIN-FOC and InstaSPIN-MOTION framework purely as a torque controller by removing the speed loop.

Objectives Learned

- How to enable direct torque commands by putting InstaSPIN-FOC or InstaSPIN-MOTION into user external speed control mode and directly providing an Iq reference
- How to get torque information from FAST

"User external speed control mode" disables the provided Speed controller and sets the FOC control system to expect an Iq torque command as an input. This torque command input can be provided by the user or can be the output from a user provided speed controller (which will be demonstrated in future labs).

Background

FOC at its core is a torque controller. We just often see it wrapped inside of a speed or position loop. By adding extra control loops there is more of a chance to generate instability. So it is in Torque mode where the FAST algorithm and its estimation and angle tracking capabilities should truly be judged. You can have a perfect torque controller, but if your speed loop is commanding an under-damped, oscillating torque reference signal the overall performance of the system will still be quite poor. The speed controlled system also masks the true performance of the FAST FOC system. By adding a speed controller, the total response of the FAST FOC system is reduced.

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

During the initialization, a function is called to disable the speed PI and the SpinTAC™ speed controller. The figure below lists the function and its description used to disable the speed control and then allow the Iq reference to be sent directly to the Iq PI controller.

Table 20: New API function calls during initialization.

Initialization		
	CTRL	
	CTRL_setFlag_enableSpeedCtrl	Sending a false in this function's parameters will cause the controller to bypass the speed PI controller and allow direct access to the Iq current PI controller.

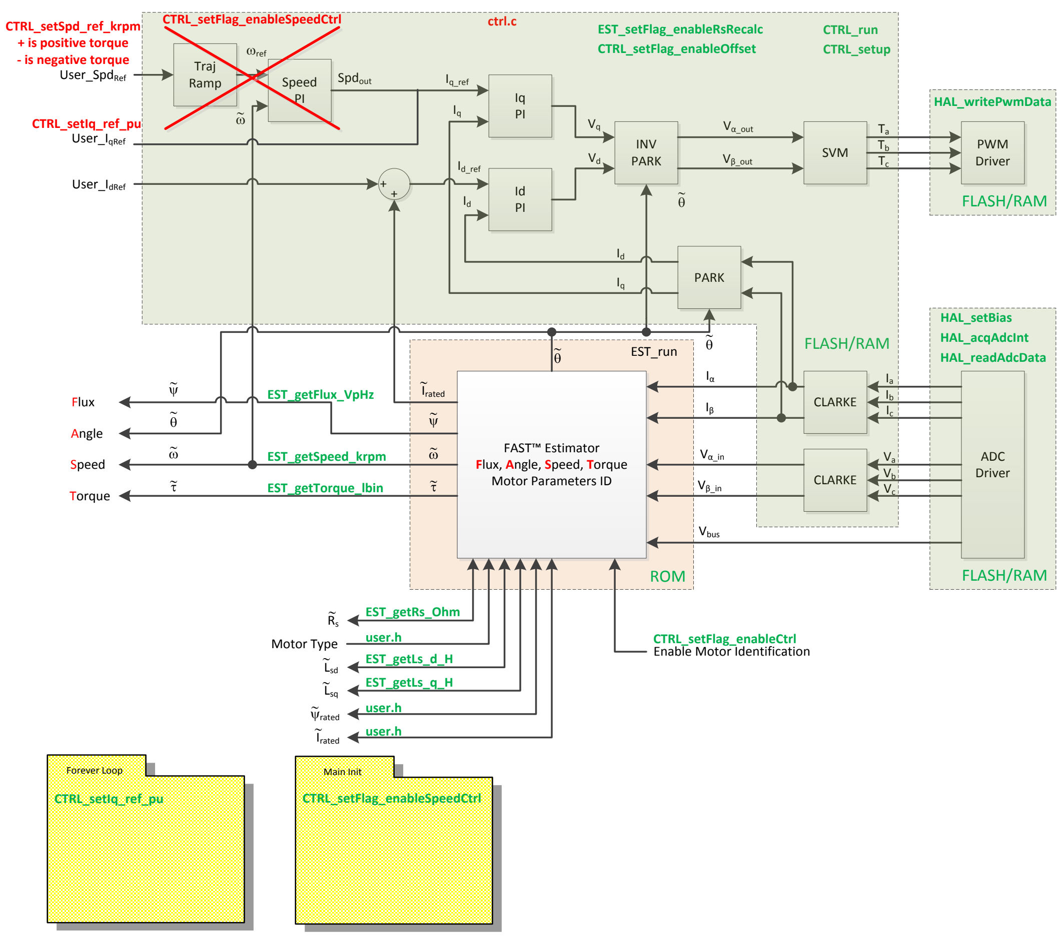


Figure 28: Block diagram of only FAST in ROM with the rest of InstaSPIN-FOC in user memory

Main Run-Time loop (forever loop)

Inside of the forever loop, the Iq reference is converted from a user input of amps (gMotorVars.IqRef_A) to per unit with the following code.

```
  _iq iq_ref =
_IQmpy(gMotorVars.IqRef_A,_IQ(1.0/USER_IQ_FULL_SCALE_CURRENT_A));
```

To set the Iq reference in the FOC control, the function CTRL_setIq_ref_pu() passes the Iq reference to the non-inverting input of the Iq current PI controller with the following code.

```
  // Set the Iq reference that use to come out of the PI speed control

  CTRL_setIq_ref_pu(handle, iq_ref);
```

One other task needs to be completed while doing torque control. When motoring, the controller needs to know the direction that the motor will be applying torque. The function CTRL_setSpd_ref_krpm() function is used to set whether the motor starts in the clock-wise or counter-clockwise direction. When using the CTRL_setSpd_ref_krpm() function in torque control mode, the magnitude of the speed parameter does not matter only the sign.

```
 // set the speed reference so that the forced angle rotates in the correct
direction

 // for startup.

  if(_IQabs(gMotorVars.Speed_krpm) < _IQ(0.01))

  {

      if(iq_ref < 0)

          CTRL_setSpd_ref_krpm(handle,_IQ(-0.01));

      else if(iq_ref > 0)

          CTRL_setSpd_ref_krpm(handle,_IQ(0.01));

  }
```

Another point to look out for is during the zero-speed crossing. The sign for the speed reference must not be in the incorrect direction until the actual speed is zero. To emphasize this, the flowchart in Figure 16 shows what is going on in the software. As you can see, the software sets the direction for the speed reference when speed is reduced to under 10 RPM, using the Iqref sign to set direction.

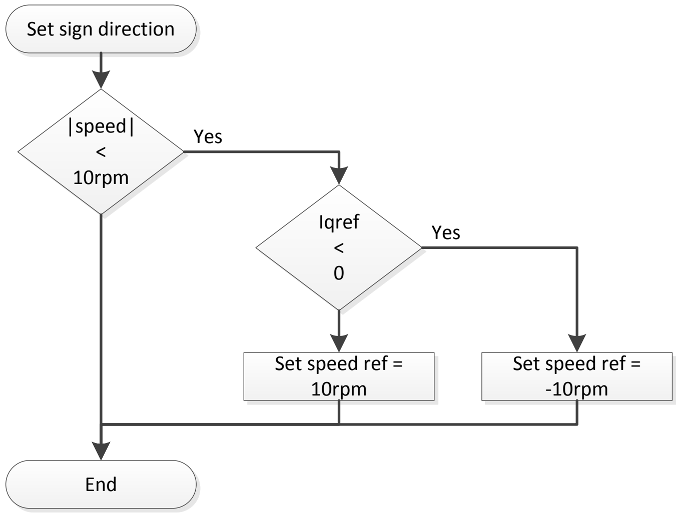


Figure 29: Flowchart describing how to test for the speed direction before setting the speed sign.

The following table lists the new functions in the main loop that are needed to implement the torque controller.

Table 21: New API function calls during the main run-time loop.

Forever Loop		
	CTRL	
	CTRL_setIq_ref_pu	Take the Iq current reference as a parameter and set the input of the Iq PI current controller.

Main ISR

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

Lab Procedure

Build lab4, connect to the target and load the .out file.

- Add the appropriate watch window variables by calling the script “proj_lab04.js”.
- Enable the real-time debugger.
- Click the run button.
- Enable continuous refresh on the watch window.

Start spinning the motor.

- Set “Flag_enableSys” to true and then set “Flag_Run_Identify” to true.
- Increase the variable “gMotorVars.IqRef_A” until the shaft starts spinning, try increments of 0.1 (Q-Value(24)).
    - Notice that if the IqRef value is not set high enough, the shaft will slowly oscillate. The slow oscillation (1Hz) is due to the forced startup routine which is called “forced angle”. The forced angle allows InstaSPIN to start from zero speed at full motor torque. If the motor shaft is spinning above 1Hz then the forced angle is off and true closed loop control is performed. The forced angle can be manually turned off. One reason to turn off the forced angle is to have smooth transitions through zero speed.
    - Notice that because the control is a torque controller and the motor is unloaded, the shaft will spin to full (voltage limited) speed.

Change to different torque references

- With the motor spinning, try stopping the shaft. If the Iqref value was set to only overcome startup static friction and cogging torques, it will feel like the motor is providing a very small torque.
- Continue to increase the Iq reference and note how the torque provided by the motor changes.
    - Note that the “gMotorVars.IqRef_A” sets the Iq reference in amps. The Iq reference setting is also the peak current of the motor line currents. In “user.h” under the motor settings, the maximum motor current is listed as “USER_MOTOR_MAX_CURRENT”. The Iq PI controller will control motor peak current up to the “user.h” maximum motor current. The Iq reference can be set higher than the maximum motor current, but the controller will never allow more current to flow than what is listed in the “user.h” setting.

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

Many applications will require only a torque controller. InstaSPIN can easily covert a motor controller to either control speed or torque. In this lab the motor controller was converted into a torque controller. With the FOC system, the Iq reference directly relates to torque.

Lab 4a – Current Loop Control (Create a Torque Controller), with fpu32

Abstract

This lab runs Lab 4 with floating point unit enabled. This lab only applies to 6x devices, as it has a floating point unit.

Objectives Learned

Running a torque controller with fpu32 enabled.

Lab Procedure

Follow the exact same procedure as in Lab 4.

Conclusion

We conclude that the libraries in ROM also work when fpu32 is enabled in 6x devices.

Lab 5a – Tuning the Current Loops

Abstract

A technique of setting the proportional gain (Kp) and integral gain (Ki) for the current controllers of the 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.

Lab5h will show how to use the graphing features of CCS to do step response testing to check the tuning of your PI controllers.

Introduction

This lab explains how to tune PI current gains 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 17.

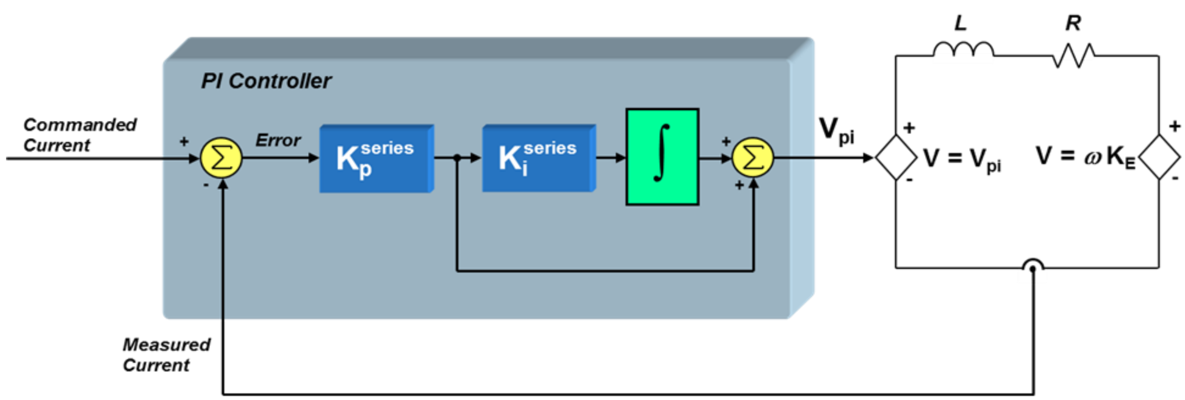


Figure 30: Series PI current controlled motor system including the stator.

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 5b. 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 below:

$$K_i^{series} = \frac{R}{L}$$

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

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.

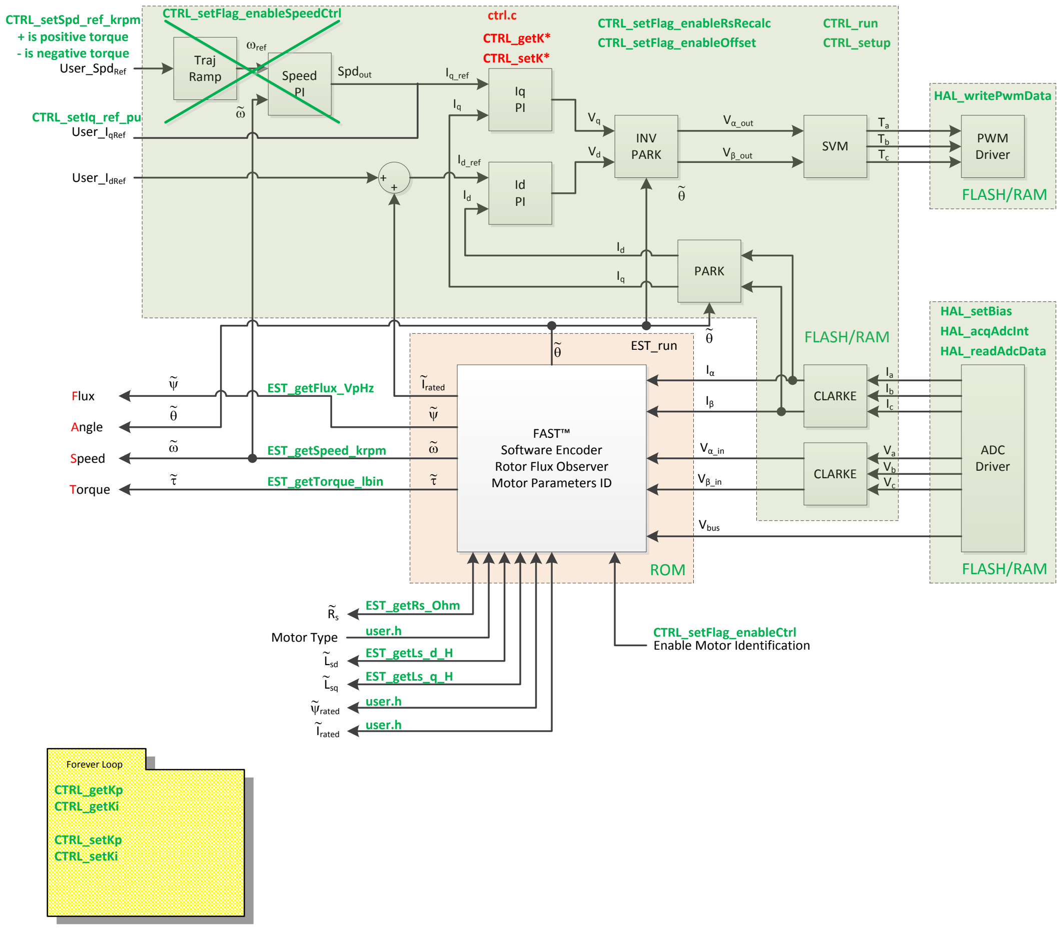


Figure 31: Block diagram of an opened source InstaSPIN implementation.

Main Run-Time loop (forever loop)

During motor identification or even 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 calcPIgains(). After calcPIgains() is called, the global variables gMotorVars.Kp_Idq and gMotorVars.Ki_Idq are initialized with the newly calculated K_p^{series} and K_i^{series} gains. The following figure shows the logic flowchart needed to implement the current controller gain initialization.

Table 22: New API function calls during the main run-time loop.

Forever Loop		
	CTRL	
	CTRL_getKp	Get the Kp gain from the CTRL object.
	CTRL_getKi	Get the Ki gain from the CTRL object.
	CTRL_setKp	Set the Kp gain in the CTRL object.
	CTRL_setKi	Set the Ki gain in the CTRL object.

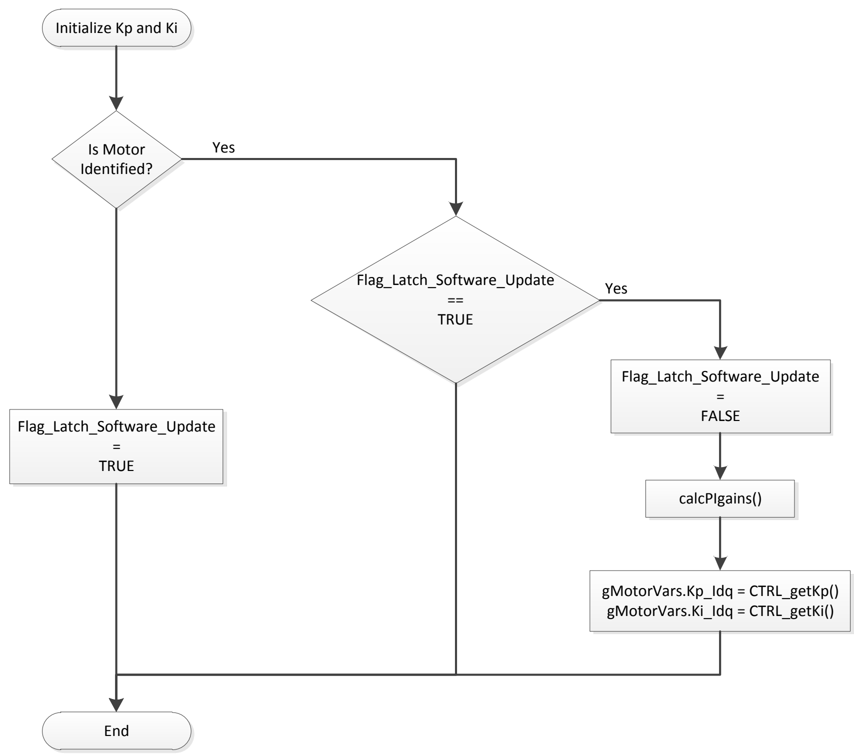


Figure 32: Flowchart showing how the watch window K_p^{series} and K_i^{series} variables are initialized.

Main ISR

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

Lab Procedure

Build lab5a, connect to the target and load the .out file.

- Add the appropriate watch window variables by calling the script “proj_lab05a.js”.
- Enable the 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 (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
- $PWMvsISRtick$ is the tick rate between PWM and interrupts, USER_NUM_PWM_TICKS_PER_ISR_TICK
- $ISRvsCTRLtick$ is the tick rate between interrupts and controller state machine, USER_NUM_ISR_TICKS_PER_CTRL_TICK
- $CTRLvsCURRENTtick$ is the tick rate between controller state machine and current controllers, USER_NUM_CTRL_TICKS_PER_CURRENT_TICK

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

Start the control
- Set Flag_enableSys = TRUE
- Set Flag_Run_Identify = TRUE

Compare K_i^{series} values
- Compare your calculated K_i^{series} from the K_i^{series} that is initially stored in “gMotorVars.Ki_Idq”. The two values should be the same.

Compare and adjust K_p^{series} values
- Since the K_p^{series} gain controls bandwidth of the control, its adjustment is better 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"
  - $V_{fs} = \text{USER\_IQ\_FULL\_SCALE\_VOLTAGE\_V}$ → found in "user.h"

- Put your new calculated K_p^{series} into gMotorVars.Kp_Idq.

As the bandwidth gets wider, the sampling delay has more of a negative effect on the phase margin of the controller and causes the control loop to become unstable. Figure 33 is a plot of the motor current waveform with a stable K_p^{series} setting. As the K_p^{series} is increased, the phase margin of the control loop becomes smaller. After a while the control loop is unstable and starts to oscillate as shown in Figure 34. The current controller gain should not be set to this high of a value. When a current loop instability occurs, lower the K_p^{series} gain until the current waveform is like the one in the following figure.

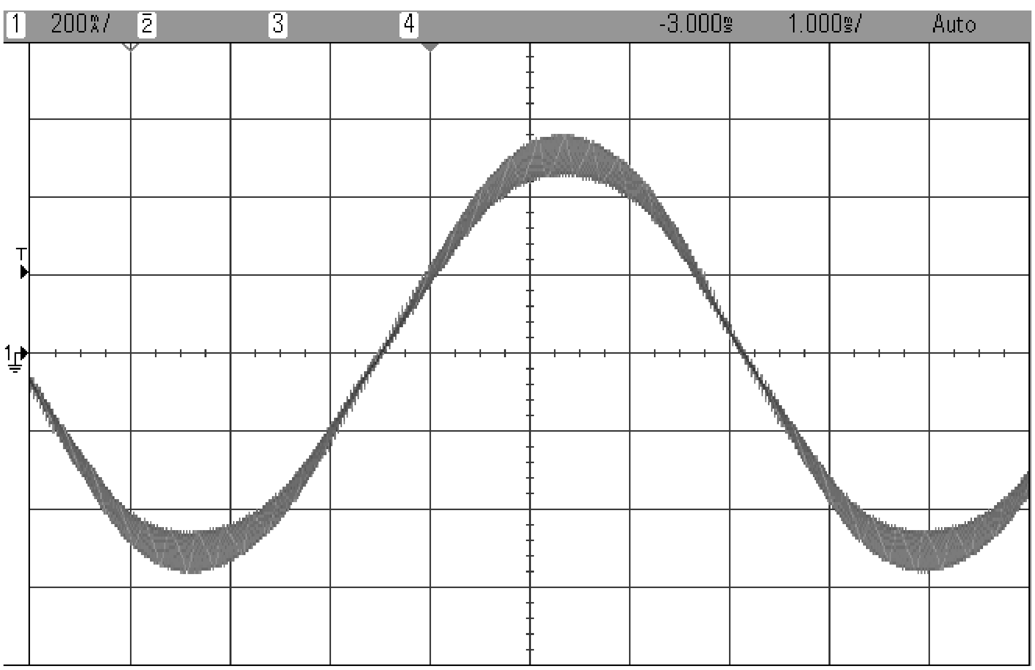


Figure 33: K_p^{series} setting that has been calculated at 1/20 of the bandwidth.

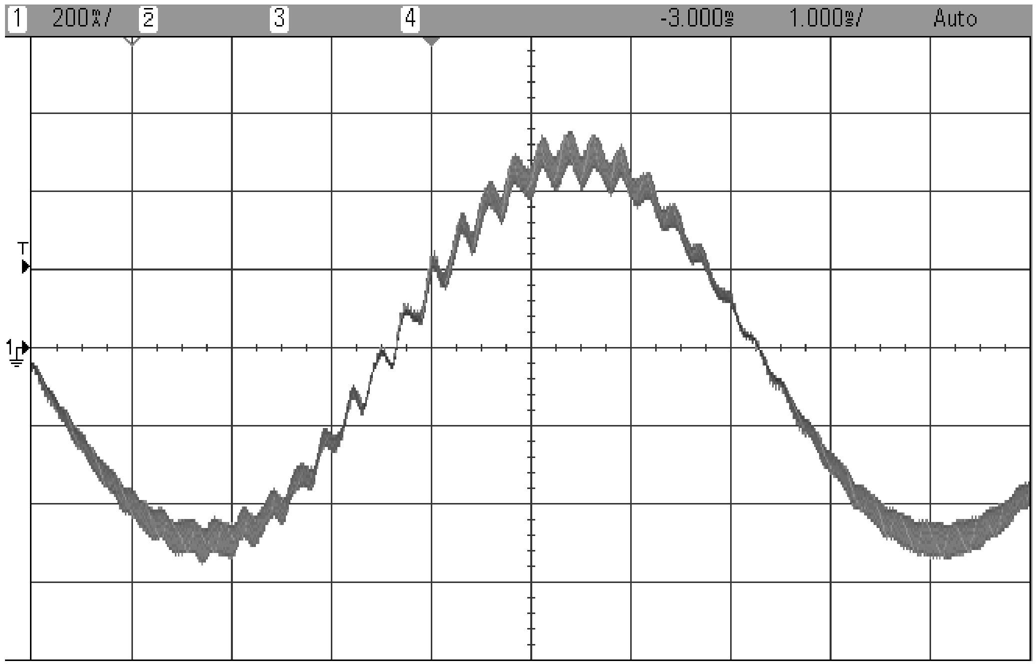


Figure 34: K_p^{series} setting that is too high resulting in the controller becoming unstable.

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, its current waveform looks like that in Figure 34.
- Reset K_p^{series} back to the value that was calculated before.
    - Actually this is a pretty good way tuning the current control bandwidth when no current measurement is available.

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms 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 consider a 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

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 relate to the time constant of the speed controlled system and it is best to wait until more knowledge of the mechanical system is attained before calculating the current controller's K_p^{series}.

Lab 5b – Tuning the Speed Loop

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 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 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.

Lab5h will show how to use the graphing features of CCS to do step response testing to check the tuning of your PI controllers.

The InstaSPIN-MOTION disturbance-rejecting speed controller replaces the standard PI controller. The InstaSPIN-MOTION controller offers several ease of use and performance advantages: 1) it proactively estimates and compensates for system errors; 2) the controller offers single-parameter tuning that typically works over the entire operating range. If you would like to use the InstaSPIN-MOTION controller, you may skip Lab 5b and proceed to Lab 5c.

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, playing more of an effect on 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, then it 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 the next figure.

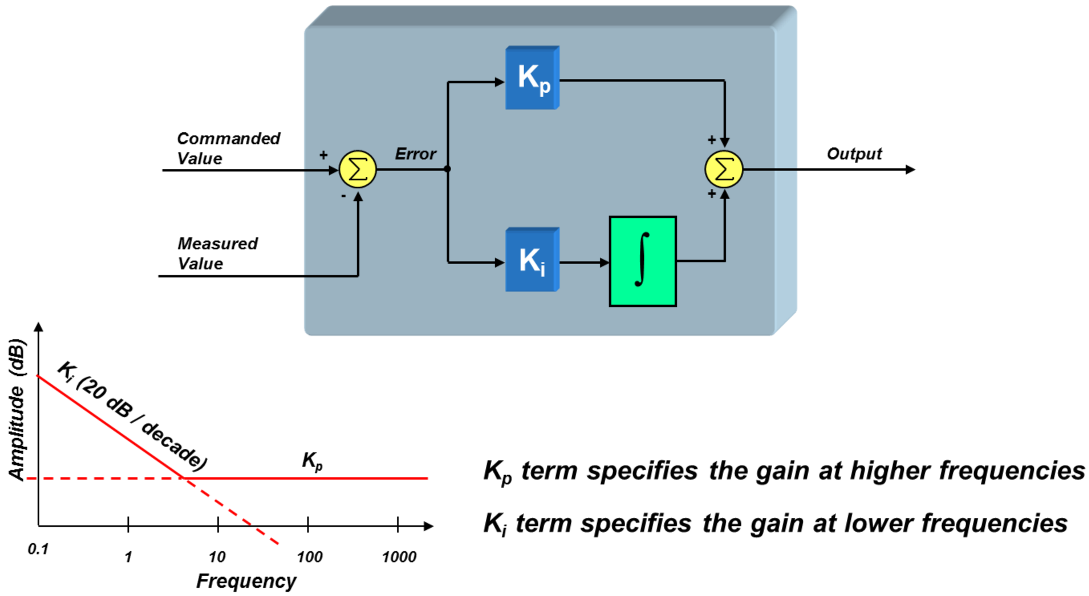


Figure 22: Parallel PI control.

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 real 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

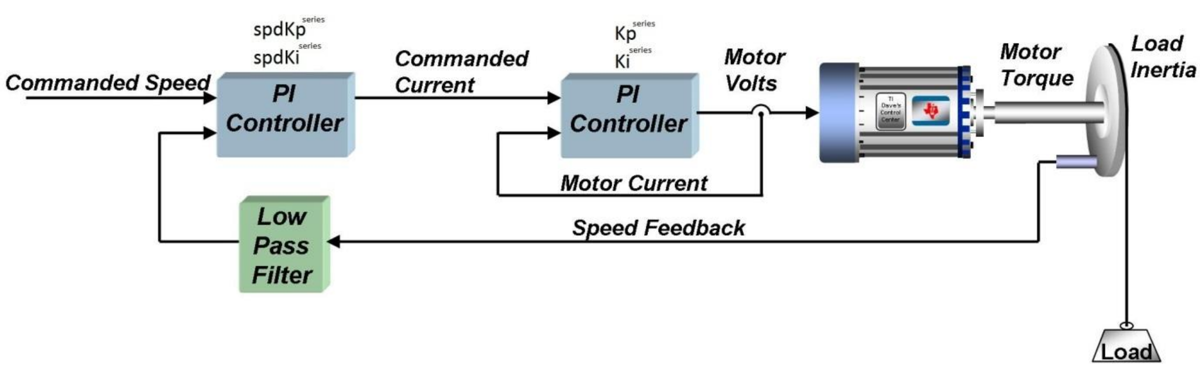


Figure 23: Speed controller cascaded with a current controller and speed filter.

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

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

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 1

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 following Equation 2 and $G_{current}$ can be approximately 1 if the current control bandwidth is enough high than the speed control bandwidth like Equation 1. So if we eliminate the effect of the current controller pole, the openloop transfer function becomes like Equation 3.

$$K = \frac{K_T}{J}$$ Equation 2

$$GH(s) = \frac{K \cdot spdK_p^{series} \cdot spdK_i^{series}\left(1 + \frac{s}{spdK_i^{series}}\right)}{s^2(1 + \tau s)}$$ Equation 3

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 this:

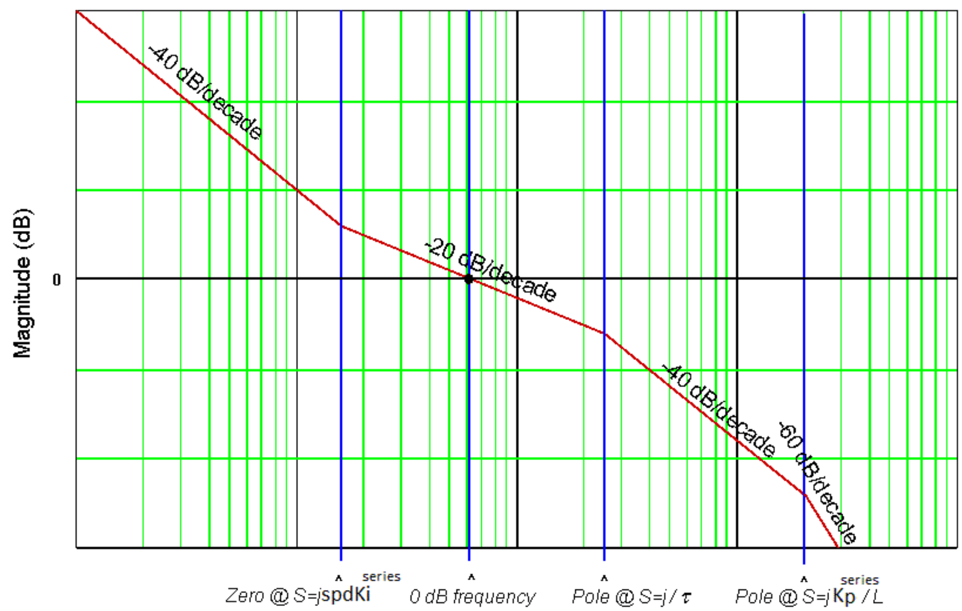


Figure 24: Bode plot of how the system should be tuned.

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 δ we will define 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 25, 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.

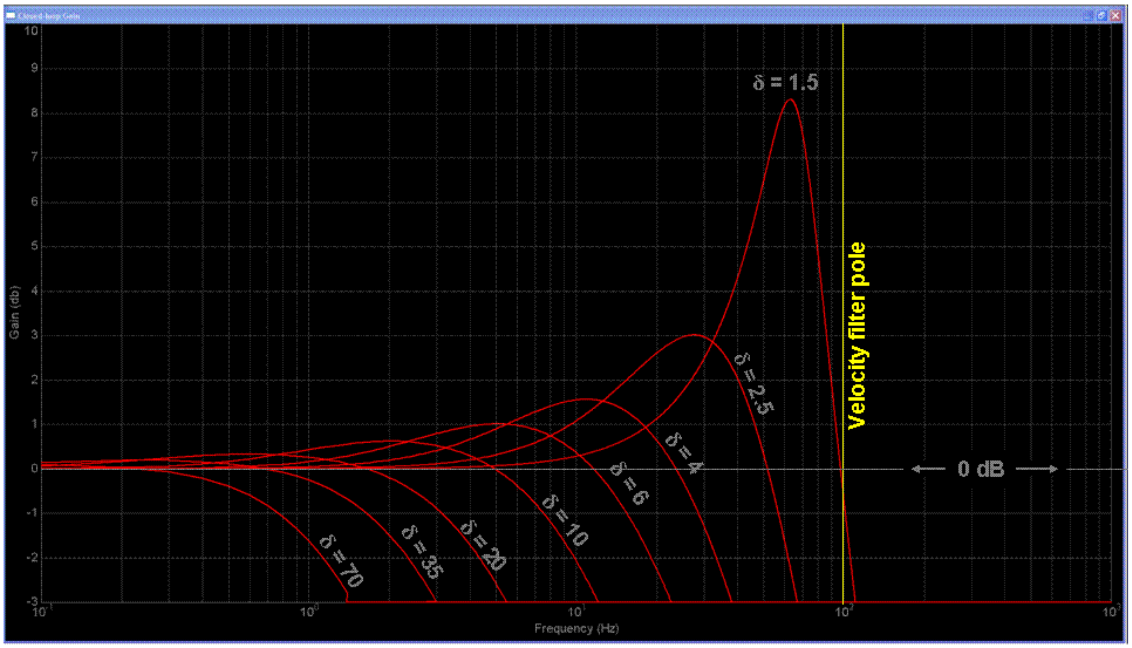


Figure 25: Closed loop magnitude response of the speed loop for various δ.

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 following.

$$spdK_i^{series} = \frac{1}{\delta^2 \cdot \tau} \qquad \text{Equation 4}$$

$$spdK_p^{series} = \frac{\delta \cdot spdK_i^{series}}{K} = \frac{1}{\delta \cdot K \cdot \tau} \qquad \text{Equation 5}$$

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 certainly 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:

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

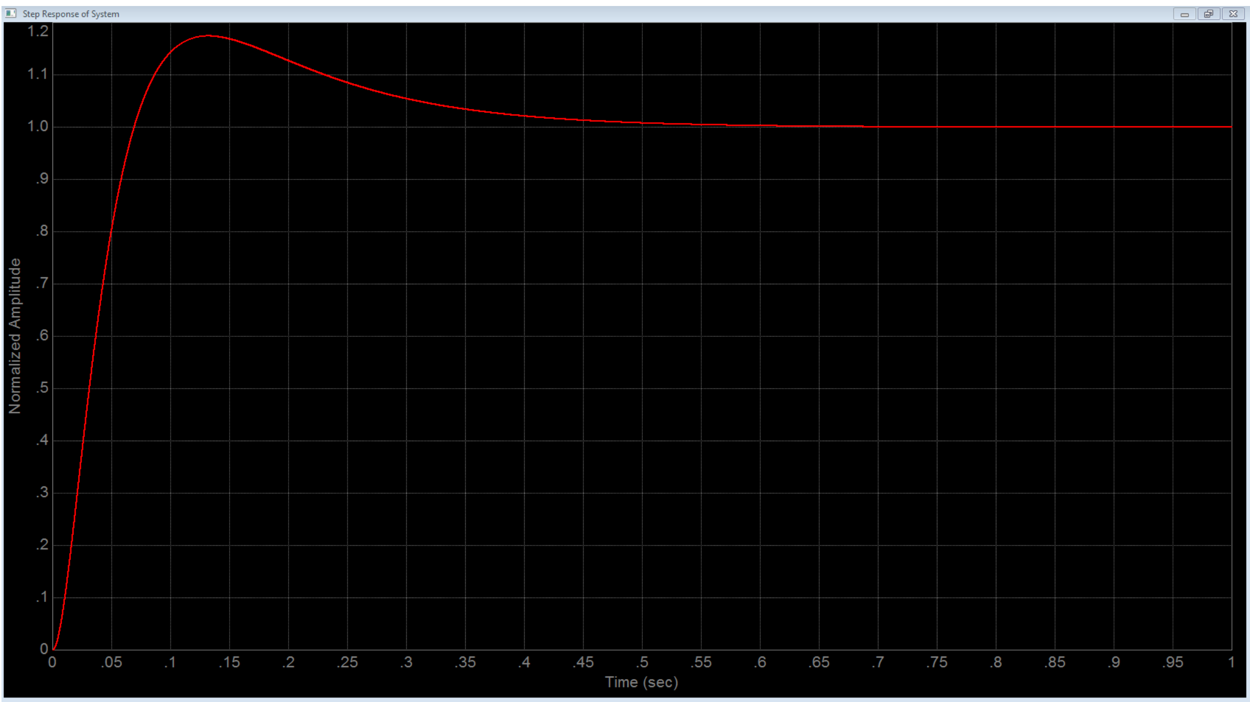
Finally, recall that

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

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 the following figure where the time axis is now scaled appropriately for this design example.



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.

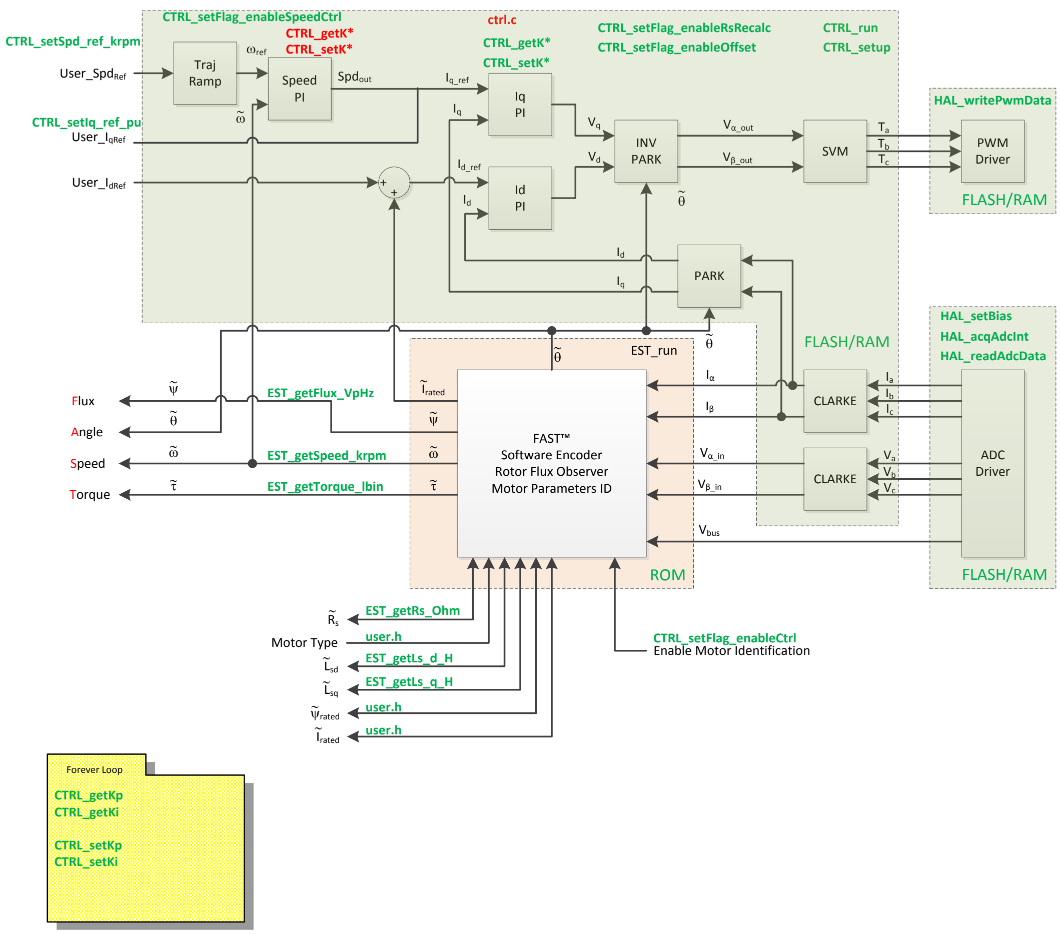


Figure 26: Block diagram of an opened source InstaSPIN implementation.

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 “gMotorVars” structure.

Table 23: New API function calls during the main run-time loop.

Forever Loop		
	CTRL	
	CTRL_getKp	Get the Kp gain from the CTRL object.
	CTRL_getKi	Get the Ki gain from the CTRL object.
	CTRL_setKp	Set the Kp gain in the CTRL object.
	CTRL_setKi	Set the Ki gain in the CTRL object.

Main ISR

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

Lab Procedure

Build lab5b, connect to the target and load the .out file.

- Add the appropriate watch window variables by calling the script “proj_lab05b.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 “Flag_enableOffsetcalc” = 1
- Set “Flag_enableSys” = 1
- Set “Flag_Run_Identify” = 1
- Set “Flag_enableForceAngle” = 0

Turn the motor into a spring

- Set “SpeedRef_krpm” = 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 8312 kit and 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 8312 kit 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 8312 kit 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:

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

Where:

- T_i is the current controller period
- PWM_Freq_kHz can be taken from USER_PWM_FREQ_kHz parameter in user.h
- $PWMvsISRtick$ is the tick rate between PWM and interrupts, USER_NUM_PWM_TICKS_PER_ISR_TICK
- $ISRvsCTRLtick$ is the tick rate between interrupts and controller state machine, USER_NUM_ISR_TICKS_PER_CTRL_TICK
- $CTRLvsCURRENTtick$ is the tick rate between controller state machine and current controllers, USER_NUM_CTRL_TICKS_PER_CURRENT_TICK

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

Where:

- T_v is the speed controller period
- $CTRLvsSPEEDtick$ is the tick rate between controller state machine and speed controllers, USER_NUM_CTRL_TICKS_PER_SPEED_TICK

$Vel_{fs} = USER\_IQ\_FULL\_SCALE\_FREQ\_Hz$ – Full scale frequency in Hz

$I_{fs} = USER\_IQ\_FULL\_SCALE\_CURRENT\_A$ – Full scale current in A

$V_{fs} = USER\_IQ\_FULL\_SCALE\_VOLTAGE\_V$ – 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 constan 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 shows the conversion.

$$spdK_p^{parallel} = spdK_p^{series}$$
$$spdK_i^{parallel} = spdK_i^{series} \cdot spdK_p^{parallel}$$

Equation 6

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

- $spdK_i^{PU} = spdK_i^{parallel} \cdot T_v \cdot \frac{4\pi \cdot Vel_{fs}}{I_{fs} \cdot P}$

- $curK_i^{PU} = K_i^{series} \cdot T_i$

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

- $spdK_p^{PU} = spdK_p^{parallel} \cdot \frac{4\pi \cdot Vel_{fs}}{I_{fs} \cdot P}$

- $curK_p^{PU} = K_p^{series} \cdot \frac{I_{fs}}{V_{fs}}$

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

- gMotorVars.Kp_spd =$spdK_p^{PU}$
- gMotorVars.Ki_spd =$spdK_i^{PU}$
- gMotorVars.Kp_Idq = $curK_p^{PU}$
- gMotorVars.Ki_Idq = $curK_i^{PU}$

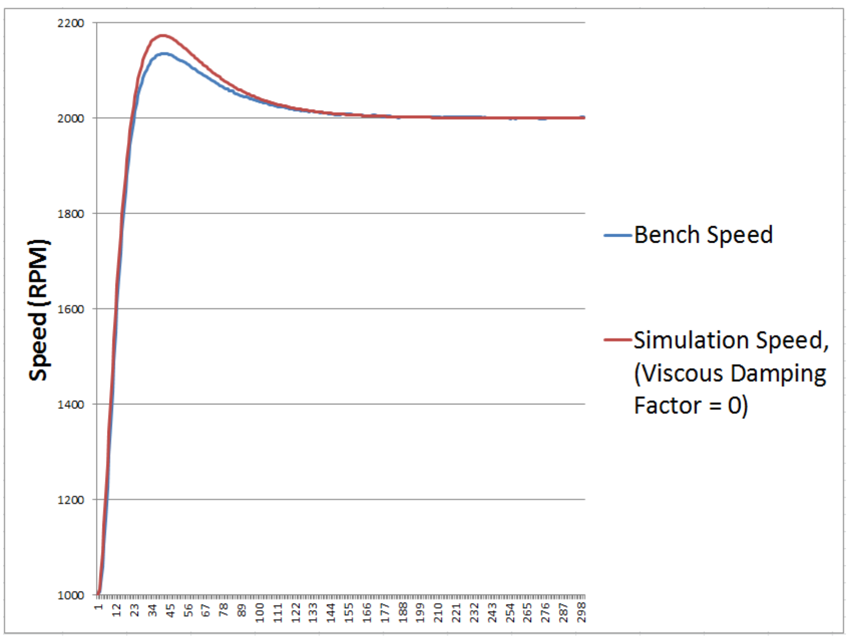
Run the motor and load the shaft to see the performance

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

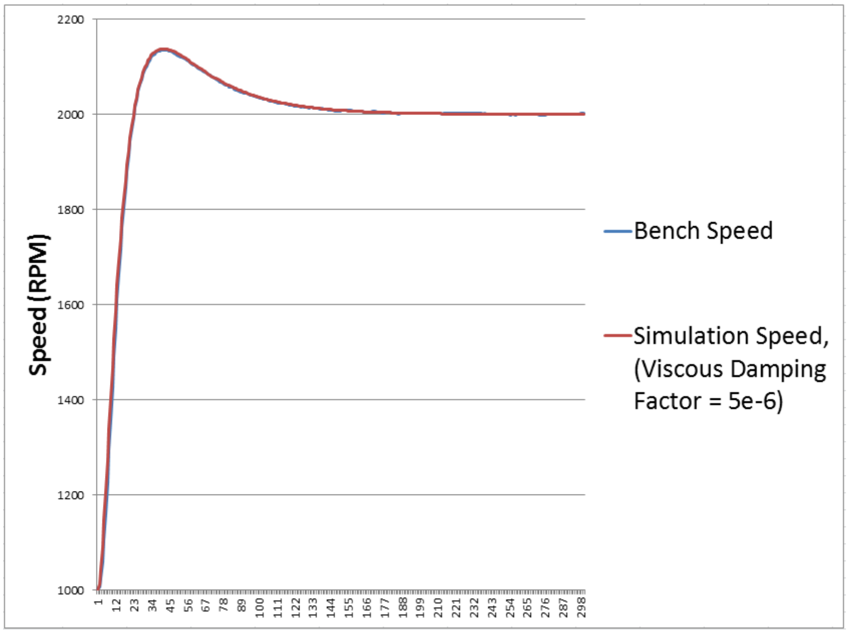
When done experimenting with the motor:

- Set the variable "Flag_Run_Identify" to false to turn off the pwms 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 this:



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



Conclusion

Tuning the speed controller has more unknowns than when 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.

Lab 5c - InstaSPIN-MOTION Inertia Identification

Abstract

Both InstaSPIN-FOC and InstaSPIN-MOTION are sensorless FOC solutions that identify, tune and control your motor in minutes. Both solutions feature:

- 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 option for user adjustments
- Automatic or manual field weakening and field boosting
- Bus voltage compensation

InstaSPIN-MOTION combines this functionality with SpinTAC™ components from LineStream Technologies. SpinTAC features:

- Speed controller: A disturbance-rejecting speed controller proactively estimates and compensates for system errors. SpinTAC automatically estimates system inertia (bypassing the need for lengthy calculations). The controller offers single-parameter tuning that typically works over the entire operating range.
- Motion profile planning: Trajectory planning for easy design and execution of complex motion sequences,
- Motion profile generation: A motion engine that ensures that your motor transitions from one speed to another as smoothly as possible.

Inertia identification is the first step in enabling the SpinTAC™ speed controller. The inertia value is used by the SpinTAC Velocity Control to determine how strongly to respond to the disturbances in the system.

InstaSPIN-MOTION provides a mechanism to identify the system inertia. If the inertia of your system is known you can populate the inertia value in your project, and bypass the SpinTAC™ Inertia Identification process (see the Bypassing Inertia Identification section of the InstaSPIN-FOC and InstaSPIN-MOTION User Guide).

Once the inertia is identified, it can be set as the default value and does not need to be estimated again unless there is a change in your system.

In this lab, you will learn how to run the inertia identification process from within your MotorWare project. Additional information about Inertia Identification can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section Inertia Identification).

Introduction

Inertia is a measure of the motor's resistance to change in velocity. The greater the motor inertia, the greater the torque needed to accelerate or decelerate the motor. The SpinTAC Velocity Control uses the system's inertia value to provide the most accurate control. SpinTAC™ Velocity Identify automatically measures the system inertia by spinning the motor and measuring feedback.

In this lab, you will learn how to run SpinTAC™ Velocity Identify's inertia identification process from within your MotorWare project

Prerequisites

The user motor settings from the user.h file need to be copied into the InstaSPIN-MOTION user.h file. If you are working with the DRV8312 Rev D evaluation kit:

1. Open the user.h file that was modified as part of InstaSPIN-FOC lab 2a. It is located in "sw\solutions\instaspin_foc\boards\drv8312_revD\f28x\f2806xF\src"
2. Locate the USER_MOTOR settings that you identified in lab 02a. It should appear similar to the following:
```
#elif (USER_MOTOR == MY_MOTOR)
#define USER_MOTOR_TYPE                  MOTOR_Type_Pm
#define USER_MOTOR_NUM_POLE_PAIRS        (4)
#define USER_MOTOR_Rr                    (NULL)
#define USER_MOTOR_Rs                    (0.4051206)
#define USER_MOTOR_Ls_d                  (0.0006398709)
#define USER_MOTOR_Ls_q                  (0.0006398709)
#define USER_MOTOR_RATED_FLUX            (0.03416464)
#define USER_MOTOR_MAGNETIZING_CURRENT   (NULL)
#define USER_MOTOR_RES_EST_CURRENT       (1.0)
#define USER_MOTOR_IND_EST_CURRENT       (-1.0)
#define USER_MOTOR_MAX_CURRENT           (5.0)
#define USER_MOTOR_FLUX_EST_FREQ_Hz      (20.0)
```
3. Open the user.h file for InstaSPIN-MOTION. It is located in "sw\solutions\instaspin_motion\boards\drv8312_revD\f28x\f2806xM\src"
4. Copy the USER_MOTOR settings from the InstaSPIN-FOC user.h into the InstaSPIN-MOTION user.h, Your new entry should appear similar to the following:
```
#elif (USER_MOTOR == MY_MOTOR)
#define USER_MOTOR_TYPE                  MOTOR_Type_Pm
#define USER_MOTOR_NUM_POLE_PAIRS        (4)
#define USER_MOTOR_Rr                    (NULL)
#define USER_MOTOR_Rs                    (0.4051206)
#define USER_MOTOR_Ls_d                  (0.0006398709)
#define USER_MOTOR_Ls_q                  (0.0006398709)
#define USER_MOTOR_RATED_FLUX            (0.03416464)
#define USER_MOTOR_MAGNETIZING_CURRENT   (NULL)
#define USER_MOTOR_RES_EST_CURRENT       (1.0)
#define USER_MOTOR_IND_EST_CURRENT       (-1.0)
#define USER_MOTOR_MAX_CURRENT           (5.0)
#define USER_MOTOR_FLUX_EST_FREQ_Hz      (20.0)
#define USER_MOTOR_ENCODER_LINES         (2000.0)
#define USER_MOTOR_MAX_SPEED_KRPM        (4.0)
#define USER_SYSTEM_INERTIA              (0.02)
#define USER_SYSTEM_FRICTION             (0.01)
```
5. Notice that there are now four new fields for MY_MOTOR:
    - **USER_MOTOR_ENCODER_LINES** – This should be set to the number of pulses on your motor's encoder. If your motor does not have an encoder, set this to 1.0.

- **USER_MOTOR_MAX_SPEED_KRPM** – This should be set to the maximum speed that your motor can run.
- **USER_SYSTEM_INERTIA** – We will determine this value as part of this lab. Please set it the default value of 0.02.
- **USER_SYSTEM_FRICTION** - We will determine this value as part of this lab. Please set it the default value of 0.01.

6. There is an additional new define for InstaSPIN-MOTION, **USER_SYSTEM_BANDWIDTH_SCALE** (not included in the picture). This definition represents the default bandwidth for the SpinTAC controller. We will determine this value in lab 05e. Please set it to the default value of 1.0

In addition to the USER_MOTOR settings, it is important that you copy ANY field that you modified as part of the previous labs or as part of your hardware design process into this new user.h file.

If you are using a different evaluation kit, you should replace the drv8312_revD directory with your kit's directory.

Objectives Learned

- Call the API functions to set up SpinTAC™ Velocity Identify
- Start the inertia identification process
- Update the inertia value for your motor in user.h

Background

Inertia is the resistance of an object to rotational acceleration around an axis. This value is typically calculated as the ratio between the torque applied to the motor and the acceleration of the mass rigidly coupled with that motor. This test needs to be done under negligible friction and load.

There is a common misunderstanding that inertia is equivalent to load. Load usually consists of two components, load inertia and load torque. Load inertia is the mass that will spin simultaneously with the motor rotor, while the load torque appears as an external torque applied on the motor rotor shaft. An easy way to differentiate the load inertia from load torque is to consider whether the load will spin together with the rotor shaft if the rotor shaft changes spinning direction. Direct couplers and belt pulleys with the mass rigidly mounted to the load shaft are examples of load inertia. Load inertia and motor rotor inertia contribute to the system inertia. Examples of load torque include: gravity of a mass applied to one side of the motor rotor shaft, distributed clothes in a washing machine drum during the spin cycle, and the fluid viscosity of a pump. SpinTAC Velocity Identify estimates the load inertia and the friction of the system; Eliminate or minimize the load torque before running SpinTAC Velocity Identify.

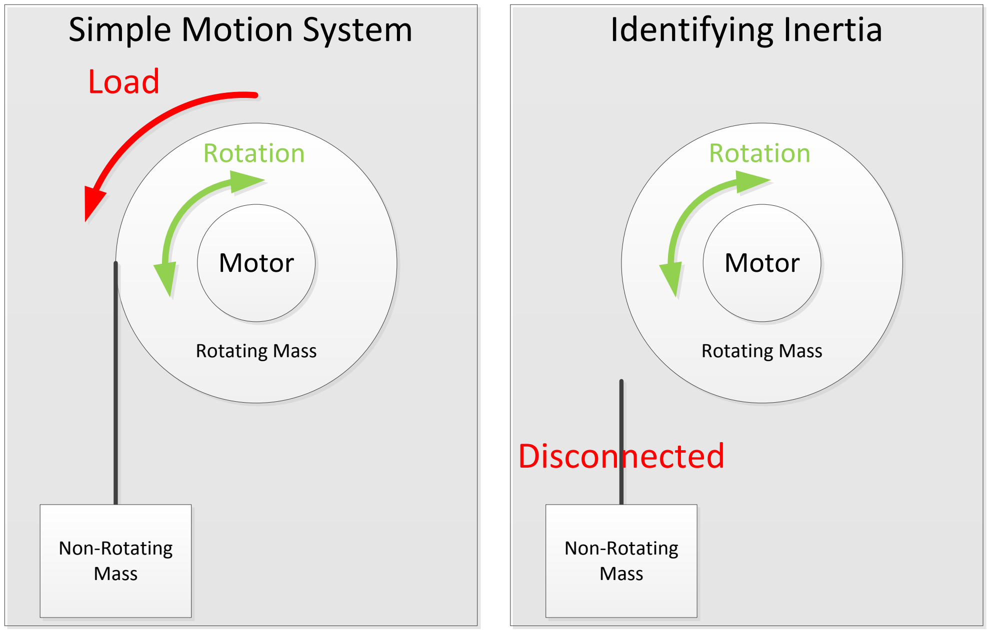


Figure 27: Example of inertia identification in a simple motion system

Figure 27 shows an example of a simple motion system. In this system, the Rotating Mass is rigidly coupled with the Motor. This means that the Rotating Mass rotates along with the motor and is considered as part of the inertia. The Non-Rotating Mass is not rigidly coupled with the motor and is considered as part of the load. During the inertia identification process, this Non-Rotating Mass should not be attached to the motor. Table 24 discusses how your system should be configured during inertia identification for common applications.

Table 24: System configuration for identifying inertia on common applications

Application	**System Configuration for Identifying Inertia**
Washing Machine	Drum should be attached to motor and free of clothes or water
Pump / Compressor	Motor should be connected to pumping / compressing apparatus, but system should have the load minimized
Conveyor Belt	Motor should be attached to conveyor, but should be free of objects.
Fan	Fan blades should be attached to motor

This lab adds the critical function calls for identifying the motor inertia. The block diagram in Figure 28 shows how the SpinTAC components fit in with the rest of the InstaSPIN library.

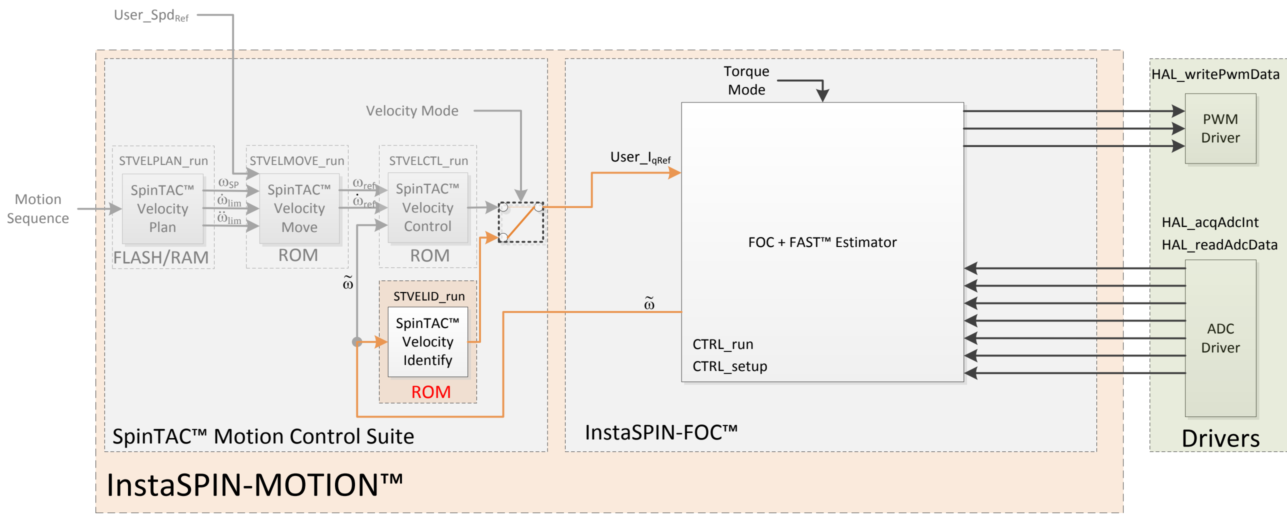


Figure 28: InstaSPIN-MOTION™ block Diagram for lab 05c

It is important to note that only the SpinTAC Velocity Identify block is highlighted in the block diagram. This indicates that only the SpinTAC Velocity Identify is included as part of this lab. This block accepts the speed feedback and outputs the Iq reference that is provided to the FOC, which is placed into torque mode. This block will generate the system inertia and friction as outputs to the user.

Prior to the Inertia Identification process, a couple of conditions need to be satisfied. These conditions have already been satisfied in the lab code, where applicable.

- The motor should not be spinning, or should be spinning very slowly.
  - The estimate of the inertia could be incorrect if it begins the torque profile while the motor is already moving.
- The InstaSPIN-FOC PI speed controller needs to be disabled.
  - SpinTAC Velocity Identify needs to provide the Iq reference in order to test the inertia. This can be achieved only if the InstaSPIN-FOC PI speed controller is disabled.
- A positive speed reference must be set in FAST.
  - The FAST estimator needs to know the spinning direction of the motor via speed reference in order for it to correctly estimate the speed. The value can be any positive value for speed reference setting.
- Force Angle must be enabled.
  - The Force Angle provides a good start from zero speed, and produces better inertia estimates.

Figure 29 is a flow chart that shows the steps required prior to beginning the inertia identification process. This chart is implemented in the code of this lab.

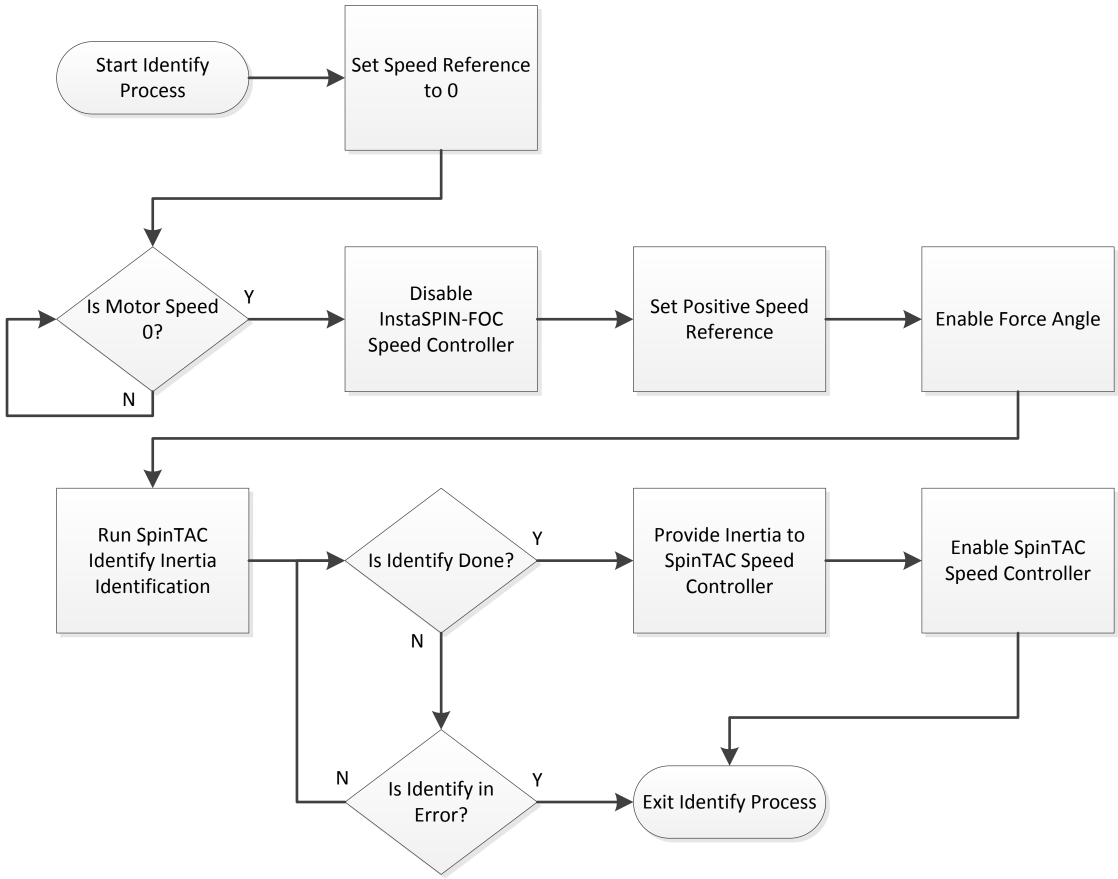


Figure 29: Flow Chart to Begin the Inertia Estimation Process

Include the Header File

A description of the new included files critical for InstaSPIN setup is shown in Table 25, below. Note that main.h is common across all labs so there will be more includes in main.h than are needed for this lab.

Table 25: Important header files needed for the motor control.

main.h	Header file containing all included files used in main.c	
	SpinTAC	
	spintac.h	SpinTAC component setup and configuration.

The critical header file for the SpinTAC components is spintac_velocity.h. This header file is common across all labs so there will be more includes in spintac_velocity.h than are needed for this lab.

Table 26: Important header filed needed for the SpinTAC components.

spintac.h	Header file containing all SpinTAC header files used in main.c	
	SpinTAC	
	spintac_vel_id.h	SpinTAC Velocity Identify structures and function declarations.

Declare the Global Structure

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.

Table 27: Global object and variable declarations important for the motor control

globals		
	SpinTAC	
	ST_Obj	The object that holds all of the structure and handles required to interface to the SpinTAC components.

Initialize the Configuration Variables

The new functions that are added to this lab to setup the SpinTAC components are listed in Table 28.

Table 28: Important setup functions needed for the motor control

setup		
	SpinTAC	
	ST_init	Initializes all variables required for configuration of the SpinTAC (ST) object.
	ST_setupVelId	Sets up the SpinTAC Identify object with default values.

Main Run-Time loop (forever loop)

The forever loop remains the same as lab 5b.

Main ISR

The main ISR calls very critical, time dependent functions that run the SpinTAC components. The new functions that are required for this lab are listed in Table 29.

Table 29: InstaSPIN functions used in the main ISR.

mainISR		
	SpinTAC	
	ST_runVelId	The ST_runVelId function calls the SpinTAC Identify object. It also handles the state machine to enable/disable components.

Call SpinTAC™ Velocity Identify

SpinTAC Velocity Identify is called from the ST_runVelId function. This function handles both the state machine of SpinTAC Velocity Identify as well as calling SpinTAC Velocity Identify. These functions are listed in Table 30.

Table 30: InstaSPIN functions used in ST_runVelId

ST_runVelId		
	EST	
	EST_getFm_pu	This function returns the speed estimate in pu from the FAST Estimator
	EST_setFlag_enableForceAngle	This function sets the ForceAngle flag in the FAST Estimator
	CTRL	
	CTRL_setSpd_ref_krpm	This function sets the speed reference to the FAST Estimator
	SpinTAC	
	STVELID_getEnable	This function gets the enable (ENB) bit in SpinTAC Velocity Identify.
	STVELID_setVelocityPositive	This function sets the goal speed (cfg.VelPos) of SpinTAC Velocity Identify
	STVELID_setTorqueRampTime_sec	This function sets the torque ramp rate (cfg.RampTime_sec) of SpinTAC Velocity Identify
	STVELID_setEnable	This function sets the enable (ENB) bit in SpinTAC Velocity Identify.
	STVELID_setVelocityFeedback	This function calls into the SpinTAC Inertia Identification to estimate the system inertia.
	STVELID_run	This function calls into the SpinTAC Inertia Identification to estimate the system inertia.
	STVELID_getDone	This function return the done (DON) bit of SpinTAC Velocity Identify
	STVELID_getErrorID	This function return the error (ERR_ID) of SpinTAC Velocity Identify
	STVELID_getTorqueReference	This function return the torque reference (Out) of SpinTAC Velocity Identify.

Lab Procedure

The code for this lab is setup according to the block diagram shown in Figure 28.

In Code Composer, build proj_lab05c. Start a Debug session and download the proj_lab05c.out file to the MCU.

- Open the command file "sw\solutions\instaspin_motion\src\proj_lab05c.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

To run the motor a couple of steps are required:

- To start the project, set the variable "gMotorVars.Flag_enableSys" equal to 1.
- To start the InstaSPIN controller, set the variable "gMotorVars.Flag_Run_Identify" equal to 1.

The InstaSPIN controller will now run the motor. In order to maintain the sensorless angle of the motor it will spin at a slow speed. This speed the motor spins at in between inertia estimations can be modified via the variable gMotorVars.StopSpeedRef_krpm. Prior to SpinTAC™ Velocity Identify inertia identification the motor will decelerate to zero speed and then begin the inertia identification process.

- Set "gMotorVars.SpinTAC.VelIdGoalSpeed_krpm" equal to the rated speed of your motor. This will ensure a more accurate inertia result.
- Set "gMotorVars.SpinTAC.VelIdRun" to 1. Watch how the motor spins for a few seconds. It is running the open-loop inertia identification provided by SpinTAC.
  - If the value of "gMotorVars.SpinTAC.VelIdErrorID" is set to non-zero then the inertia identification process has failed.
    - If the value is 2004 and the motor spun at a speed, most likely that the goal speed was set too high. Reduce "gMotorVars.SpinTAC.VelIdGoalSpeed_krpm" by half and try again.
    - If the value is 2003, most likely that the torque rate was set too low. Decrease "gMotorVars.SpinTAC.VelIdTorqueRampTime_sec" by 1.0 to have the torque be applied quicker.
    - If the value is 2006, this means that the motor did not spin through the entire inertia identification process. Decrease "gMotorVars.SpinTAC.VelIdTorqueRampTime_sec" by 1.0 to have the torque change quicker during the test.
  - The value of the motor inertia is placed into "gMotorVars.SpinTAC.InertiaEstimate_Aperkrpm"
  - The value of the motor friction is placed into "gMotorVars.SpinTAC.FrictionEstimate_Aperkrpm"

Open user.h following these steps:

1. Expand user.c from the Project Explorer window
2. Right-mouse click on user.h 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

Opening the Outline View will provide an outline of the user.h contents

In the section were you have defined your motor there should be two additional definitions to hold the inertia and the friction of your system. Place the values for inertia and friction from gMotorVars.SpinTAC.InertiaEstimate_Aperkrpm and gMotorVars.SpinTAC.FrictionEstimate_Aperkrpm as the values for these defines.

- USER_SYSTEM_INERTIA (gMotorVars.SpinTAC.InertiaEstimate_Aperkrpm)
- USER_SYSTEM_FRICTION (gMotorVars.SpinTAC.FrictionEstimate_Aperkrpm)

The motor inertia is now identified. Set "gMotorVars.SpinTAC.VelIdRun" to 1 in order to run the inertia identification process multiple times.

When done experimenting with the motor

- Set the variable "gMotorVars.Flag_Run_Identify" to 0 to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab has demonstrated how to identify a motor's inertia from within your MotorWare project. A motor's inertia has been identified and the value was entered into the user.h file. The recorded inertia value will be used in the following labs as we begin to build our motion system.

Lab 5d - InstaSPIN-MOTION Speed Controller

Abstract

The industry standard PI speed controller has a number of inherent deficiencies:

- Tuning parameters are interdependent and create tuning challenges.
- PI controllers may need to be tuned for many speed and load points

The SpinTAC Velocity Control solves these challenges. SpinTAC™ features Active Disturbance Rejection Control (ADRC), which actively estimates system disturbances and compensates for them in real time. The SpinTAC Velocity Control also features a single tuning parameter, bandwidth, which determines the stiffness of the system and dictates how aggressively the system will reject disturbances. Once tuned, the controller typically works over a wide range of speeds and loads.

In this lab, you will learn how to replace the InstaSPIN-FOC speed controller with the SpinTAC Velocity Control in your MotorWare project. In follow-on labs, you will learn how to tune the speed controller and optimize system performance. Additional information about the SpinTAC Velocity Control can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section InstaSPIN-MOTION Controllers).

Introduction

Inertia identification is the first step in enabling the SpinTAC™ speed controller. This lab uses the motor inertia that was identified in Lab 5c - InstaSPIN-MOTION Inertia Identification and stored in the user.h file. The inertia value is used by the SpinTAC Velocity Control to determine how strongly to respond to the disturbances in the system. This lab focuses on using the SpinTAC Velocity Control to spin your motor.

Prerequisites

The motor inertia value has been identified and populated in user.h. This process should be completed inLab 5c - InstaSPIN-MOTION Inertia Identification.

Objectives Learned

Use the SpinTAC Velocity Control to replace the InstaSPIN-FOC speed controller

Background

This lab has new API function calls for the SpinTAC™ speed controller. Figure 30 shows the block diagram for this project.

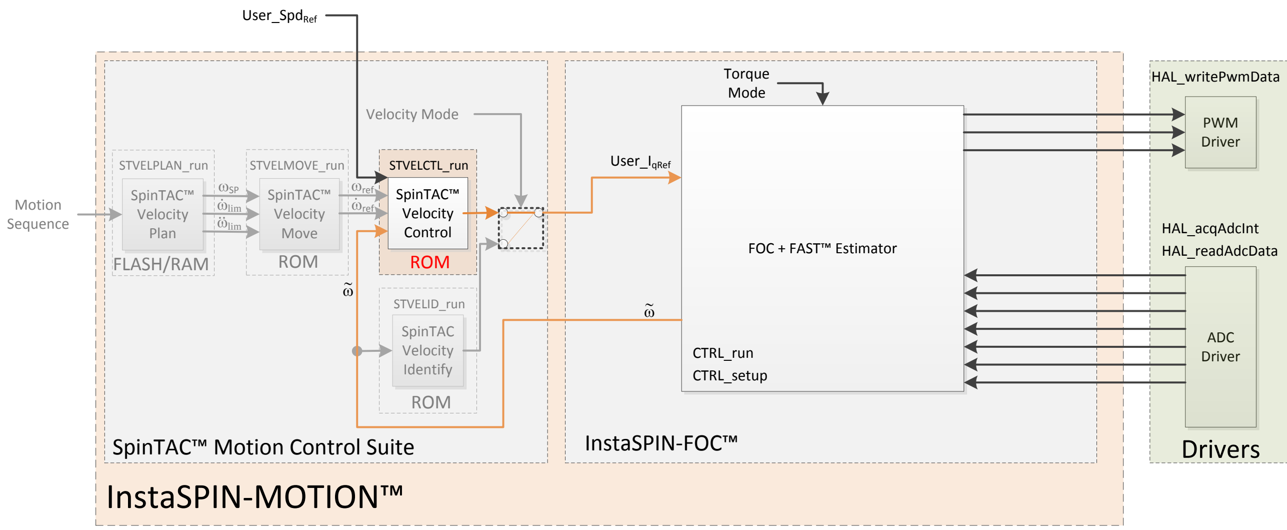


Figure 30: InstaSPIN-MOTION™ block diagram for lab 05d

The difference between the block diagram for lab 5d (Figure 30) and lab 5c (Figure 28) is that the SpinTAC Velocity Identify component has been removed from this project. Now that the system inertia has been identified, the SpinTAC Velocity Control component can be placed in the project. This block accepts the user speed reference and the speed feedback and outputs the Iq reference that is provided to the FOC, which is placed into torque mode.

Project Files

There are no new project files.

Include the Header File

The critical header file for the SpinTAC components is spintac_velocity.h. This header file is common across all labs so there will be more includes in spintac_velocity.h than are needed for this lab.

Table 31: Important header files needed for SpinTAC components

spintac.h	Header file containing all SpinTAC header files used in main.c	
	SpinTAC	
	spintac_vel_ctl.h	SpinTAC Velocity Control structures and function declarations.

Declare the Global Structure

There are no new global object and variable declarations.

Initialize the Configuration Variables

The new functions added to the lab to configure SpinTAC components are listed in Table 32.

Table 32: InstaSPIN functions needed to setup the motor control

setup		
	SpinTAC	
	ST_setupVelCtl	Sets up the SpinTAC Velocity Controller object with default values.

Main Run-Time loop (forever loop)

There are no changes in the main run-time forever loop for this lab.

Main ISR

The main ISR calls very critical, time dependent functions that run the SpinTAC components. The new functions that are required for this lab are listed in Table 33.

Table 33: InstaSPIN functions used in the main ISR

mainISR		
	SpinTAC	
	ST_runVelCtl	The ST_runVelCtl function calls the SpinTAC Velocity Controller object.

Call the SpinTAC™ Speed Controller

The function ST_runVelId and the supporting logic have been removed from this project. This was done because the SpinTAC Inertia Identification only needs to be done once, during development, and is not needed in the final system. ST_runVelCtl has been added to this project and it handles calling the SpinTAC Velocity Control. Table 34 shows the functions that are called in the ST_runVelCtl function.

Table 34: InstaSPIN functions used in ST_runVelCtl

ST_runVelCtl		
	EST	
	EST_getFm_pu	This function returns the speed feedback in pu from the FAST Estimator
	SpinTAC	
	STVELCTL_setVelocityReference	This function sets the velocity reference (VelRef) of SpinTAC Speed Controller
	STVELCTL_setAccelerationReference	This function sets the acceleration reference (AccRef) of SpinTAC Speed Controller
	STVELCTL_VelocityFeedback	This function sets the velocity feedback (VelFdb) of SpinTAC Speed Controller
	STVELCTL_run	This function calls into the SpinTAC Velocity Controller to control the system velocity.
	STVELCTL_getTorqueReference	This function returns the torque reference (Out) of the SpinTAC Speed Controller
	TRAJ	
	TRAJ_getIntValue	This function returns the velocity reference from the ramp generator
	CTRL	
	CTRL_setIq_ref_pu	This function sets the Iq reference to the Iq Current Controller

Lab Procedure

After verifying that user.h has been properly updated with your motor's inertia, build lab5d, connect to the target and load the .out file.

- Open the command file "sw\solutions\instaspin_motion\src\proj_lab5d.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

To run the motor a couple of steps are required:

- To start the project, set the variable "gMotorVars.Flag_enableSys" equal to 1.
- To start the InstaSPIN controller, set the variable "gMotorVars.Flag_Run_Identify" equal to 1.
- At this point the motor can be spun at any speed. The motor inertia and friction will be loaded automatically into SpinTAC Velocity Control. The values "gMotorVars.SpinTAC.Inertia_Aperkrpm" and "gMotorVars.SpinTAC.Friction_Aperkrpm" should reflect the values you put into user.h.
- The process for setting speed references to the SpinTAC Velocity Control is the same as setting speed references to the PI controller. Update the value in "gMotorVars.SpeedRef_krpm" with the speed you would like the motor to run.
- The acceleration can also be modified by adjusting the value in "gMotorVars.MaxAccel_krpmps. Verify that the motor responds to speed reference changes the same as in previous labs
- It is important to notice that in this lab the SpinTAC Velocity Control is not tuned, this step will be accomplished in the next lab

When done experimenting with the motor:

- Set the variable "Flag_Run_Identify" to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how to replace the InstaSPIN-FOC speed controller with the SpinTAC™ speed controller. It also demonstrated that interfacing to the SpinTAC Velocity Control is no different than interfacing to a PI speed controller. This lab is the basis for all subsequent labs where the more advanced features of the SpinTAC™ library will be used to tune and optimize your system.

Lab 5e - Tuning the InstaSPIN-MOTION Speed Controller

Abstract

InstaSPIN-MOTION provides maximum control with minimal effort. InstaSPIN-MOTION features the SpinTAC™speed controller with Active Disturbance Rejection Control (ADRC). In real-time, ADRC estimates and compensates for system disturbance caused by:

- Uncertainties (e.g. - resonant mode)
- Nonlinear friction
- Changing loads
- Environmental changes

The SpinTAC Velocity Control presents better disturbance rejection and trajectory tracking performance than a PI controller, and can tolerate a wide range of inertia change. This means that SpinTAC™ improves accuracy and minimizes mechanical system duress.

With single coefficient tuning, SpinTAC™ allows you to quickly test and tune your velocity control from soft to stiff response. This single gain (bandwidth) works across the entire variable speed and load range of an application, reducing complexity and system tuning.

In this lab, you will tune the SpinTAC Velocity Control to obtain the best possible system performance. Additional information about the SpinTAC Velocity Control can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section InstaSPIN-MOTION Controllers).

Introduction

The SpinTAC Velocity Control features a single tuning parameter, bandwidth. Once the motor inertia has been identified, you're ready to tune the controller. This tuning process will allow you to quickly zero in on the optimal control setting by adjusting a single variable (bandwidth).

If you have completed Lab 5b – Tuning the Speed Loop, you will notice that the tuning process in this lab for the SpinTAC Velocity Control will be much faster.

Prerequisites

This lab assumes that the motor parameters and inertia have been identified (as part of Lab 5c - InstaSPIN-MOTION Inertia Identification), and that you have used the SpinTAC Velocity Control to spin your motor (Lab 5d - InstaSPIN-MOTION Speed Controller).

Objectives Learned

- Quickly tune the SpinTAC™ controller for your motor
- Note the differences between tuning a PI controller and tuning the SpinTAC™ controller
- Realize the advanced control capabilities and enhanced performance characteristics of the SpinTAC™ speed controller

Background

This lab has no new API function calls. Figure 31 shows the block diagram for this project.

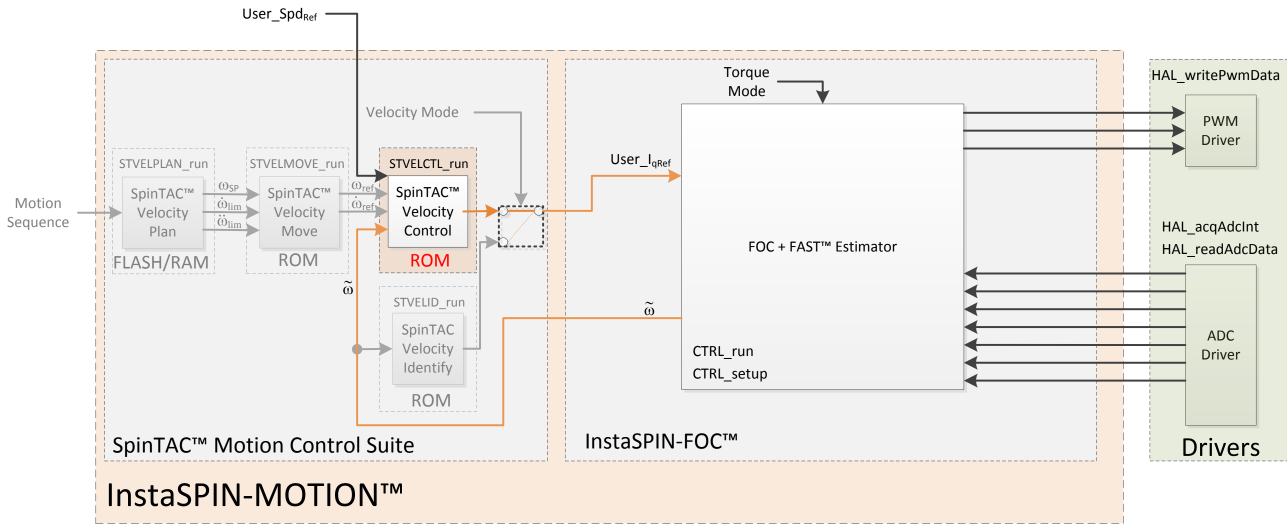


Figure 31: InstaSPIN-MOTION block diagram for lab 05e

The block diagram for this lab has not been modified from lab 5d. The difference between these two labs is that as part of this lab you will be tuning the SpinTAC Velocity Controller.

The single tuning parameter used by the SpinTAC Velocity Control is called bandwidth. Bandwidth identifies how quickly the system will respond to changes. The bandwidth of the SpinTAC Velocity Control is adjusted with a scalar value called BwScale. The SpinTAC Velocity Control then uses Active Disturbance Rejection Control to automatically compensate for disturbances in your system. Figure 32 shows the controller response when a load is applied (system disturbance), and when a load is removed (system disturbance). When load torque is applied to the system the speed will dip until the controller can return the speed to the setpoint and when the load torque is removed from the system the speed will peak until the controller can return the speed to the setpoint. Notice that the controller to responds more quickly to these disturbances as the bandwidth is increased.

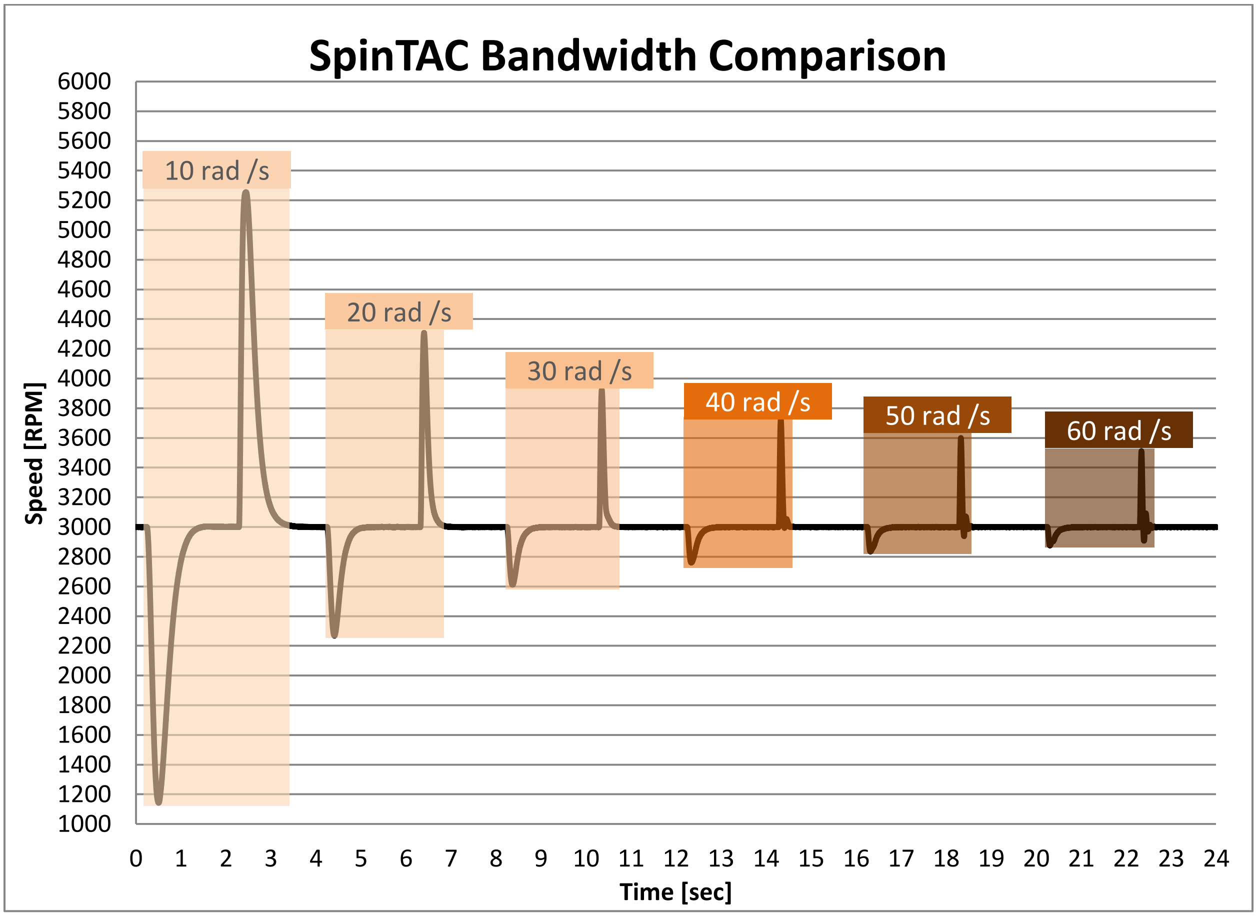


Figure 32: SpinTAC Velocity Control Response to Increasing Bandwidth

Project Files

There are no new project files.

Include the Header File

There are no new includes.

Declare the Global Structure

There are no new global object and variable declarations.

Initialization the Configuration Variables

Nothing has changed in the initialization and setup from the previous lab.

Main Run-Time loop (forever loop)

Nothing has changed in the forever loop from the previous lab.

Main ISR

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

Call the SpinTAC™ Speed Controller

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

Lab Procedure

Build lab5e, connect to the target and load the .out file.

- Open the command file “sw\solutions\instaspin_motion\src\proj_lab05e.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

To run the motor a couple of steps are required:

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To start the current loop controller, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

In order to properly hold zero speed, the ForceAngle feature needs to be turned off. Set “gMotorVars.Flag_enableForceAngle” to 0.

Make sure the motor is at zero speed by setting “gMotorVars.SpeedRef_krpm” to 0. Once the motor is at zero speed, manually rotate the motor shaft with your hand to feel how tightly the motor is holding zero, this is an indication of how aggressively the motor is tuned.

- Increase the bandwidth scale “gMotorVars.SpinTAC.VelCtlBwScale” in steps of 1.0, continuing to feel how tightly the motor is holding zero speed.
  - For motors where the shaft is not accessible, give the motor a speed reference via “gMotorVars.SpeedRef_krpm”. Change that speed reference and monitor how aggressively the controller tries to achieve the new speed setpoint.
- Once the SpinTAC Velocity Control is tightly holding zero the bandwidth scale has been tuned.

This process has tuned the bandwidth scale for zero speed. It still needs to be verified at higher operating speeds. Occasionally a bandwidth scale can work very well at zero speed but cause instability at higher speeds.

- Set the maximum motor speed as a reference in “gMotorVars.SpeedRef_krpm”
- If the motor speed begins to oscillate or show other signs on instability, decrease “gMotorVars.SpinTAC.VelCtlBwScale” until it no longer oscillates. It typically only needs to be decreased by 10-15%

At the top of the USER MOTOR defines section in user.h there is a parameter called USER_SYSTEM_BANDWIDTH_SCALE. Update the value for this define with the value you identified during the tuning process.

- USER_SYSTEM_BANDWIDTH_SCALE (gMotorVars.SpinTAC.VelCtlBwScale)

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how easy it is to tune the SpinTAC Velocity Control for your motor. The single parameter tuning of the SpinTAC Velocity Control alleviates the pain of tuning PI regulators at different speed and load configurations. The bandwidth scale value found in this lab will be used in future labs to showcase the SpinTAC™ controller's performance.

Lab 5f - Comparing Speed Controllers

Abstract

The SpinTAC Velocity Control bundled with InstaSPIN-MOTION shows remarkable performance when compared against a traditional PI controller. This lab will lead you through a comparison of these two controllers. Additional information about the comparison of speed controllers can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see Optimal Performance section 13.4).

Introduction

In addition to a single tuning parameter the SpinTAC Velocity Control is also a more robust controller than a traditional PI controller. This can be observed by setting step responses into the two controllers.

Prerequisites

This lab assumes that the SpinTAC Velocity Control has been tuned (Lab 5e - Tuning the InstaSPIN-MOTION Speed Controller) and that the PI speed controller has been tuned (Lab 5b – Tuning the Speed Loop).

Objectives Learned

- See how the SpinTAC Velocity Control provides superior speed regulation then a PI controller

Background

This lab has no new API function calls. Figure 33 shows the block diagram for this project.

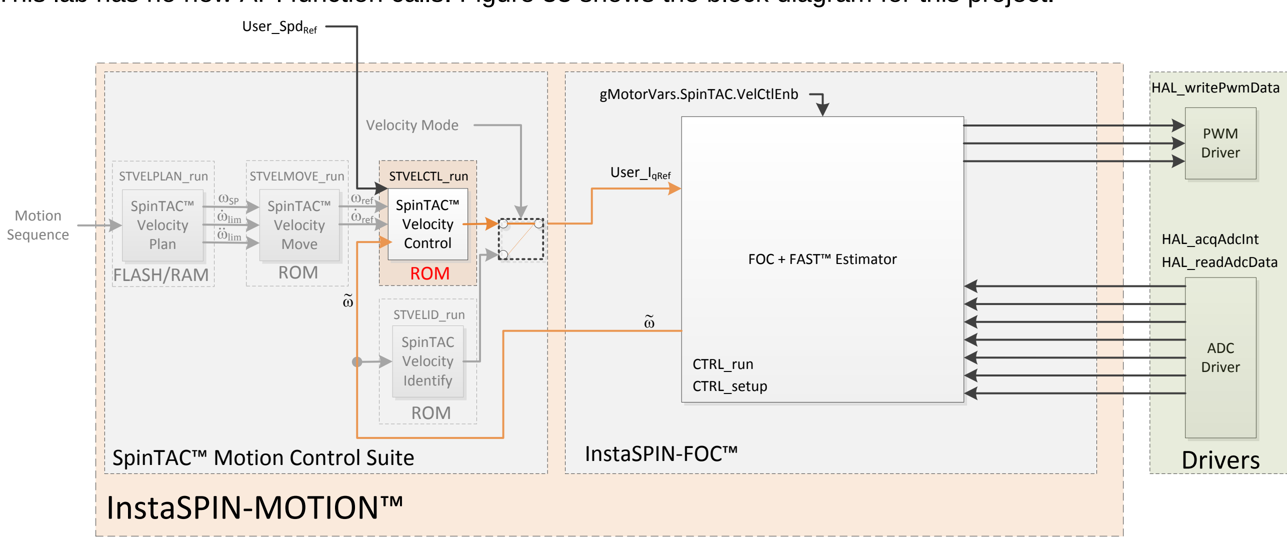


Figure 33: InstaSPIN-MOTION block diagram for lab 05f

In this lab the Traj Ramp and Speed PI blocks provided by InstaSPIN-FOC can be re-enabled. This is done in order to compare the control performance between the SpinTAC Velocity Control and the Speed PI. The control selection is provided by the switch gMotorVars.SpinTAC.VelCtlEnb which is discussed in the Lab Procedure section of this document.

We will use the graphing features of Code Composer in order to visually compare the two controllers. It will allow us to see the step response of the two speed controllers.

Project Files

There are no new project files.

Includes

There are no new includes.

Global Object and Variable Declarations

There are new global variables that are designed to support the graphing features of Code Composer..

Initialization and Setup

Nothing has changed in the initialization and setup from the previous lab.

Main Run-Time loop (forever loop)

This version of the forever loop will allow you to switch between the SpinTAC Velocity Control and the PI speed controller. This is controlled via the "gMotorVars.SpinTAC.VelCtlEnb" variable. When this variable is set to 1 SpinTAC is providing the speed control, when it is set to 0 PI is providing the speed control.

Main ISR

This lab has been modified to store values into a buffer for graphing purposes. It will store the speed feedback into "Graph_Data" and the Iq reference into "Graph_Data2". This has been done so that the actual signals for speed feedback and Iq reference can be compared visually in Code Composer.

Call the SpinTAC™ Speed Controller

The ST_runVelCtl function has been updated to set the integrator term in the InstaSPIN-FOC PI speed controller. This is done in order to allow the speed controller to seamlessly switch between the SpinTAC Velocity Control and the PI speed controller.

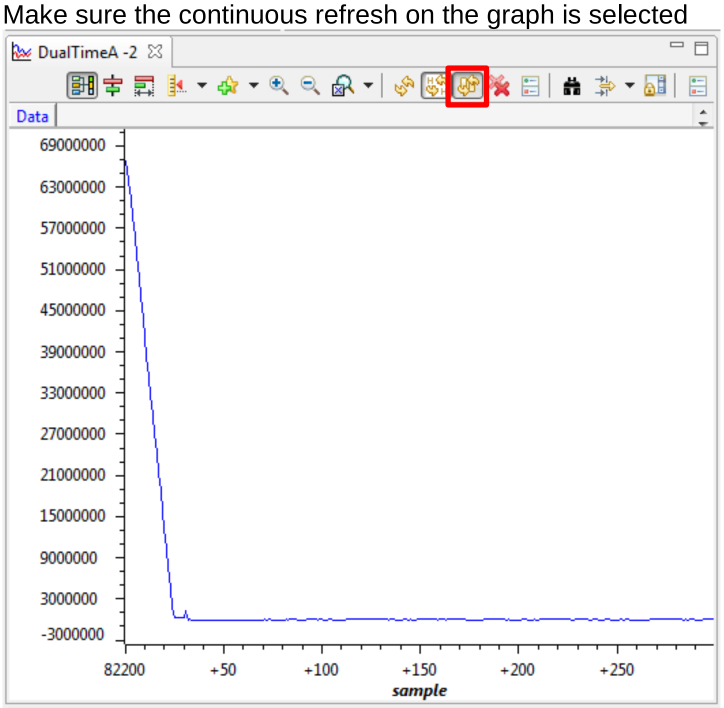
Table 35: InstaSPIN funcions used in ST_runVelCtl

ST_runVelCtl		
	CTRL	
	CTRL_setIq_ref_pu	This function sets the Iq reference to the Iq Current Controller
	CTRL_getKp	This function gets the Kp gain for a PI controller
	CTRL_setKi	This function gets the Ki gain for a PI controller
	CTRL_setUi	This function sets the integrator term of a PI controller

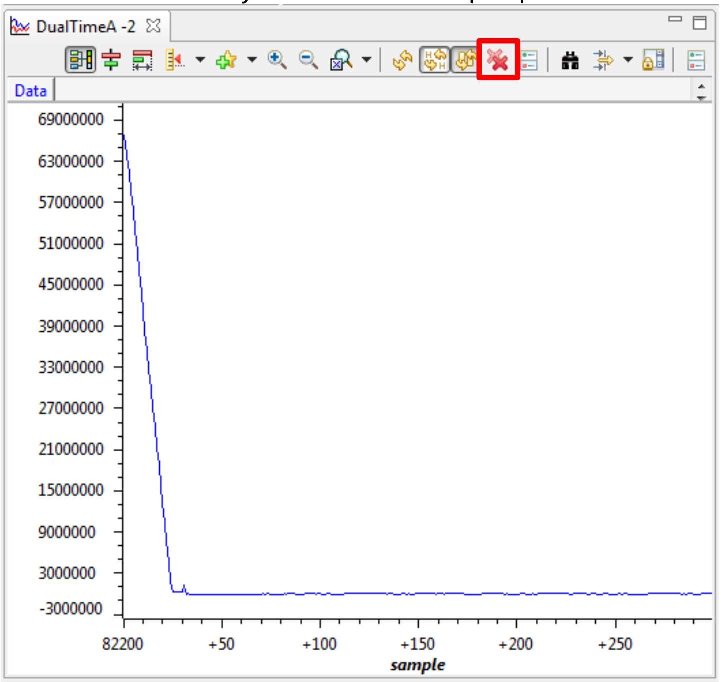
Lab Procedure

In Code Composer, build proj_lab05f. Start a Debug session and download the proj_lab05f.out file to the MCU.

- Open the command file “sw\solutions\instaspin_motion\src\proj_lab05f.js” via the Scripting Console
    - This will add the variables that we will be using for this project into the watch window
- Select “Tools -> Graph -> Dual Time” to open two plot windows
    - To configure these plot windows select “Import” on the pop-up window
    - Open the graph file “sw\solutions\instaspin_motion\src\proj_lab05f.graphProp”
    - This will configure the two plots that will be used in this lab
        - DualTimeA displays the speed feedback
        - DualTimeB displays the torque (Iq) reference generated by the speed controller
    - These two plots will assist you in comparing the two speed controller. They will serve as a visual indicator of how each controller is performing when given a step input.
    - Make sure the continuous refresh on the graph is selected



- In order to reset the y-axis scale on the plot please click the button with the red ‘x’



- 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

The graphs will allow you to visually see the speed response of the speed controllers and will allow you to more easily compare the performance of the PI speed controller and the SpinTAC speed controller

To run the motor a couple of steps are required:

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To start the current loop controller, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

To provide a fast reference to the control loops, the acceleration that the ramp generator is using needs to be faster than the motor can accelerate. The acceleration in “gMotorVars.MaxAccel_krpmps” needs to be set to at least 20.0.

Set a speed reference in “gMotorVars.SpeedRef_krpm” in order to get the motor spinning. Each time you set a new speed reference it will update the graph with the controller response to the new speed reference.

Switch between SpinTAC & PI using the variable “gMotorVars.SpinTAC.VelCtlEnb”. If this variable is set to 1 it will use SpinTAC to control the speed. Setting this to 0 will use PI to control the speed.

Compare the response of the two speed controllers for different speed references. You will notice that the SpinTAC controller will have less overshoot and will return to the target speed with less oscillation than the PI controller.

The two controllers can also be tuned in this lab to allow for a more in-depth comparison.

- To tune PI, adjust the values “gMotorVars.Kp_spd” and “gMotorVars.Ki_spd”. Note that these parameters may need to be modified to support different speeds and loads.
- To tune SpinTAC, adjust the value “gMotorVars.SpinTAC.VelCtlBwScale”. Note that this single tuning parameter typically works across the entire operating range.

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed the advanced performance of the SpinTAC Velocity Control in a direct and head-to-head comparison with a PI controller.

Lab 5g – Adjusting the InstaSPIN-FOC Speed PI Controller, with fpu32

Abstract

This lab runs Lab 5b with floating point unit enabled. This lab only applies to 6x devices, as it has a floating point unit.

Objectives Learned

Tuning the speed controller with fpu32 enabled.

Lab Procedure

Follow the exact same procedure as in Lab 5b.

Conclusion

We conclude that the libraries in ROM also work when fpu32 is enabled in 6x devices.

Lab 5h – Step Response Generation & Graphing for Controller Tuning

Abstract

When using InstaSPIN-FOC/MOTION in default configuration, the current controller parameters are automatically set based on the bandwidth of the system as seen in the function USER_calcPIgains found in the file user.c.

Kp_Iq = _IQ((0.25*Ls_q*fullScaleCurrent)/(ctrlPeriod_sec*fullScaleVoltage));
Ki_Iq = _IQ(RoverLs_q*ctrlPeriod_sec);

Notice that the Kp term is scaled by a factor of ¼ from the ideal bandwidth to soften the PI controller and allow for more stability. In certain applications the Kp term may need to be increased for best performance. In others it may need to be reduced even further to give a softer response or reduce current control hunting which can produce audible noise.

Also note that both Kp and Ki terms rely on an accurate Ls value. In some motors Ls may be difficult to identify and in most motors Ls will actually vary at different frequencies and especially under increasing current as the magnetics saturate.

The speed controller gains are not configured to the motor at all. Starting gains are chosen based on a simple "rule of thumb" in an attempt to get a reasonable stable starting point for most motors. These gains are set in the function CTRL_setParams in the file ctrl.c

Kp_spd = (0.02 * USER_IQ_FULL_SCALE_FREQ_Hz *
USER_MOTOR_MAX_CURRENT) / USER_IQ_FULL_SCALE_CURRENT_A)
Ki_spd = (2.0 * USER_CTRL_PERIOD_sec * USER_IQ_FULL_SCALE_FREQ_Hz *
USER_MOTOR_MAX_CURRENT) / USER_IQ_FULL_SCALE_CURRENT_A

The define values can be found in the files user.h

Testing and adjusting the controller gains allows you to optimize the system for your specific requirements. Using a step response it is possible to test the effects of the controller gains in your application and choose the values which meet your system response, stability, and efficiency targets. When generating the step responses for the speed and the current signals it is done using the framework already available in the MotorWare defines. An overview of the used variables are shown in the below image.

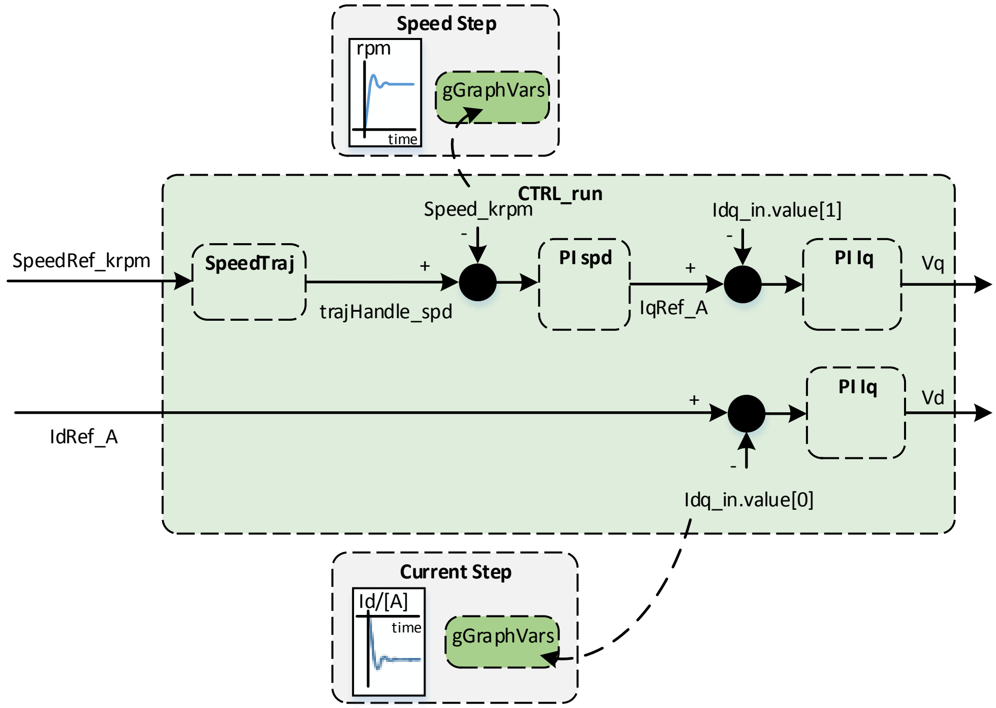


Figure 34: Cascaded speed and current controller of InstaSPIN-FOC

The idea is then to generate a step response in the Id current controller to be almost independent of load torque changes. The Id step response can be used as an estimate of the Iq step response too. Doing this removes the need to have a load emulator like a dynamometer when trying to generate the step response in Iq.

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

Current Step Response Generation
As mentioned previously, the current controller gains are automatically set based on the bandwidth of the system. Depending on customer use it can be beneficial to the system stability and performance to re-optimize. Cases include very low speed, high load applications, extremely high speed motors, and very low current motors.

Speed Step Response Generation
The default PI controller is just a starting point and must be adjusted to ensure best possible performance of the system.

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

Background
The purpose of this lab guide is to enable the user to generate step response of the current PI controller and the speed PI controller, the information of the step response can be used improve the system performance and stability. Proj_lab05h is built upon proj_lab10 (which includes over-modulation). It is not required that you are familiar yet with proj_lab10 to run this lab.

Import proj_lab05h from the hardware and f280x specific MotorWare folder. If you do not see proj_lab05h available, review the EVM Project Support table on page 2. If your evaluation board is not supported, a porting of the project may be done.

After importing the project look in the file proj_lab05h.c.

All graphs shown in this lab guide has been done using the low voltage Anaheim_BLY172S motor with no load.

Includes
New include files

graph.h	Header file containing all defines and external function definitions used to record data	
graph.c	C-file containing functions used to record data	

Global Object and Variable Declarations
The global and variable declarations that are added extra in comparison to proj_lab10 are listed in the Table below.

Globals		
	GRAPH	
	GRAPH_Vars_t	Not needed for the implementation of the graph functions, but in

		the project this structure contains all of the flags and variables to turn on and capture data. The structure defined by this declaration will be put into the CCS watch window

Globals		
	STEP	
	STEP_Vars_t	Structure needed to generate a predefined step for both speed and current step responses

Initialization and Setup

Nothing has changed between proj_lab10 and this lab for initialization and setup.

Main Run-Time loop (forever loop)

In the forever loop the following function has been added to generate the step response for easy use of the graph tool.

Forever loop		
	STEP	
	GenerateStepResponse	Set one variable to start the step response data acquisition

mainISR

The main ISR calls very critical, time dependent functions that run the FOC and FAST observer. The functions that are added extra in comparison to proj_lab10 are listed in the Table below.

mainISR		
	GRAPH	
	GRAPH_Data	Record wanted Iq value into gGraphVars.Buffer_data[x].data

When adding the GRAPH_Data function the data gather function is added. To add other data options the following needs to be done.

```
case 3:
        GRAPH_Data_Gather(pGraphVars,GRAPH_Buffer_NR1,EST_getAngle_pu(((CTRL_Obj
        *)ctrlHandle)->estHandle),pMotorVars->IqRef_A);
        GRAPH_Data_Gather(pGraphVars,GRAPH_Buffer_NR0,
        CPU_USAGE_getMaxDeltaCntObserved(cpu_usageHandle),pMotorVars->IqRef_A);
break;
```

Important to notice here is that the data gathered is in _IQ24 format, any data put into the function needs to be converted accordingly or the scale _IQ24 scale factor on the graph tool is changed to fit the new data format.

The above code added an additional data gather case in to chosen, in this you can see the estimator angle and the CPU usage per ISR, this can be used to see if the interrupt is finished before the next interrupt happen.

Lab Procedure
When using the project it is assumed that the motor is identified and the motor parameters have been saved in user.h.

Build lab5h, connect to the target and load the .out file.

- Add the appropriate watch window variables by calling the script “proj_lab05b.js”.
- Enable the real-time debugger.
- Click the run button.
- Enable continuous refresh on the watch window.

Using the Project
Now the project is ready to be used.

gMotorVars.Flag_enableOffsetcalc should be enabled if you have not already saved your OFFSET values in user.h per proj_lab03. gMotorVars.Flag_enableRsRecalc may optionally be enabled (recommended for best results).

To run the identified motor, as usual set the variables

- *“Flag_enableSys”* = true
- *“Flag_Run_Identify”* = true

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

“SpeedRef_krpm”

The actual motor speed can be seen in the variable:

“Speed_krpm”

The variable “*gGraphVars.Buffer_mode”* selects between current step response generation or speed step response generation.

- Current step response generation is done setting the variable to 1.0 (default)
- Speed step response generation is done setting the variable to 2.0

Graph definitions

gGraphVars	struct _GRAPH_Vars_t_	{...}
Buffer_reset	unsigned long	0
Buffer_counter	unsigned long	256
Buffer_tick_counter	unsigned long	0
Buffer_tick	unsigned long	1
Buffer_mode	unsigned long	2
Buffer_reset_wait	unsigned char	.
Buffer_previous	long	33554432
Buffer_data	struct _Buffer_[2]	0x00BA4E@Program

Figure 35: gGraphVars

The variables are used in the following way.

"Buffer_reset" used to reset the trigger for the data gathering algorithm
"Buffer_counter" used as an index to write into the buffer array structure
"Buffer_tick_counter" used to count the interrupts
"Buffer_tick" defines how many interrupts happen per graph write
"Buffer_mode" can be used to define different values to record for each mode during run time
"Buffer_previous" used to store the trigger value for the next rewrite of the buffer in _iq
"Buffer_data" the array where the recorded data is stored

When compiling the code in the file graph.h the definition.

```
#define BUFFER_NR 2                 // Number of data arrays
#define BUFFER_SIZE 256             // Size of data arrays
```

The defines are used to set the size of the chosen array and how many arrays are available.

If the following error message happens, the graph size is too big for the used C2000 device.

```
>> Compilation failure
   program will not fit into available memory.  placement with
   alignment/blocking fails for section ".text" size 0x3896 page 0.  Available
   memory ranges:
   RAML0_L8     size: 0xb800       unused: 0x344a       max hole: 0x344a
error #10010: errors encountered during linking; "proj_lab05h.out" not built
gmake: *** [proj_lab05h.out] Error 1
gmake: Target `all' not remade because of errors.
```

Now lower the value of the value BUFFER_SIZE and BUFFER_NR until the code will fit the C2000 device and the error message disappears.

Setup and use the graph window

To ensure the full functionality of the graph window use version 6.1.1 of code composer studio or newer. If an earlier version is used some features might not be available.

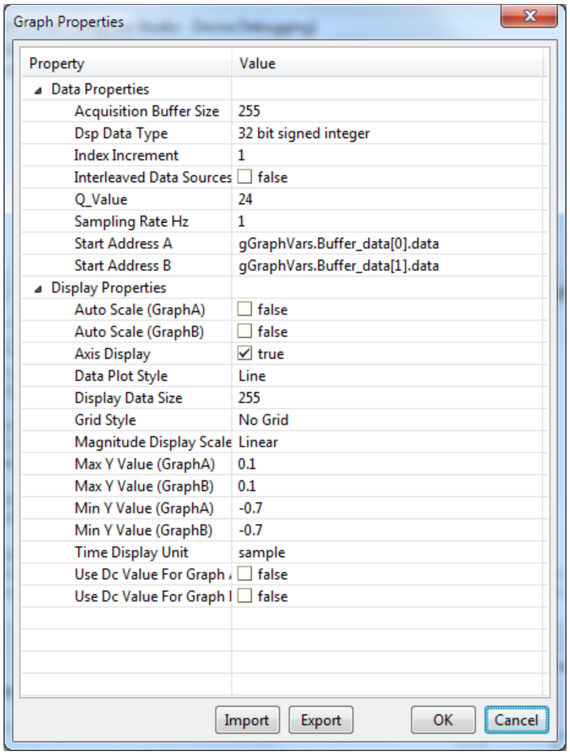
To setup the graph window click

Tools->Graph->(Single time or Dual time)

Click the Import button and select the appropriate single or dual time file from ..\sw\solutions\instaspin_foc\src

proj_lab05h_dual_view.graphProp

Please note that the Auto Scale (GraphA) and Auto Scale (GraphB) values in the following figure are set to false. This enables the option of Max Y value and Min Y value of GraphA and GraphB.



Then select OK. The graphs will import into CCS. Arrange the windows as you like in your workspace.

The graph windows display the following depending on the Buffer_mode

Current step response "*Buffer_mode"* equal 1:

Graph data	**proj_lab05h_dual_view.graphProp**
Single time or View A	Showing Id current measurement
View B	Not showing any data

Speed step response *Buffer_mode* equal 2:

Graph data	**proj_lab05h_dual_view.graphProp**
Single time or View A	Showing speed krpm
View B	Showing Iq current measurement

Using the Graph window

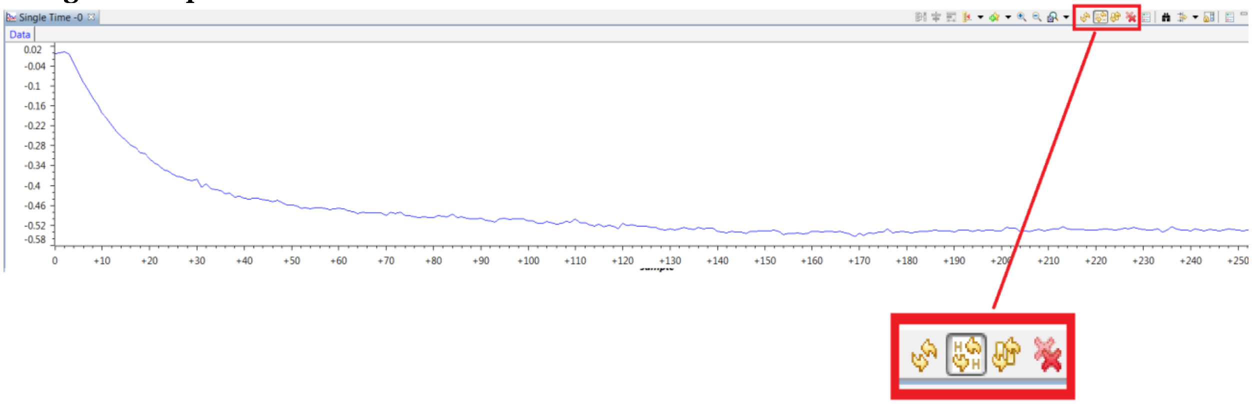
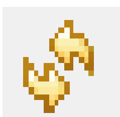
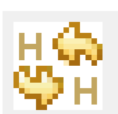


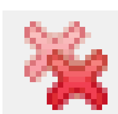
Figure 36: Graph window

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


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


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

Refresh  the graph every time you halt the debug session.


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

Step response generation for PI current controller

In InstaSPIN-FOC the Speed and current PI controller are cascaded, so to start generating the step response the user has to begin with the current controller.

- “gGraphVars.Buffer_mode” = 1.0 (default)

For your real application it will be important to test the step response generation while running with loads similar to those in the real system, but you can start testing the system with a lightly loaded or no load system. For this example we are running the low voltage Anaheim_BLY172S motor with no load and we set the speed to a relatively low speed around ~10% of rated where the motor is running stable. The graphs measured are done on the BoostXL-DRV8305EVM.

- “SpeedRef_krpm” = 0.5

From 0 RPM the control system is either using an unknown angle value or a ForceAngle value, so control loop step response generation should be done away from low-speed operation and with Flag_enableForceAngle disabled.

To generate the step response the following structure has been defined.

- “gStepVars” // all values Q24

To generate the current step the Id current is used. Because both Id and Iq controllers will use the same gains, we can keep test the effects of the step response on Id more simply without having to disconnect the output of the speed controller from IqRef_A. The Id current is set using the variable

- “IdRef_A”

Which is the reference current that is put on the direct axis of the magnetic field. Normally this is set to 0 A to orient the stator flux to the rotor. A negative “IdRef_A” results in field weakening. For more details on this topic see proj_lab09a in this lab guide.

The step chosen with the Anaheim BLY172S motor was be -0.5A, which was around 10% of the maximum rated Id current for that motor.. For your specific motor you may choose e.g. a step of 10% of the rated Id current as an initial value.

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

- “gStepVars.IdRef_Default” = 0.0 // Default starting value of IdRef_A
- “gStepVars.IdRef_StepSize” = -0.5 // Step size of IdRef_A

Now set the following variable to:

- “gStepVars.StepResponse” = 1

This will now generate the data of a step response on *IdRef_A* with the step defined in the vaiable “gStepVars.IdRef_Value[1]”.

In this example the step response for the Anaheim_BLY172S motor is as follows:

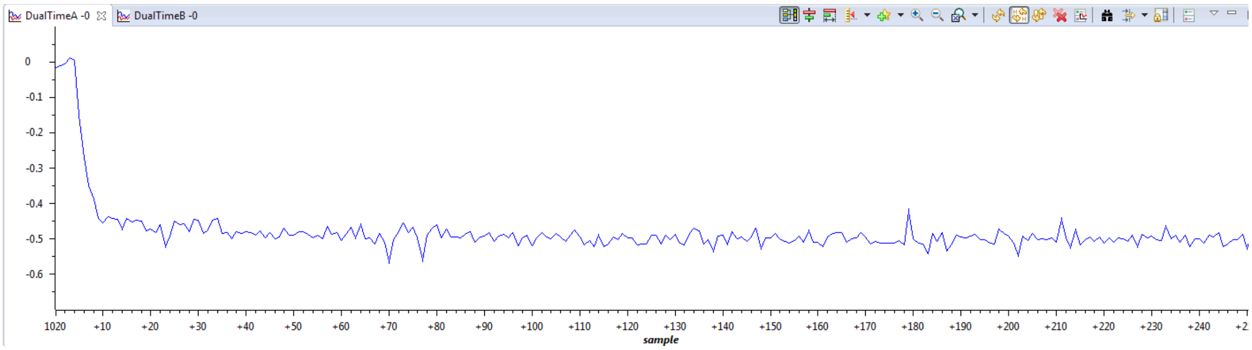


Figure 37: Anaheim_BLY172S Current step response at 500 rpm with automatic Kp/Ki values

The above response was done with the default calculations of:

- *"gMotorVars.Kp_Idq"* = 2.659803629
- *"gMotorVars.Ki_Idq"* = 0.03863072395

As these gains are defined using the motor parameters and other system settings in user.h these values may differ from yours.

To tune your controller watch the impact on the step response of the current controller changing the following variables from gMotorVars:

- *"gMotorVars.Kp_Idq"*
- *"gMotorVars.Ki_Idq"*

Now you change either Kp or Ki one at a time. When one of the values has been changed, generate a new step response by setting the variable "gStepVars.StepResponse" to 1 to see the effect.

Every time a new step response has been generated refresh the graph window to see the new step response (or turn on continuous refresh).

As an example, increase the value of Kp to get to the target IdRef_A value with a quick (stiff) response. Increasing it too much results in overshoot. Reducing it too much results in a very slow (soft) response.

The Ki value is calculated based on the ratio of Rs to Ls, and the controller period, so it will be as accurate as those values were identified. In most cases the Ki value does not need to be changed, but you can try out values (half the default, twice the default) to see the effect on the settling time or ringing of the step response.

Below you can see some different step responses generated with the Anaheim_BLY172S motor:

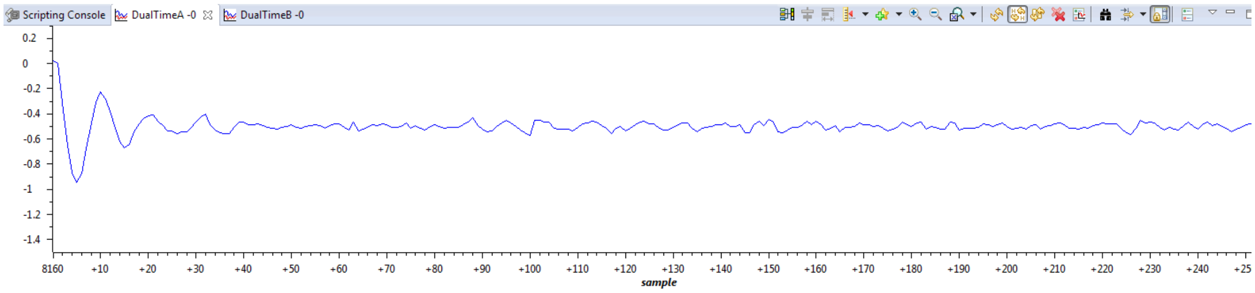


Figure 38: Current step response showing overshoot with slight instability with too high Kp(3.5) and too high Ki(0.5)

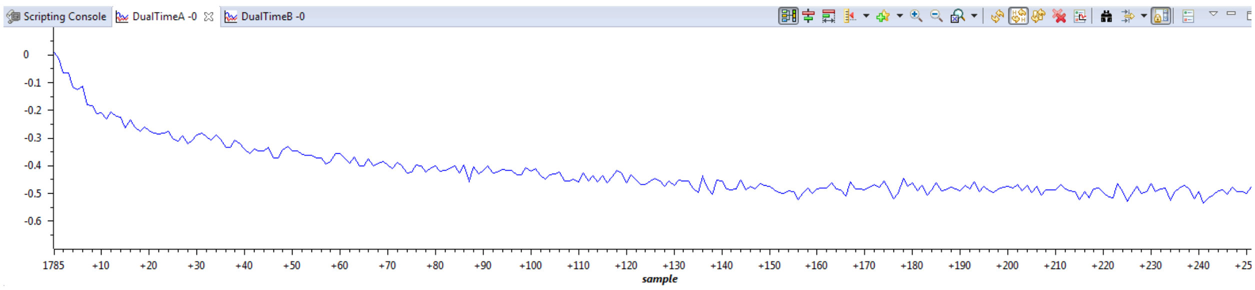


Figure 39: Current step response showing no over shoot with medium bandwidth with low Kp(0.7), low Ki(0.025)

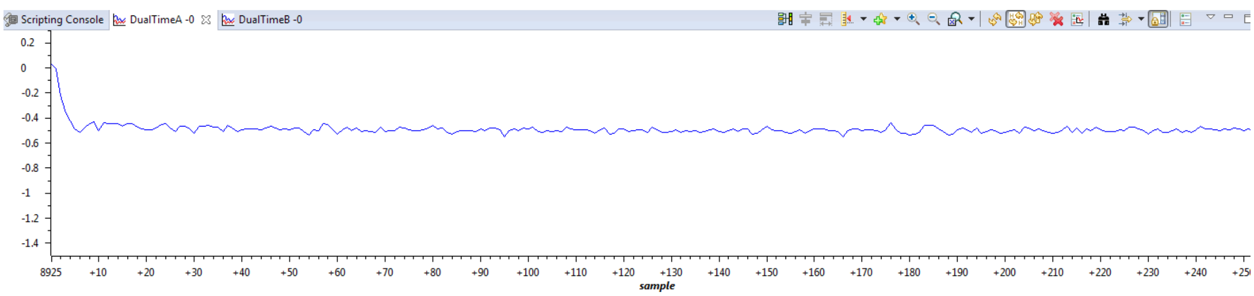


Figure 40: Current step response showing no over shoot at high bandwidth with Kp(3.5) and Ki(0.035)

It has been shown how to generate the step responses.

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

Step response generation of PI speed controller

To change the configuration of measuring the PI controller from current step response to a speed step response change the variable

- "gGraphVars.Buffer_mode" = 2.0

To create a step speed response set

- "gMotorVars.MaxAccel" = 100.0

The absolute max value to be used is 127.0 above this value the software will enter an overflow condition.

If a better resolution of the step response is needed this can be done by changing the value

- "gGraphVars.Buffer_tick"

For generating the step responses seen below a value of 25 was chosen. This means that every 25 interrupts a value is written into the data array.

Insure the current control "gMotorVars.Kp_Idq" and "gMotorVars.Ki_Idq" is set to the parameters you chose while finding the user optimum in the current step response section.

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

- "gStepVars.spdRef_Default" = 0.5 // Default starting value of spdRef
- "gStepVars.spdRef_Stepsize" = 0.5 // Step size of spdRef

Depending on your motor max speed, define a step from around 1/8 to 2/8 of the max speed of the motor with the given Vbus voltage. For the Anaheim_BLY172S motor this is approximately a "gMotorVars.SpeedRef_krpm" command from 0.5 to 1.0. When the step response is executed and the date has been logged the value of the "gMotorVars.SpeedRef_krpm" returns back to the default value "gStepVars.spdRef_Value[0]".

As done with the current step response generation, set the following variable to:

- "gStepVars.StepResponse" = 1

Now refresh the graph window, to see the speed step response.

To change the step response of the speed controller the following variables from the gMotorVars are used.

- "gMotorVars.*Kp_spd"*
- "gMotorVars.*Ki_spd"*

The graph seen in Figure 41 is a typical example using default values with the changed current controller values.

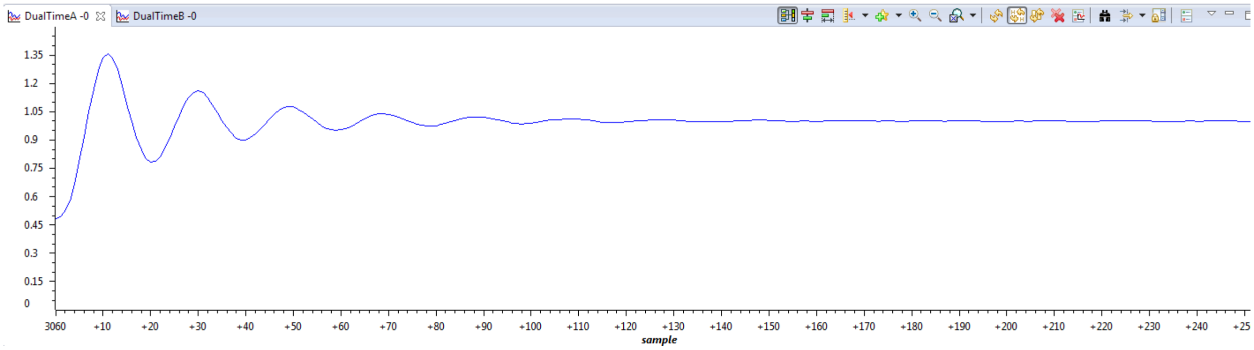


Figure 41: Anaheim_BLY172S Motor Speed step response from 500 rpm to 1k rpm with automatic Kp/Ki values

The above response was done with the values:

- "gMotorVars.*Kp_spd*" = 3.333333254
- "gMotorVars.*Ki_spd*" = 0.0222222209

As these are defined using the motor parameters these values can vary a little compared to the ones of your setup.

Similar to the current step response, changing either of Kp or Ki. Increase the value of Kp to get to the target Speed sooner, but increasing it too much results in overshoot. Reducing it too much results in a very slow (soft) response. When the value of Ki is too high it "rings", taking longer to settle.

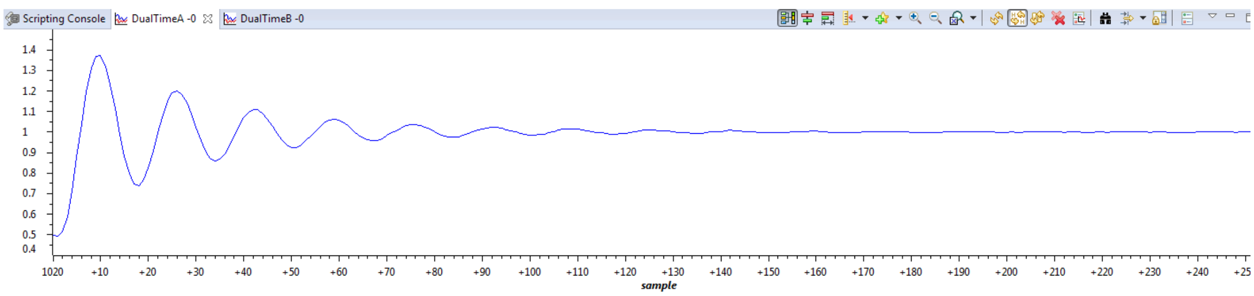


Figure 42: Speed step response showing overshoot with slight instability with too high Kp(4.0) and too high Ki(0.04)

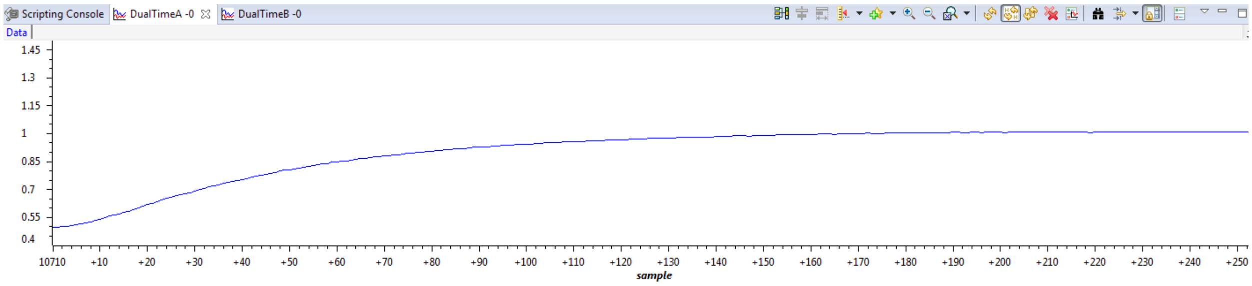


Figure 43: Slow speed step response showing no over shoot with medium bandwidth with high Kp(0.09), middle Ki(0.0002)

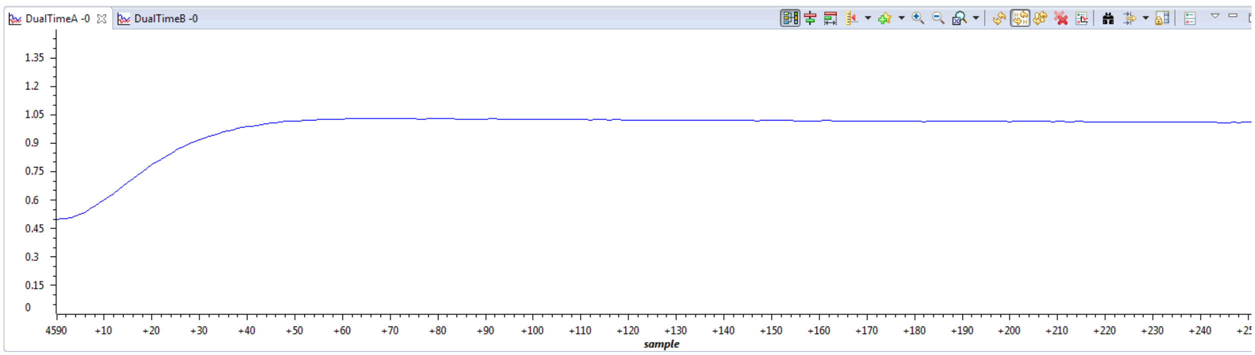


Figure 44: Speed step response showing no overshoot at high bandwidth with Kp(0.25) and Ki(0.001)

This shows different choices of step responses for the Anaheim_BLY172S motor depending on motor application either of those step responses can be used for the system.

After this it is important to also validate the controller with the normal trajectory, this has been done with the step response from

For this example normal trajectory is “gMotorVars.*MaxAccel_krpmps”* to 0.5, here it is also needed to change the value “gGraphVars.*Buffer_tick”* a good choice would be 250.

For the slow speed controller in, the following ramp behaviour was seen.

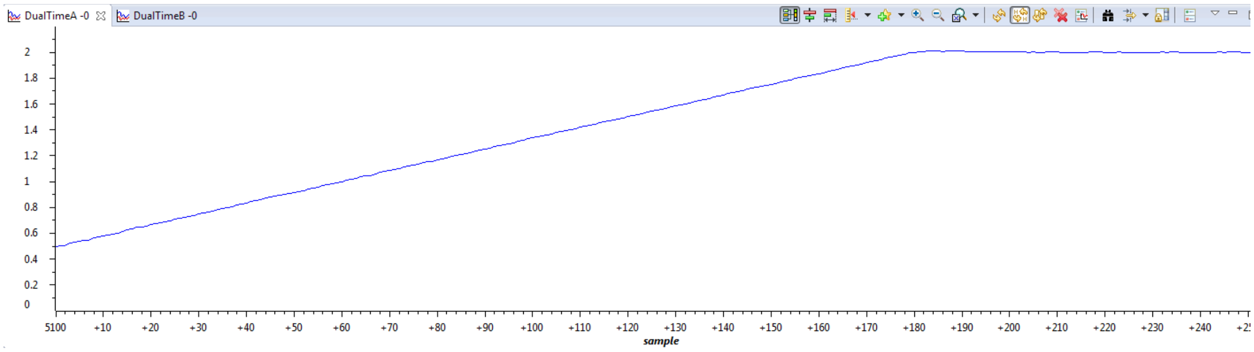


Figure 45: Anaheim_BLY172S speed step response from 500 rpm to 2k rpm with "gMotorVars.MaxAccel_krpmps" equal 0.5

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

Conclusion

Using this project and its options it is possible to generate step responses for the current and the speed controller. With these step responses it is possible to configure the PI speed and current controller to fit the customer system requirements.

This adjusting of the PI controller can improve the overall system performance, efficiency and reliability, ensuring that the system is not regulating the current and speed too fast or too slow.
For more information on the theory of control loop considerations, it is highly recommended to either see the InstaSPIN user guide (spruhj1) or read the book "Control Theory Fundamentals" ISBN-13 978-1496040732.

Lab 6a - Smooth system movement with SpinTAC Move

Abstract

InstaSPIN-MOTION includes SpinTAC Move, a motion profile generator that will generate constraint-based, time-optimal motion trajectory curves. It removes the need for look-up tables and runs in real-time to generate the desired motion profile. This lab will demonstrate the different configurations and their impact on the final speed change of the motor. Additional information about the SpinTAC™ Move can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section Trajectory Planning).

Introduction

SpinTAC™ Move is a velocity profile generator that computes the time-optimal curve within the user defined acceleration and jerk bounds. It supports basic ramp profiles as well as advanced s-curve and st-curve (Linestream Proprietary) curves. The proprietary st-curve features a continuous jerk to provide additional smoothing on the trajectory.

Prerequisites

This lab assumes that the motor inertia has been identified, that you are able to control the speed of the motor through your MotorWare project, and that the SpinTAC Velocity Control has been tuned.

Objectives Learned

- Use SpinTAC™ Move to transition between speeds.
- Become familiar with the bounds that can be adjusted as part of SpinTAC™ Move
- Continue exploring how the SpinTAC Velocity Control takes advantage of the advanced features of SpinTAC™ Move.

Background

This lab adds new API function calls to call SpinTAC™ Move. Figure 46 shows how SpinTAC™ Move connects with the rest of the SpinTAC™ components.

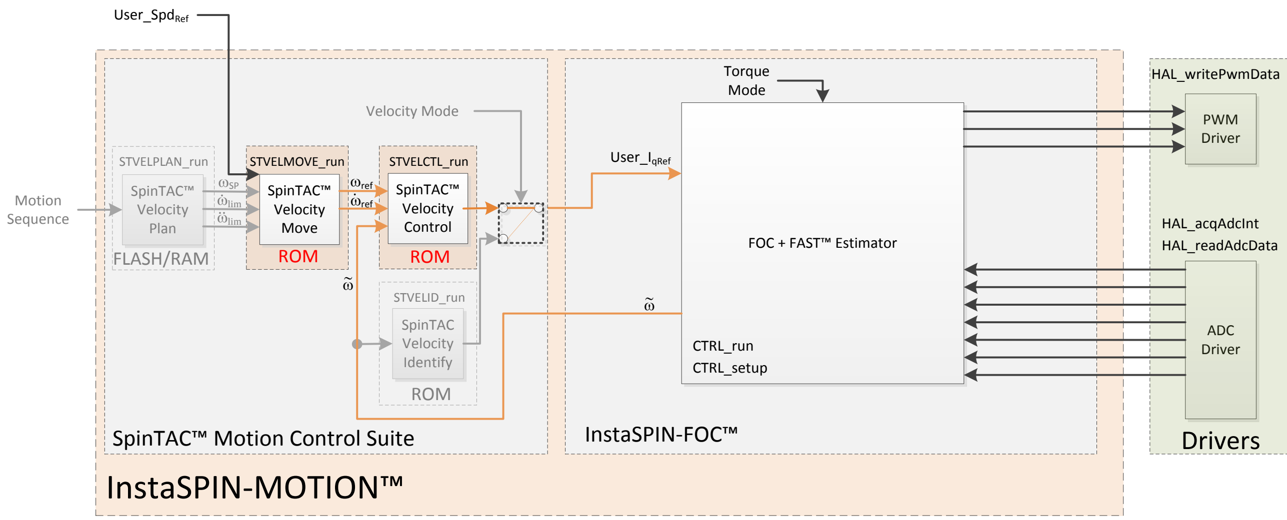


Figure 46: InstaSPIN-MOTION block diagram for lab 06a

This lab adds SpinTAC Velocity Move to provide trajectory, or speed transition, curves to the SpinTAC Velocity Control. The block diagram shows the connections between the two components. SpinTAC Velocity Move accepts the user speed reference and outputs the speed & accelerations references to SpinTAC Velocity Control.

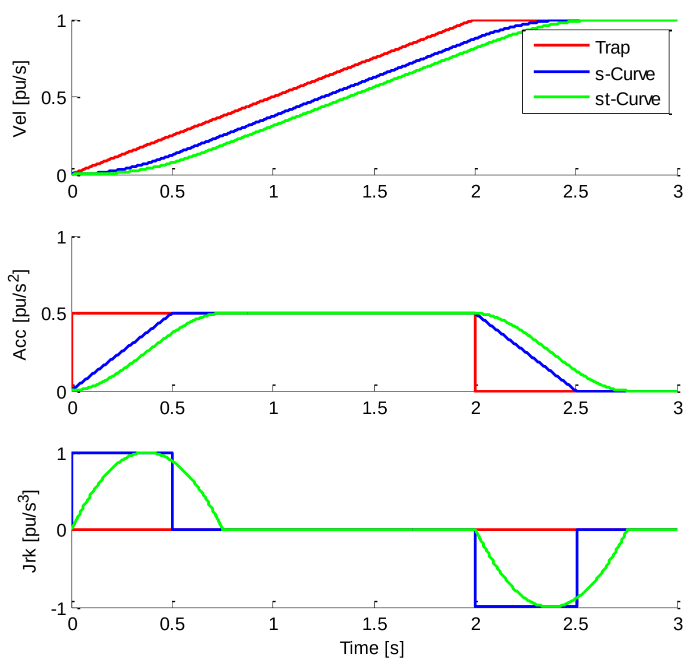


Figure 47: Trajectory Curves Available from SpinTAC™ Move

Figure 47 illustrates the differences between the three curve types available in SpinTAC™ Move. The st-curve represents the smoothest motion, which is critical for systems that are sensitive to large amounts of jerk. Jerk represents the rate of change of acceleration. A larger jerk will increase the acceleration at a faster rate. Steps, or sharp movement between two speeds, can cause systems to oscillate. The bigger the step in speed, the greater this tendency for the system to oscillate. Control over jerk can round the velocity corners, reducing oscillation. As a result, acceleration can be set higher. Controlling the jerk in your system will lead to less mechanical stress on your system components and can lead to better reliability and less failing parts.

Signals	Trapezoidal	S-Curve	st-Curve
Velocity	Continuous	Smooth	Smooth
Acceleration	Bounded	Continuous	Smooth
Jerk	N/A	Bounded	Continuous

Figure 48: Chart describing curve characteristics

Figure 48 shows the different characteristics of the three curve types provided by SpinTAC Velocity Move. St-Curve provides the smoothest motion by smoothing out the acceleration of the profile. For most applications the st-Curve represents the best motion profile.

Project Files

There are no new project files.

Include the Header File

The critical header file for the SpinTAC components is spintac_velocity. This header file is common across all labs so there will be more includes in spintac_velocity than are needed for this lab.

Table 36: Important header files needed for SpinTAC components

spintac.h	Header file containing all SpinTAC header files used in main.c	
	SpinTAC	
	spintac_vel_move.h	SpinTAC Velocity Move structures and function declarations.

Define the Global Structure

There are no new global object and variable declarations.

Initialization the Configuration Variables

The new functions that are added to this lab to setup the SpinTAC components are listed in Table 37.

Table 37: Important setup functions needed for SpinTAC

setup		
	SpinTAC	
	ST_setupVelMove	Sets up the SpinTAC Velocity Move object with default values.

Main Run-Time loop (forever loop)

Nothing has changed in the forever loop from the previous lab.

Main ISR

The main ISR calls very critical, time dependent functions that run the SpinTAC components. The new functions that are required for this lab are listed in Table 38.

Table 38: InstaSPIN functions used in the main ISR

mainISR		
	SpinTAC	
	ST_runVelMove	The ST_runVelMove function calls the SpinTAC Velocity Move object. This also handles enabling the SpinTAC Move object.

Call SpinTAC™ Velocity Move

The ST_runVelMove function has been added to the project to call the SpinTAC™ Move component and to use it to generate references for the SpinTAC™ speed controller. The functions called in ST_runVelMove are listed in Table 39.

Table 39: InstaSPIN functions used in ST_runVelMove

ST_runVelMove		
	SpinTAC	
	STVELMOVE_getReset	This function sets the reset (RES) bit in SpinTAC Velocity Move
	STVELMOVE_getVelocityEnd	This function returns the velocity setpoint (VelEnd) in SpinTAC Velocity Move
	STVELMOVE_setCurveType	This function sets the curve type (CurveType) in SpinTAC Velocity Move
	STVELMOVE_setVelocityEnd	This function sets the velocity setpoint (VelEnd) in SpinTAC Velocity Move
	STVELMOVE_setAccelerationLimit	This function sets the acceleration limit (AccLim) in SpinTAC Velocity Move
	STVELMOVE_setJerkLimit	This function sets the jerk limit (JrkLim) in SpinTAC Velocity Move
	STVELMOVE_setEnable	This function sets the enable (ENB) bit in SpinTAC Velocity Move
	STVELMOVE_getVelocityStart	This function gets the velocity start (VelStart) of SpinTAC Velocity Move
	STVELMOVE_run	This function calls into the SpinTAC Move to generate motion profiles.
	EST	
	EST_setFlag_enableForceAngle	This function sets the flag to enable Force Angle in the Fast Estimator

Call SpinTAC™ Velocity Controller

The ST_runVelCtl function has been updated to use SpinTAC Velocity Move as the speed reference instead of the ramp generator.

Table 40: InstaSPIN functions used in ST_runVelCtl

ST_runVelCtl		
	SpinTAC	
	STVELMOVE_getVelocityReference	This function returns the velocity reference (VelRef) from SpinTAC Velocity Move
	STVELMOVE_getAccelerationReference	This function returns the acceleration reference (AccRef) from SpinTAC Velocity Move

Lab Procedure

In Code Composer, build lab6a, connect to the target and load the .out file.

- Open the command file “sw\solutions\instaspin_motion\src\proj_lab06a.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

To run the motor a couple of steps are required:

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To start the current loop controller, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

SpinTAC Move will generate motion profiles every time the speed reference is updated.

- Set “gMotorVars.SpeedRef_krpm” to 1.0 to get the motor spinning at 1000 rpm

The motor will accelerate up to the goal speed at the slow default acceleration. The acceleration can be set much faster.

- “gMotorVars.MaxAccel_krpmps” configures the acceleration used in the profile. Set this value to 2.0. This will cause the motor to accelerate much quicker.
- Change the speed reference via “gMotorVars.SpeedRef_krpm” to -1.0 to see how changing the acceleration limit allows the motor to change speeds much more quickly.
- Set “gMotorVars.MaxAccel_krpmps” to 120.0. This is the maximum acceleration for this project. This acceleration maximum is based on the maximum value of an IQ24 variable. This value is set to be the maximum acceleration in SpinTAC™ Move. If a larger acceleration is configured, this will cause SpinTAC™ Move to report an error via “gMotorVars.SpinTAC.VelMoveErrorID.”
- Set “gMotorVars.SpeedRef_krpm” to 1.0 to see the very fast acceleration

SpinTAC™ Move provides a feedforward reference that is used by the SpinTAC Velocity Control to enable very accurate profile tracking. This feedforward reference is the acceleration reference and the rate at which it changes is the jerk. The jerk sets the rate of change of the acceleration.

- To see the impact of jerk set “gMotorVars.MaxJrk_krpmps2” to 750.0. This is the maximum value for jerk in this project. This jerk maximum is based on the maximum value of an IQ20 variable. This value is set to be the maximum jerk in SpinTAC™ Move. If a larger jerk is configured, this will cause SpinTAC™ Move to report an error via “gMotorVars.SpinTAC.VelMoveErrorID.”
- Set “gMotorVars.SpeedRef_krpm” to -1.0 to see how adjusting the jerk allows the motor to accelerate even faster.

SpinTAC™ Move supports three different curve types: trapezoid, s-curve, and st-curve. The curve can be selected by changing “gMotorVars.VelMoveCurveType.” The differences between the three curves types are discussed in detail in the InstaSPIN-MOTION User’s Guide. Note that using an extremely fast acceleration and jerk with a trapezoid curve can cause the motor to brown-out the processor. This limitation is due to the power supply on the TI evaluation board. If that happens, stop the debugger and relaunch the lab.

SpinTAC™ Move will alert the user when it has completed a profile via the done bit. When the profile is completed, “gMotorVars.SpinTAC.VelMoveDone” will be set to True. This could be used in a project to alert the system when the motor reaches goal speed.

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how easy it is to use SpinTAC Move to generate constraint-based, time-optimal motion profiles. This lab also shows the different curves that can be used with SpinTAC Move. The st-curve provides a continuous jerk profile that will enable very smooth motion for jerk sensitive applications.

Lab 6b - Motion Sequence Example

Abstract

InstaSPIN-MOTION includes SpinTAC™ Velocity Plan, a motion sequence planner that allows you to easily build complex motion sequences. This will allow you to quickly implement your application's motion sequence and speed up development time. This lab provides a very simple example of a motion sequence. Additional information about trajectory planning, motion sequences, and SpinTAC™ Velocity Plan can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section Trajectory Planning).

Introduction

SpinTAC™ Velocity Plan implements motion sequence planning. It allows for you to quickly build a motion sequence to run your application. SpinTAC™ Velocity Plan features: conditional transitions, variables, state timers, and actions. This lab will use a simple example to show how to quickly implement your application's motion sequence.

Objectives Learned

- Use SpinTAC™ Velocity Plan to design a motion sequence.
- Use SpinTAC™ Velocity Plan to run a motion sequence.
- Understand the features of SpinTAC™ Velocity Plan

Background

Lab 6b adds new API function calls for SpinTAC™ Velocity Plan. Figure 49 shows how SpinTAC™ Velocity Plan connects with the rest of the SpinTAC™ components.

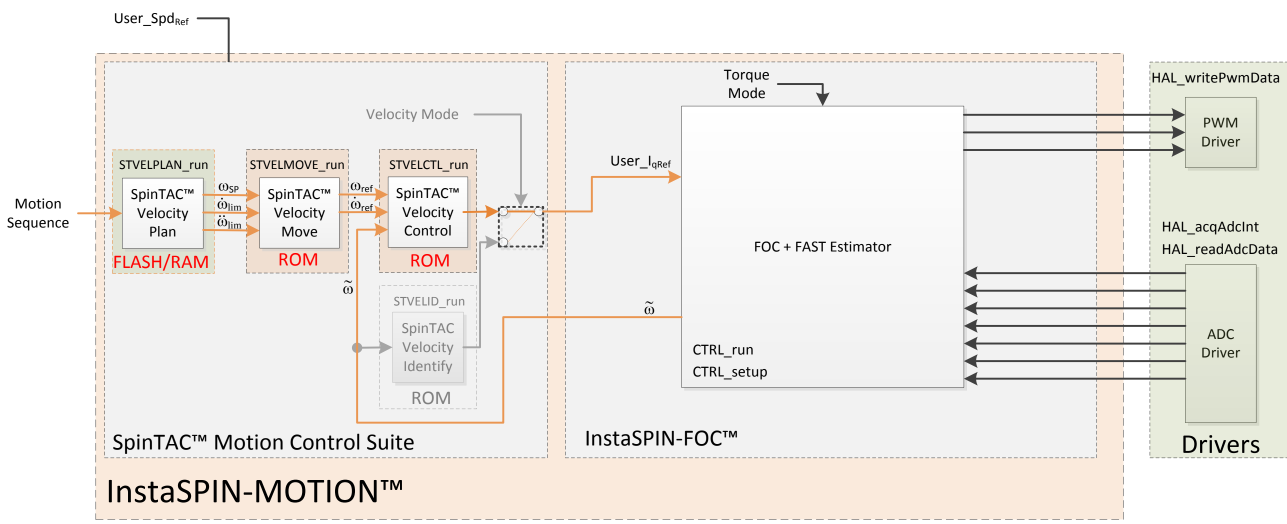


Figure 49: InstaSPIN-MOTION block diagram for lab 06b

This lab adds the SpinTAC Velocity Plan into the project. This block diagram shows how to integrate the SpinTAC Velocity Plan with the rest of the SpinTAC components. SpinTAC Velocity Plan accepts a motion sequence as an input and outputs the speed set point, acceleration limit, and jerk limit. These get passed into SpinTAC Velocity Move which takes these limits and generates a profile to provide to SpinTAC Velocity Control.

T This lab implements an example of a simple motion sequence. It includes the basic features provided by SpinTAC Velocity Plan. Figure 50 shows the state transition map for the example motion sequence.

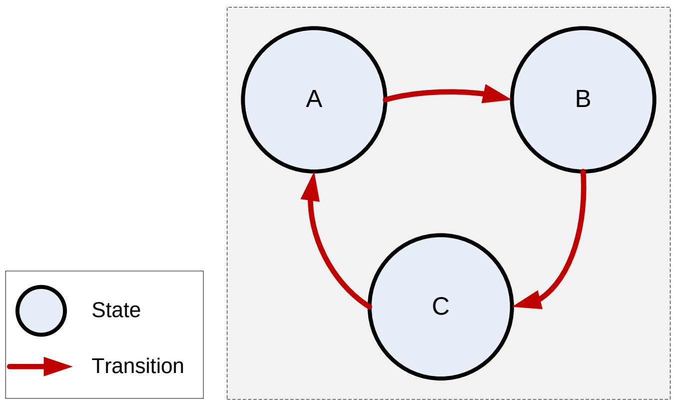


Figure 50: State transition map of Example Motion Sequence

This motion sequence transitions from state A to state B and onto state C as soon as the state timer has elapsed.

SpinTAC Velocity Plan only consumes the amount of memory that is required to configure your motion sequence. A simple motion sequence, like the one in this lab, will consume less memory than a more complicated motion sequence, like in the next lab. It is important that the allocated configuration array is correctly sized for your motion sequence. This topic is further covered in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section SpinTAC Plan Element Limits).

Additional details around the operation and configuration of SpinTAC™ Velocity Plan are found in the InstaSPIN-MOTION User's Guide.

Project Files

There are no new project files.

Include the Header File

The critical header file for the SpinTAC components is spintac_velocity. This header file is common across all labs so there will be more includes in spintac_velocity than are needed for this lab.

Table 41: Important header files needed for SpinTAC components

spintac.h	Header file containing all SpinTAC header files used in main.c	
	SpinTAC	
	spintac_vel_plan.h	SpinTAC Velocity Plan structures and function declarations.

Declare the Global Structure

Lab 06b adds global variables to monitor the internal states of SpinTAC™ Velocity Plan and to control SpinTAC™ Velocity Plan. These variables provide an external interface to start, stop, and pause SpinTAC™ Velocity Plan. These variables are also used to store the configuration of SpinTAC Velocity Plan. This array needs to be declared in the user code so that it can be sized to fit the requirements of the motion sequence defined by the user.

Table 42: Global object and variable declarations for SpinTAC™ Velocity Plan

globals		
	SpinTAC Plan	
	ST_PlanButton_e	Used to handle controlling SpinTAC Plan. This defines the different states that can be set to control the operation.
	ST_VELPLAN_CFG_ARRAY_DWORDS	The MARCO define used to establish the size of the SpinTAC Plan configuration array. This value is calcualted based on the number of elements that will be used in the SpinTAC Plan configuration
	stVelPlanCfgArray	This array is used to store the SpinTAC Plan configuration.

Initialize the Configuration Variables

During the initialization and setup, the project will call the ST_setupVelPlan function to configure and load the motion sequence into SpinTAC™ Velocity Plan. This function is declared in the main source file for this project. A detailed explanation of the API calls in ST_setupVelPlan can be found in the InstaSPIN-MOTION User's Guide.

Table 43: InstaSPIN functions used in Initialization and Setup

Setup		
	SpinTAC	
	ST_setupVelPlan	This function calls into SpinTAC Plan to configure the motion sequence.

Configuring SpinTAC™ Velocity Plan

When the motion sequence in SpinTAC Velocity Plan is configured there are many different elements that build the motion sequence. The elements covered in this lab are States and Transitions. Each of these elements has a different configuration function. It is important that the configuration of SpinTAC Velocity Plan is done in this order. If the configuration is not done in this order it could cause a configuration error.

States

STVELPLAN_addCfgState(Velocity Plan Handle, Speed Setpoint [pu/s], Time in State [ISR ticks])
This function adds a state into the motion sequence. It is configured by setting the speed that you want the motor to run during this state and with the minimum time it should remain in this state.

Transition

STVELPLAN_addCfgTran(Velocity Plan Handle, From State, To State, Condition Option, Condition Index 1, Condition Index 2, Acceleration Limit [pu/s^2], Jerk Limit [pu/s^3])
This function establishes the transitions between two states. The From State and To State values describe which states this transition is valid for. The condition option specifies if a condition needs to be evaluated prior to the transition. The condition index 1 & 2 specify which conditions should be evaluated. If less than two conditions need to be evaluated, set the unused values to 0. Acceleration limit sets the acceleration to use to transition between the From State speed and the To State speed. This value cannot exceed the acceleration max that is configured for the motion sequence. The jerk limit sets the jerk to be used in the speed transition. This value should not exceed the jerk max that is configured for the motion sequence.

The functions used in ST_setupVelPlan are described in Table 44.

Table 44: InstaSPIN functions used in ST_setupVelPlan

ST_setupVelPlan		
	SpinTAC	
	STVELPLAN_setCfgArray	This function provides the configuration array information into SpinTAC Velocity Plan
	STVELPLAN_setCfg	This function provides the system maximum information to SpinTAC Velocity Plan
	STVELPLAN_setCfgHaltState	This function provides the system halt information to SpinTAC Velocity Plan
	STVELPLAN_addCfgState	This function adds a State into the SpinTAC Velocity Plan configuration
	STVELPLAN_addCfgTran	This function adds a Transition into the SpinTAC Velocity Plan configuration
	STVELPLAN_getErrorID	This function returns the error (ERR_ID) of SpinTAC Velocity Plan.

Main Run-Time loop (forever loop)

Nothing has changed in the forever loop from the previous lab.

Main ISR

The main ISR calls very critical, time dependent functions that run the SpinTAC components. The new functions that are required for this lab are listed in Table 45.

Table 45: InstaSPIN functions used in the main ISR

mainISR		
	SpinTAC	
	ST_runVelPlan	The ST_runVelPlan function calls the SpinTAC Velocity Plan object. This also handles enabling the SpinTAC Plan object.
	ST_runVelPlanIsr	The ST_runVelPlanIsr function calls the time-critical parts of the SpinTAC Velocity Plan object.

Call SpinTAC™ Velocity Plan

The ST_runVelPlan function has been added to the project to call the SpinTAC™ Velocity Plan component and to use it to generate motion sequences. Table 46 lists the InstaSPIN functions called in ST_runVelPlan.

Table 46: InstaSPIN functions used in ST_runVelPlan

ST_runVelPlan		
	EST	
	EST_getFm_pu	This function returns the speed feedback in pu from the FAST Estimator
	SpinTAC	
	STVELPLAN_getErrorID	This function returns the error (ERR_ID) in SpinTAC Velocity Plan.
	STVELPLAN_setEnable	This function sets the enable (ENB) bit in SpinTAC Velocity Plan.
	STVELPLAN_setReset	This function sets the reset (RES) bit in SpinTAC Velocity Plan.
	STVELPLAN_run	This function calls into SpinTAC Velocity Plan to run the motion sequence.
	STVELPLAN_getStatus	This function returns the status (STATUS) of SpinTAC Velocity Plan.
	STVELPLAN_getCurrentState	This function returns the current state (CurState) of SpinTAC Velocity Plan.
	STVELPLAN_getVelocitySetpoint	This function returns the velocity setpoint (VelEnd) produced by SpinTAC Velocity Plan.
	STVELPLAN_getAccelerationLimit	This function returns the acceleration limit (AccLim) produced by SpinTAC Velocity Plan.
	STVELPLAN_getJerkLimit	This function returns the jerk limit (JrkLim) produced by SpinTAC Velocity Plan.

The ST_ runVelPlanTick function has been added to the project to call the time-critical components of SpinTAC™ Velocity Plan. Table 47 lists the InstaSPIN functions called in ST_ runVelPlanTick.

Table 47: InstaSPIN functions used in ST_ runVelPlanTick

ST_runVelPlanIsr		
	SpinTAC	
	STVELPLAN_runTick	This function calls into SpinTAC Plan to update the timer value in SpinTAC Plan.
	STVELPLAN_setUnitProfDone	This function indicates to SpinTAC Plan that SpinTAC Move has completed running the requested profile.

Lab Procedure

In Code Composer, build lab6b, connect to the target and load the .out file.

- Open the command file “sw\solutions\instaspin_motion\src\proj_lab06b.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

To run the motor a couple of steps are required:

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To start the current loop controller, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

To start the motion sequence, the SpinTAC™ Velocity Plan button needs to be set to start once the FAST estimator is online.

- Set “gMotorVars.SpinTAC.VelPlanRun” to ST_PLAN_START to begin the motion sequence.

The motor will run through a very simple motion sequence where the motor spins anti-clockwise and then spins clockwise.

- At the conclusion of the motion sequence “gMotorVars.SpinTAC.VelPlanDone” will be set to True. This is done to indicate to the user program that the motion sequence has completed.

Now modify the SpinTAC™ Velocity Plan configuration to transition from State C to State B instead of State A..

- In the function ST_setupVelPlan, find the line highlighted in Figure 51. Change the value from STATE_A to STATE_B.

```
//Example: STVELPLAN_addCfgTran(handle,    FromState, ToState, CondOption, CondIdx1, CondiIdx2, AccLim[pups2], JrkLim[pups3]);
STVELPLAN_addCfgTran(stObj->velPlanHandle, STATE_A,   STATE_B, ST_COND_NC, 0,        0,         _IQ(0.1),      _IQ20(1)); // From StateA
STVELPLAN_addCfgTran(stObj->velPlanHandle, STATE_B,   STATE_C, ST_COND_NC, 0,        0,         _IQ(0.1),      _IQ20(1)); // From StateB
STVELPLAN_addCfgTran(stObj->velPlanHandle, STATE_C,   STATE_A, ST_COND_NC, 0,        0,         _IQ(1),        _IQ20(1)); // From StateC
```

Figure 51: Code modification to change state transition

- Recompile and download the .out file

Now the motor will not stop transitioning from anti-clockwise to clockwise until “gMotorVars.SpinTAC.VelPlanRun” is set to ST_PLAN_STOP.

Continue to explore the advanced features of SpinTAC™ Velocity Plan by making additional modifications to the motions sequence. Some examples are provided below.

- Add a State D to SpinTAC Velocity Plan
- Add a transition to and from State D
- Change the transitions to run the state machine from state C -> B -> A

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how easy it is to design motion sequences using SpinTAC™ Velocity Plan. This lab configures SpinTAC™ Velocity Plan to run a simple motion sequence. This lab also showcases how easy it is to modify the motion sequence and introduces the API calls that make up the SpinTAC™ Velocity Plan configuration.

Lab 6c - Motion Sequence Real World Example: Washing Machine

Abstract

SpinTAC™ Velocity Plan is a motion sequence planner. It allows you to easily build complex motion sequences. This will allow you to quickly implement your application's motion sequence and speed up development time. This lab provides a very complex example of a motion sequence. Additional information about trajectory planning, motion sequences, and SpinTAC™ Velocity Plan can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section 14 Trajectory Planning).

Introduction

SpinTAC™ Velocity Plan implements motion sequence planning. It allows for you to quickly build a motion sequence to run your application. SpinTAC™ Velocity Plan features: conditional transitions, variables, state timers, and actions. This lab will use the example of a washing machine to show how all of these features can be used to implement your application's motion sequence.

Objectives Learned

- Use SpinTAC™ Velocity Plan to design a complicated motion sequence.
- Use SpinTAC™ Velocity Plan to run a complicated motion sequence.
- Understand the features of SpinTAC™ Velocity Plan

Background

This lab adds new API function calls for SpinTAC™ Velocity Plan. Figure 52 shows how SpinTAC™ Velocity Plan connects with the rest of the SpinTAC™ components.

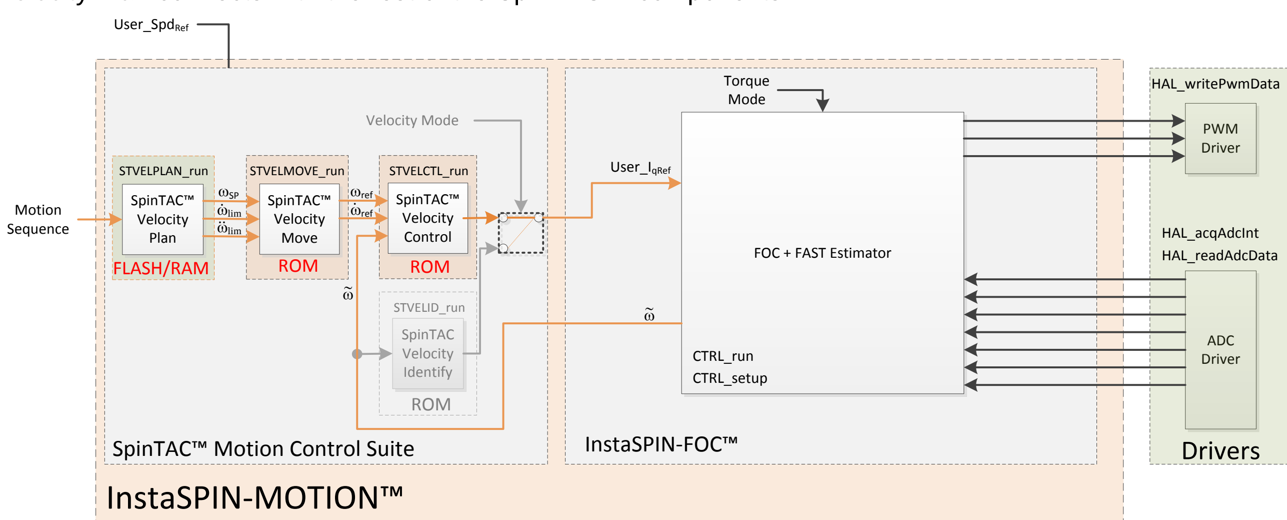


Figure 52: InstaSPIN-MOTION block diagram for lab 06c

This lab does not contain any changes from the block diagram perspective. The primary change is using a different motion sequence.

The washing machine example provided in this lab is an example of a complex motion sequence. It features many interfaces to sensors and valves as well as conditional state transitions. The entire motion sequence can be implemented in SpinTAC™ Velocity Plan. Figure 53 shows the state transition map for the washing machine.

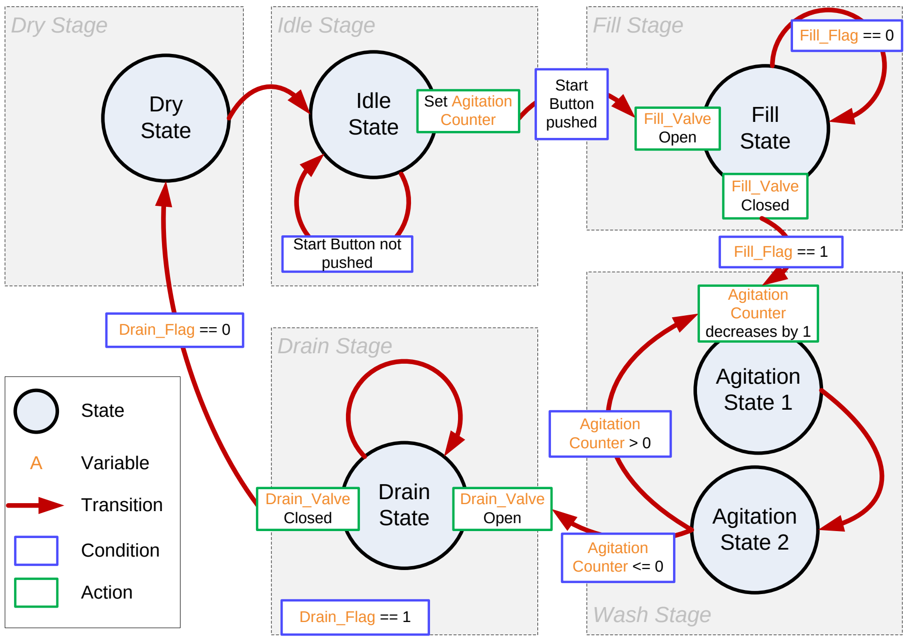


Figure 53: State Transition Map of Washing Machine Example

The washing machine example contains the basic operations of a washing machine. It contains five stages: idle stage, fill stage, wash stage, drain stage, and dry stage.
The washing machine stays in idle state until the start button is pushed. Once the start button is pressed, it will enter the fill stage and the agitation counter is set to the configured value.
Upon entering the fill stage, the water fill valve is open. A water level sensor is used to indicate when the tub is full of water. When the water is filled, the water fill valve is closed and the system goes into the wash stage.
In the wash stage, the motor agitates between a positive speed and a negative speed until the agitation counter reaches 0. Then it goes into drain stage.
When entering the drain stage, the drain valve is opened. A drain sensor is used to indicate when the water is drained. When the water is finished draining, the drain valve is closed, and the system enters the dry stage.
In dry stage, the motor spins at a certain speed for a configured time. Once the time elapses, it will enter idle stage. At this point the operation is finished.

Additional information about trajectory planning, motion sequences and SpinTAC™ Velocity Plan can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section Trajectory Planning).

Project Files

There are no new project files.

Include the Header File

There are not new includes.

Declare the Global Structure

This lab adds global variables to monitor the internal states of SpinTAC™ Velocity Plan and to control SpinTAC™ Velocity Plan. These variables provide an external interface to start, stop, and pause SpinTAC™ Velocity Plan. These new variables are covered in

Table 48: Global object and variable declarations in SpinTAC Velocity Plan

globals		
	SpinTAC Plan	
	gVelPlanVar	This array contains the states of the internal SpinTAC Plan variables.
	Washer	
	WASHER_State_e	Defines the states that make up the washing machine motion sequence.
	gWaterLevel	Holds the value for the water level in the washing machine drum. 10000 represents a full water level.

Initialize the Configuration Variables

There are no new function calls used to setup SpinTAC Velocity Plan. However the contents of ST_setupVelPlan have been modified to run the example washing machine motion sequence.

Configuring SpinTAC™ Velocity Plan

When the motion sequence in SpinTAC Velocity Plan is configured there are many different elements that build the motion sequence. These elements are States, Variables, Conditions, Transitions, and Actions. Each of these elements has a different configuration function. It is important that the configuration of SpinTAC Velocity Plan is done in this order. If the configuration is not done in this order it could cause a configuration error.

States

STVELPLAN_addCfgState(Velocity Plan Handle, Speed Setpoint [pu/s], Time in State [ISR ticks])
This function adds a state into the motion sequence. It is configured by setting the speed that you want the motor to run during this state and with the minimum time it should remain in this state.

Variables

STVELPLAN_addCfgVar(Velocity Plan Handle, Variable Type, Initial Value)
This function establishes a variable that will be used in the motion sequence. The variable type determines how SpinTAC™ Velocity Plan can use this variable. The initial value is the value that should be loaded into this variable initially. The variable can be up to a 32-bit value.

Conditions

STVELPLAN_addCfgCond(Velocity Plan Handle, Variable Index, Comparison, Comparison Value 1, Comparison Value 2)
This function sets up a condition to be used in the motion sequence. This will be a fixed comparison of a variable against a value or value range. The variable index describes which variable should be compared.

The comparison should be used to describe the type of comparison to be done. Comparison values 1 & 2 are used to establish the bounds of the comparison. If a comparison only requires one value it should be set in comparison value 1 and comparison value 2 should be set to 0.

Transition

STVELPLAN_addCfgTran(Velocity Plan Handle, From State, To State, Condition Option, Condition Index 1, Condition Index 2, Acceleration Limit [pu/s^2], Jerk Limit [pu/s^3])
This function establishes the transitions between two states. The From State and To State values describe which states this transition is valid for. The condition option specifies if a condition needs to be evaluated prior to the transition. The condition index 1 & 2 specify which conditions should be evaluated. If no conditions or one condition needs to be evaluated, set the not used values to 0. Acceleration limit sets the acceleration to use to transition between the From State speed and the To State speed. This value cannot exceed the acceleration max that is configured for the motion sequence. The jerk limit sets the jerk to be used in the speed transition. This value should not exceed the jerk max that is configured for the motion sequence.

Actions

STVELPLAN_addCfgAct(Velocity Plan Handle, State Index, Condition Option, Condition Index 1, Condition Index 2, Variable Index, Operation, Value, Action Trigger)
This function adds an action into the motion sequence. The state index describes which state the action should take place in. The condition option specifies if a condition needs to be evaluated prior to the action. The condition index 1 & 2 specify which conditions should be evaluated. If no conditions or one condition needs to be evaluated, set the not used values to 0. The variable index indicates which variable the action should be done to. The operation determines what operation should be done to the variable, the only available options are to add a value or set a value. The value is what should be added or set to the variable. The action trigger indicates if the action should be performed when entering or exiting the state.

This function has been modified to configure SpinTAC Velocity Plan to run the motion sequence of a washing machine. There are new function calls in order to take advantage of the advanced features of SpinTAC Velocity Plan. The new functions are described in Table 49.

Table 49: InstaSPIN functions used in ST_setupVelPlan

ST_setupVelPlan		
	SpinTAC	
	STVELPLAN_addCfgVar	This function adds a Variable into the SpinTAC Velocity Plan configuration
	STVELPLAN_addCfgCond	This function adds a Condition into the SpinTAC Velocity Plan configuration
	STVELPLAN_addCfgAct	This function adds a Action into the SpinTAC Velocity Plan configuration

Main Run-Time loop (forever loop)

Nothing has changed in the forever loop from the previous lab.

Main ISR

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

Call SpinTAC™ Velocity Plan

The ST_runVelPlan function has been updated to interface with the external components that make up the sensors and valves of the simulated washing machine. Table 50 lists the functions used to interface with external components in the ST_runVelPlan function.

Table 50: InstaSPIN functions used in ST_runVelPlan

ST_runVelPlan		
	SpinTAC	
	STVELPLAN_getVar	This function returns the value of a variable in SpinTAC Velocity Plan
	STVELPLAN_setVar	This function sets the value of a variable in SpinTAC Velocity Plan

Lab Procedure

In Code Composer, build lab6c, connect to the target and load the .out file.

- Open the command file “sw\solutions\instaspin_motion\src\proj_lab06c.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

To run the motor a couple of steps are required:

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To start the current loop controller, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

To start the motion sequence, the SpinTAC™ Velocity Plan button needs to be set to start.

- Set “gMotorVars.SpinTAC.VelPlanRun” to ST_PLAN_START to begin the motion sequence once the FAST Estimator is online.

The motor will run through a procedure that is designed to emulate a washing machine. It will open the WASHER_FillValve. Water will fill the drum until “gWaterLevel” reaches 1000. Once “gWaterLevel” reaches the maximum, the WASHER_FillSensor is tripped. Then the motor will then spin clockwise and counter-clockwise 20 times. After agitation the WASHER_DrainValve will open and “gWaterLevel” will decrease. When “gWaterLevel” reaches 0 the water has been fully drained from the drum. It will then begin the spin cycle. At the conclusion of the spin cycle the washer will return to the idle state ready to begin another motion sequence.

- At the conclusion of the motion sequence “gMotorVars.SpinTAC.VelPlanDone” will be set to True. This is done to indicate to the user program that the motion sequence has completed.

Now modify the SpinTAC™ Velocity Plan configuration to do 30 agitations instead of 20.

- In the function ST_setupVelPlan, find the line highlighted in Figure 54. Change the value from 20 to 30.

```
//Example: STVELPLAN_addCfgAct(handle,    StateIdx,       CondOption, CondIdx1, CondIdx2, VarIdx,              Operation,  Value, ActionTriger);
STVELPLAN_addCfgAct(stObj->velPlanHandle, WASHER_IDLE,    ST_COND_NC, 0,        0,        WASHER_CycleCounter, ST_ACT_EQ,  20,    ST_ACT_EXIT);
STVELPLAN_addCfgAct(stObj->velPlanHandle, WASHER_AGI_CCW, ST_COND_NC, 0,        0,        WASHER_CycleCounter, ST_ACT_ADD, -1,    ST_ACT_ENTR);
STVELPLAN_addCfgAct(stObj->velPlanHandle, WASHER_FILL,    ST_COND_NC, 0,        0,        WASHER_FillValve,    ST_ACT_EQ,  1,     ST_ACT_ENTR);
STVELPLAN_addCfgAct(stObj->velPlanHandle, WASHER_FILL,    ST_COND_NC, 0,        0,        WASHER_FillValve,    ST_ACT_EQ,  0,     ST_ACT_EXIT);
STVELPLAN_addCfgAct(stObj->velPlanHandle, WASHER_DRAIN,   ST_COND_NC, 0,        0,        WASHER_DrainValve,   ST_ACT_EQ,  1,     ST_ACT_ENTR);
STVELPLAN_addCfgAct(stObj->velPlanHandle, WASHER_DRAIN,   ST_COND_NC, 0,        0,        WASHER_DrainValve,   ST_ACT_EQ,  0,     ST_ACT_EXIT);
```

Figure 54: Code modification to adjust the agitation cycle counter

- Recompile and download the .out file

Now when the washing machine goes into agitation it should do 30 agitation cycles instead of the 20 before.

Continue to explore the advanced features of SpinTAC™ Velocity Plan by making additional modifications to the motions sequence. Some examples are provided below.

- Adjust the speed achieved during agitation
- Add a second dry stage at a different speed
- Adjust the agitation cycle so that it will only exit after an external event

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how easy it is to design complex motion sequences using SpinTAC™ Velocity Plan. This lab configures SpinTAC™ Velocity Plan to run a washing machine profile that features complex elements. This lab also showcases how easy it is to modify the motion sequence and introduces the API calls that make up the SpinTAC™ Velocity Plan configuration.

Lab 6d - Designing your own Motion Sequence

Abstract

Now that SpinTAC™ Velocity Plan has been introduced, this lab lets you create your own motion sequence. It is a chance to be creative and utilize the topics and skills that were learned in the previous labs. Additional information about trajectory planning, motion sequences, and SpinTAC™ Velocity Plan can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section 14 Trajectory Planning).

Introduction

SpinTAC™ Velocity Plan implements motion sequence planning. It allows for you to quickly build a motion sequence to run your application. SpinTAC™ Velocity Plan features: conditional transitions, variables, state timers, and actions. This lab will let you use these advanced features to implement your own motion sequence.

Objectives Learned

- Understand the flexibility of SpinTAC™ Velocity Plan and how it can speed up product design
- Be creative
- Have fun

Background

This lab adds no new API function calls from the previous lab. Figure 55 shows the block diagram that this project is based on.

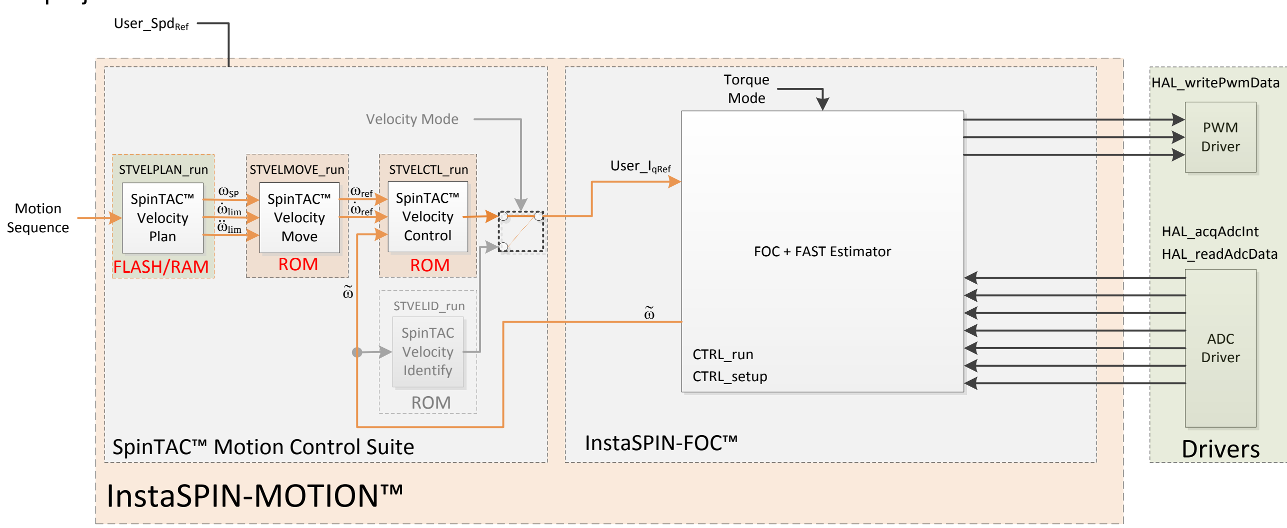


Figure 55: InstaSPIN-MOTION block diagram for lab 06d

This lab has no changes from the block diagram. The code change is in the configured motion sequence.

This lab introduces three different potential motion sequences. It features a Test Pattern, Grocery Conveyor, and Garage Door. These motion sequences can be changed at run time with the switch gSelectedPlan. The motion sequence can only be changed when SpinTAC Velocity Plan is in the IDLE state.

The Test Pattern motion sequence runs a fixed speed pattern designed to exercise the controller. This motion sequence does not contain any variables or conditional transitions. It simply runs the motor for a fixed amount of time at each speed. Figure 56 contains the state transition map for this state machine.

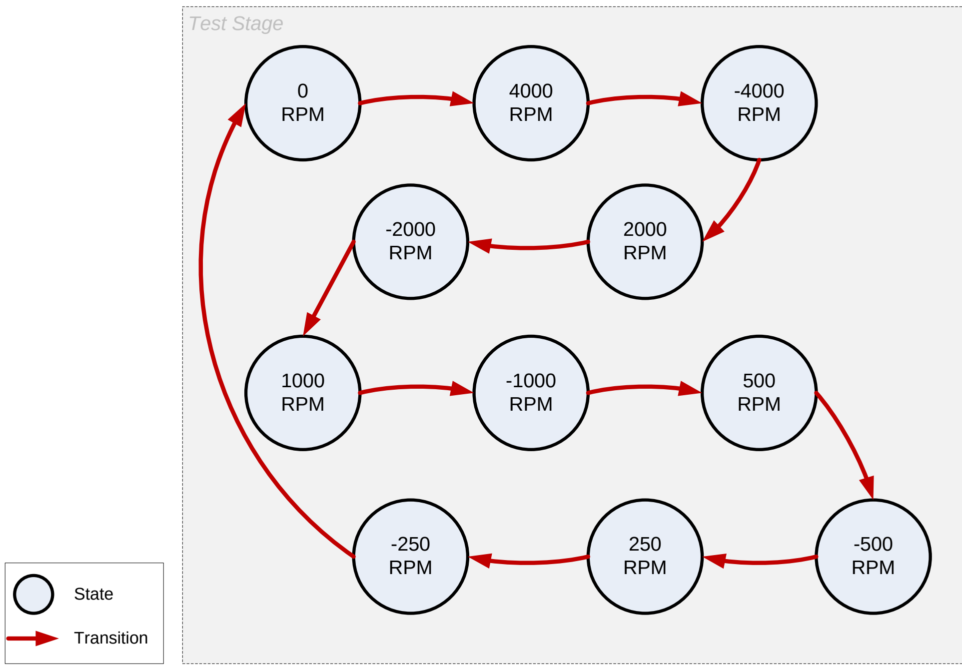


Figure 56: State transition map of Test Pattern

The Grocery Conveyor motion sequence runs a simulation of a grocery store conveyor belt. It features two variables: Switch and Proximity Sensor. The Switch variable is designed to be an On/Off switch for the conveyor belt. The Proximity Sensor is designed to stop the convey belt when it has conveyed groceries to the end of its travel. When this motion sequence is started, it will wait until the Switch is set to on and the Proximity Sensor is open. It will then spin the motor at a continuous speed until either the Switch is set to Off or the Proximity Sensor is blocked. Figure 57 contains the state transition map for this state machine.

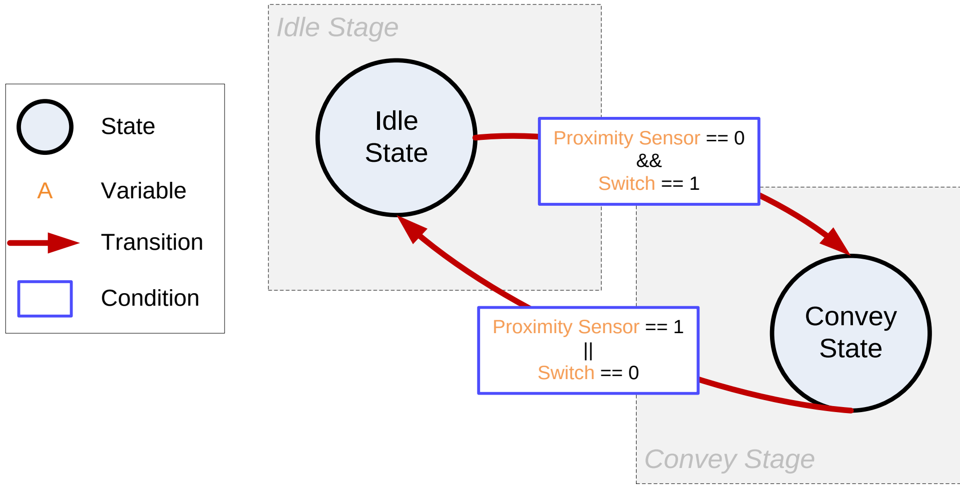


Figure 57: State transition map for Grocery Conveyor

The Garage Door motion sequence runs a simulation of a garage door opener. It features three variables: Down Sensor, Up Sensor, and Button. The Down Sensor detects that the door is in the down position. The Up Sensor detects that the door is in the up position. The Button indicates that an operation must be performed. When this motion sequence is started it waits until the Button is pressed. Depending on the state of the position sensors, the motor will either take the Garage Door up or down. When it reaches its destination, the motion sequence will return to idle. If the Button is pressed while the motor is moving it will reverse its operation. Figure 58 contains the state transition map for this state machine.

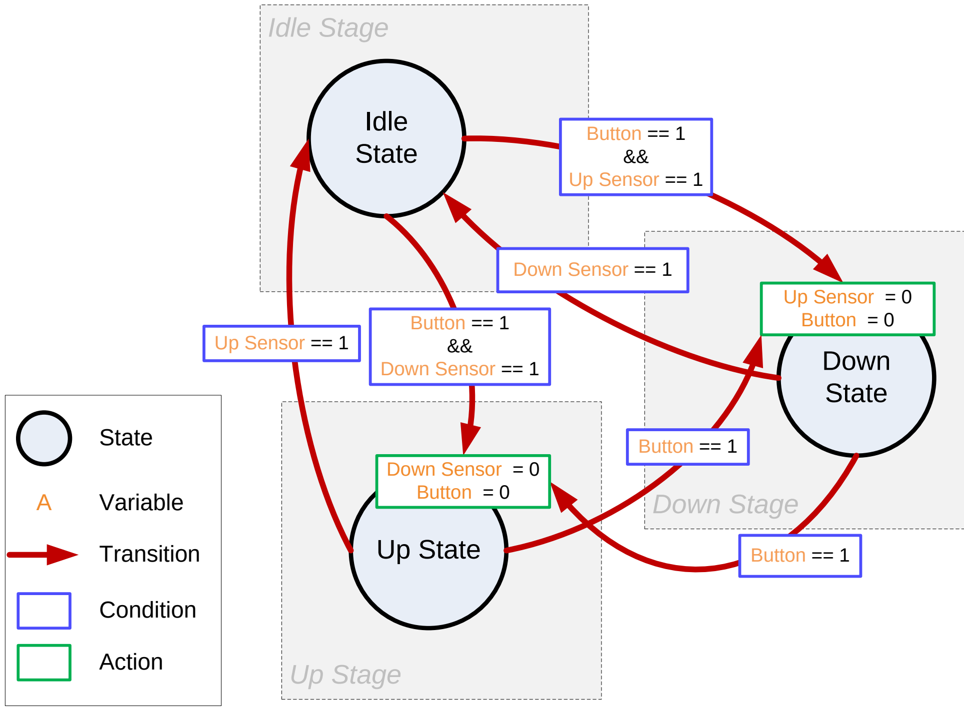


Figure 58: State Transition Map for Garage Door

Additional information about trajectory planning, motion sequences and SpinTAC™ Velocity Plan can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section Trajectory Planning).

This lab also demonstrates how to setup SpinTAC Velocity Plan in order to use the minimum amount of processor during the ISR. It splits the two function calls for SpinTAC Velocity Plan into part that runs in the main loop of the project and part that runs in the ISR of the project.

Project Files

There are no new project files.

Include the Header File

There are no new includes.

Declare the Global Structure

Lab 06d adds global variables to monitor the internal states of SpinTAC™ Velocity Plan and to control SpinTAC™ Velocity Plan. These variables are specific to the state machine that is implemented. Table 51 lists the variables that are added.

Table 51: Global object and variable declarations for SpinTAC Velocity Plan

globals		
	gSelectedPlan	Displays the Plan that is compiled into the project.
	GarageDoor	
	gGarageDoorDown	Active if the garage door is down
	gGarageDoorUp	Active if the garage door is up
	gGarageDoorButton	Button that controls the garage door operation
	GroceryConveyor	
	gGroceryConveyorOnOff	On/Off switch for the conveyor belt
	gGroceryConveyorProxSensor	Proximity sensor to detect groceries at the end of the conveyor belt

Configuring SpinTAC™ Velocity Plan

During the initialization and setup, the project will call a function to generally configure SpinTAC Velocity Plan, ST_setupVelPlan, and will call a separate function to load a motion sequence into SpinTAC Velocity Plan. This project contains three different functions to support the different state machines in this project.

Table 52: Functions that can be used to setup different motion sequences

Setup		
	SpinTAC	
	ST_setupVelPlan_GarageDoor	This function calls into SpinTAC Plan to configure the motion sequence for the Garage Door.
	ST_setupVelPlan_GroceryConveyor	This function calls into SpinTAC Plan to configure the motion sequence for the Grocery Conveyor.
	ST_setupVelPlan_TestPattern	This function calls into SpinTAC Plan to configure the motion sequence for the Test Pattern.

Main Run-Time loop (forever loop)

The main loop of the project has been modified to call SpinTAC Velocity Plan. It will call the components of SpinTAC Velocity Plan that do not need to be updated as part of the ISR. Table 53 lists the function that has been added into the main loop.

Table 53: Function to run SpinTAC Velocity Plan in Main Loop

main loop		
	SpinTAC	
	ST_runVelPlan	The ST_runVelPlan function calls the SpinTAC Velocity Plan object. This also handles enabling the SpinTAC Plan object.

Main ISR

The main ISR of the project has been modified to call only part of SpinTAC Velocity Plan. It will only call the components that need to be updated at the ISR frequency. Table 53 lists the function that has been removed from the main ISR.

Call SpinTAC™ Velocity Plan

The ST_runVelPlan code has been modified to interface with the sensors for the three state machines in this lab. It has also been modified to run in the Main Loop of the program. The only modification that needs to be done is that we should only return the variables from SpinTAC Velocity Plan when the FSM is in the STAY operation. This eliminates a race condition that exists when updating SpinTAC Velocity Plan variables from the Watch Window.

Table 54: InstaSPIN functions used in ST_runVelPlan

ST_runVelPlan		
	SpinTAC	
	STVELPLAN_getFsmState	The STVELPLAN_getFsmState function returns the current operation that the FSM is in.

Lab Procedure

In Code Composer, build lab6d, connect to the target and load the .out file.

- Open the command file "sw\solutions\instaspin_motion\src\proj_lab06d.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

To run the motor a couple of steps are required:

- To start the project, set the variable "gMotorVars.Flag_enableSys" equal to 1.
- To start the current loop controller, set the variable "gMotorVars.Flag_Run_Identify" equal to 1.

To start the motion sequence, the SpinTAC™ Velocity Plan button needs to be set to start once the FAST estimator is online.

- Set "gMotorVars.SpinTAC.VelPlanRun" to ST_PLAN_START to begin the motion sequence.

The motor will now run through a test profile where it oscillates between positive and negative speeds. Go ahead and modify the test profile to explore the capabilities of SpinTAC™ Velocity Plan.

To modify the state machine that the project uses:

- If SpinTAC™ Velocity Plan is currently running a state machine
  - Set "gMotorVars.SpinTAC.VelPlanRun" to ST_PLAN_STOP
- When "gMotorVars.SpinTAC.VelPlanStatus" is set to ST_PLAN_IDLE
  - Select the state machine you wish to run via "gSelectedPlan"

Refer to the background section in order to see the state transition map for each of the different state machines.

- After making code modifications, the project will need to be recompiled and the .out file loaded into the target
- While making modifications to the SpinTAC™ Velocity Plan configuration, you might encounter configuration errors. These will be indicated by the following variables:
  - "gMotorVars.SpinTAC.VelPlanErrorID"
    - Displays the error encountered by SpinTAC™ Velocity Plan. This indicates which function call has a configuration error.
  - "gMotorVars.SpinTAC.VelPlanCfgErrorIdx"
    - Displays the index that caused the configuration error. This should guide you to which instance of the function call has a configuration error.
  - "gMotorVars.SpinTAC.VelPlanCfgErrorCode"
    - Displays specifically what the configuration error is.

Feel free to modify the code in this lab to explore the advanced capabilities provided by SpinTAC Velocity Plan.

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how easy it is to design complex motion sequences using SpinTAC™ Velocity Plan. This lab allowed you to be creative and explore on your own how SpinTAC™ Velocity Plan can enable you to quickly design a motion sequence into your product.

Lab 6e – Dual Motor Sensorless Velocity InstaSPIN-MOTION

Abstract

Writeups will not be provided for InstaSPIN-MOTION proj_lab6e, proj_lab12c nor proj_lab13f. Please review the code to understand the technique and important variables employed in these labs.

A writeup for dual motor control using InstaSPIN-FOC is provided in the proj_lab10d writeup.

Proj_lab11d is a valid dual motor control project for InstaSPIN-FOC based on the simplified structure of proj_lab11. Labs 6e, 12c, and 13f are InstaSPIN-MOTION projects that build upon the lab 11d code base.

Please note that proj_lab11d does not include a trajectory module; as such, there is no acceleration variable, and any changes the speed reference are applied as a step input. Use caution. InstaSPIN-MOTION labs 6e, 12c, and 13f do not have this limitation

Lab 7 – 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 on the motor's windings. This resistance to temperature relationship is well defined depending on the materials used for the windings themselves.

In this lab, the user will exercise this feature by running a motor and enabling the Rs Online feature.

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

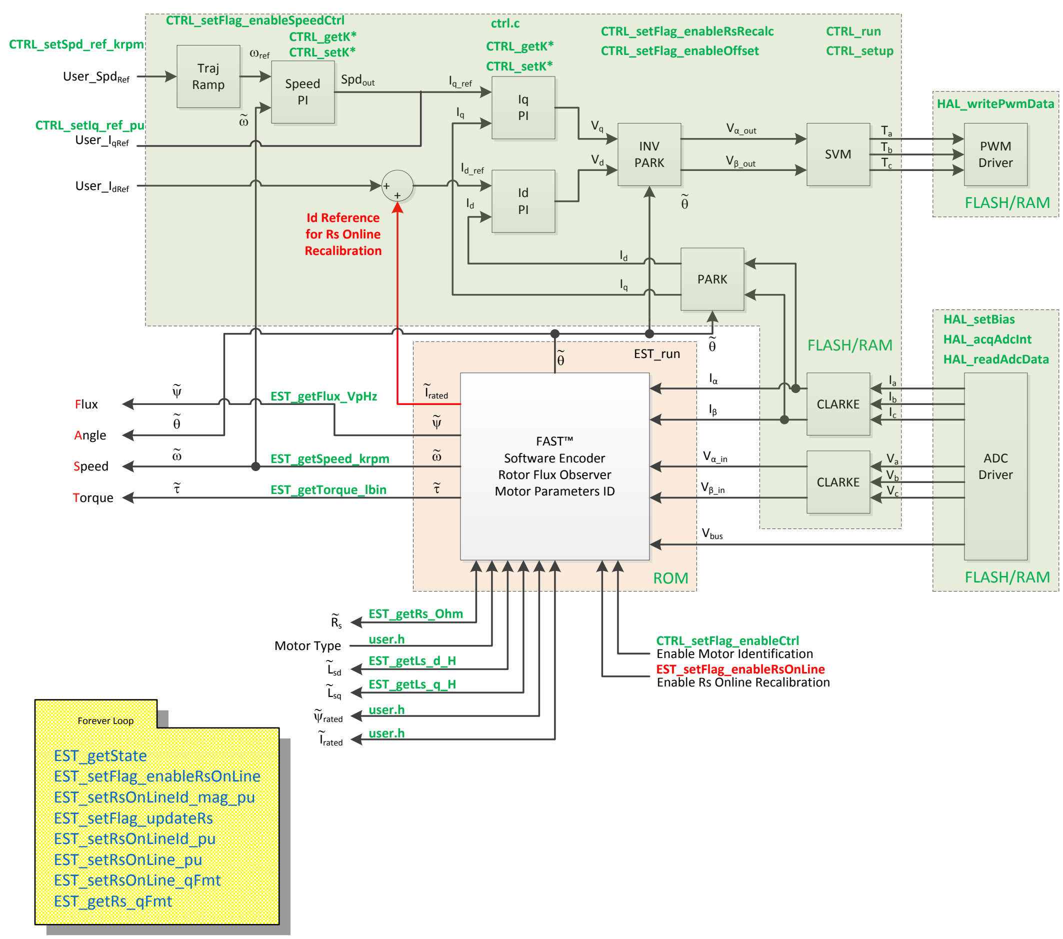
There are no new global object and variable declarations.

Initialization and Setup

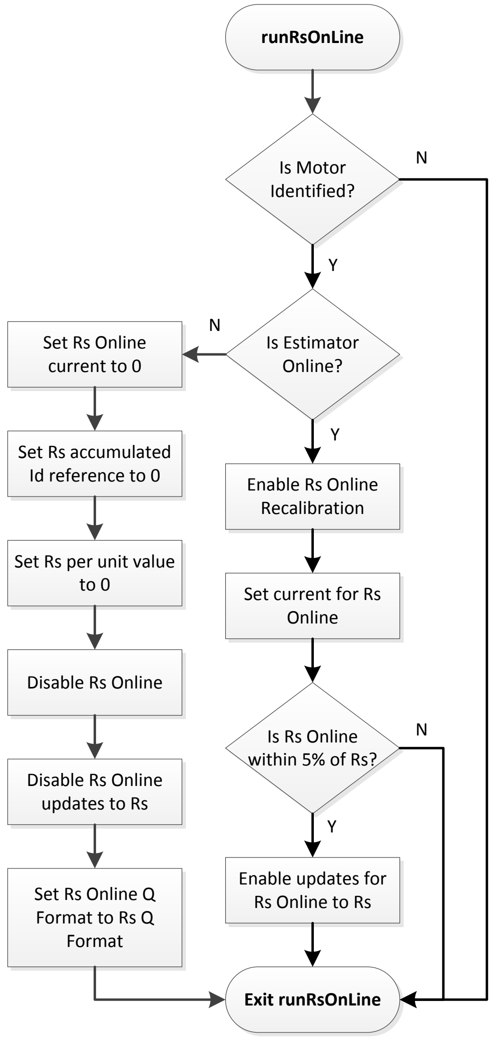
The following functions are new to this lab. These functions are all bundled in a new function called **runRsOnLine()** which is called from the main forever loop. This function contains the following estimator functions:

Forever Loop		
	EST	
	EST_getState	Gets the Estimator State to make sure the estimator is running before enabling Rs Online Recalibration
	EST_setFlag_enableRsOnLine	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_pu	This function sets the level of current to be generated in Id reference. The value needed by this function is in per unit so it needs to be scaled with USER_IQ_FULL_SCALE_CURRENT_A
	EST_setFlag_updateRs	When this function is called with a 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_pu	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_pu	It is recommended to call this function with a zero as a parameter to initialize the Rs Online value to zero before enabling Rs Online feature
	EST_setRsOnLine_qFmt	It is recommended to initialize the Q Format representation of Rs Online value with the value of Rs Q Format
	EST_getRs_qFmt	Q Format of Rs, which can be used to initialize Q Format of Rs Online

The following block diagram shows InstaSPIN, 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.



The following state machine is followed in lab 7 to allow Rs Online to work from the forever loop.

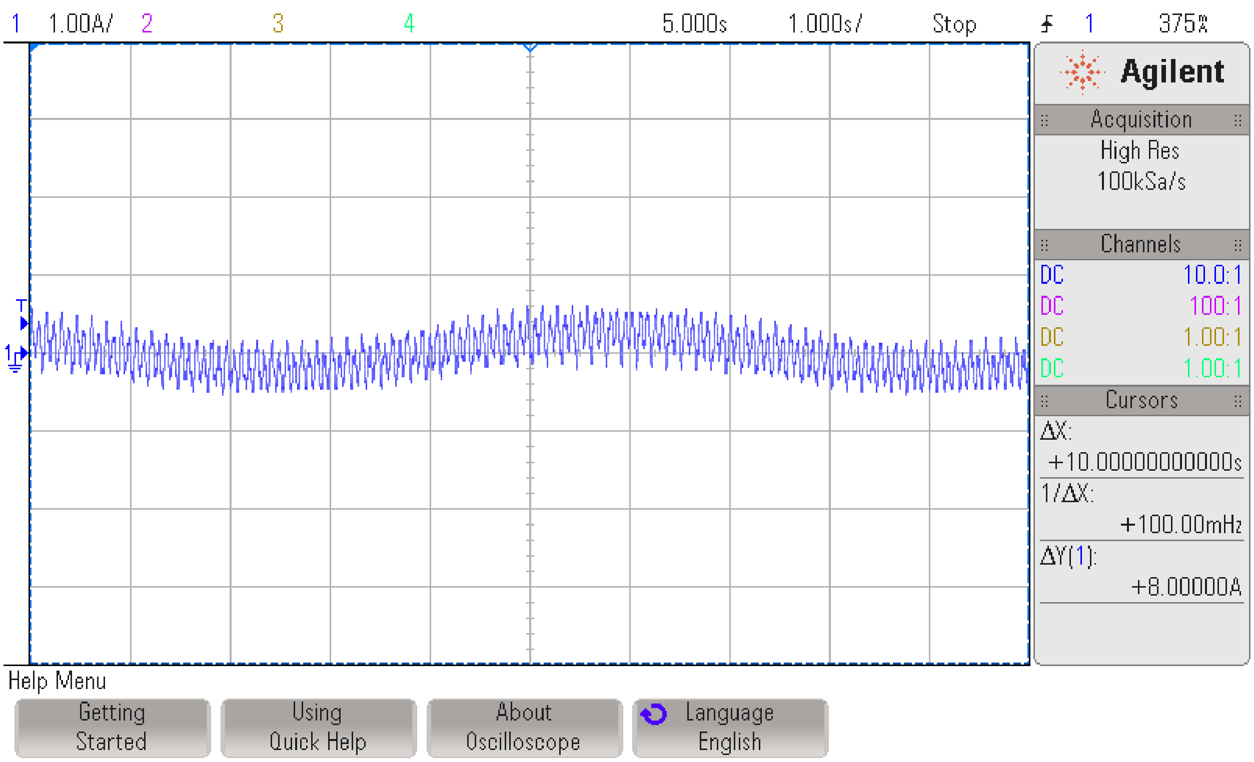


Lab Procedure

Build proj_lab07, connect to the target and load the .out file.

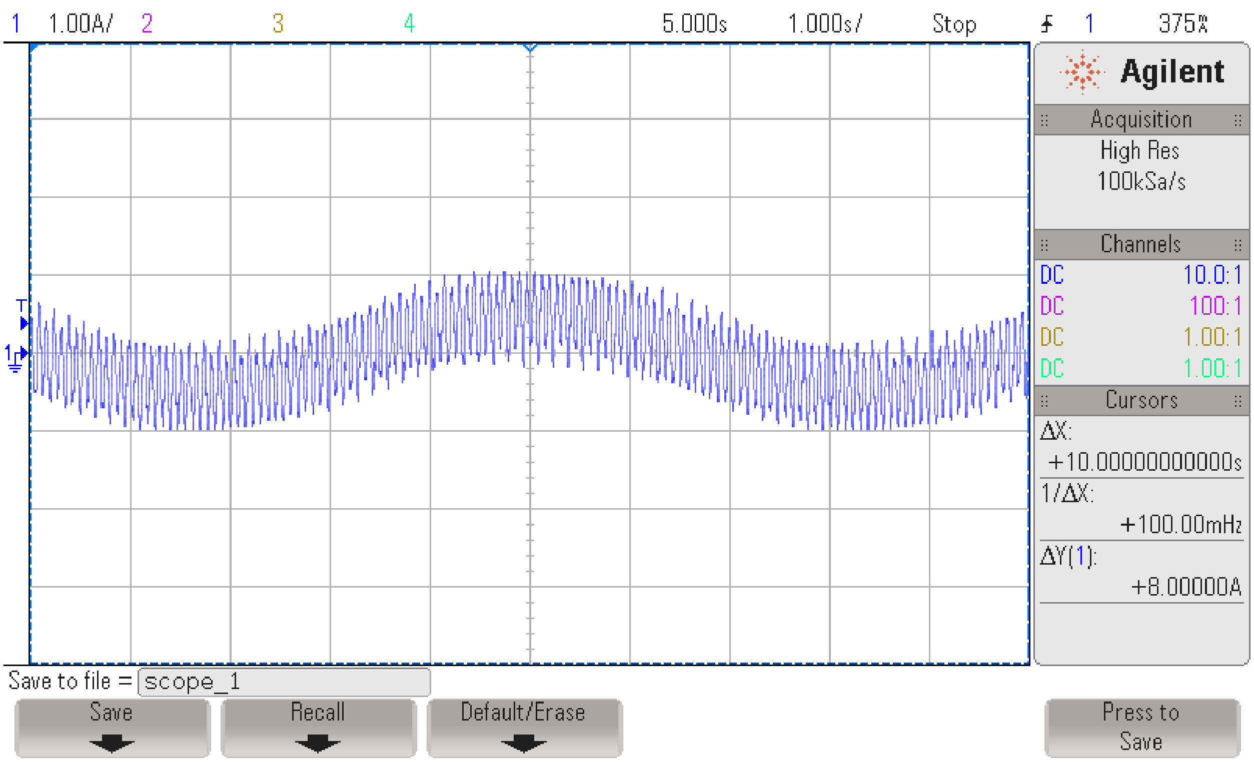
1. Add the appropriate watch window variables by calling the script “proj_lab07.js”.
2. Enable the real-time debugger.
3. Click the run button.
4. Enable continuous refresh on the watch window.

Once the motor starts running, the current will start looking as 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:



5. Now change the maximum amplitude used for Rs Online by changing the following variable to _IQ24(1.0):

(x)= gMotorVars.RsOnLineCurrent_A	1.0 (Q-Value(24))



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

(x)= gMotorVars.Rs_Ohm	0.3949452
(x)= gMotorVars.RsOnLine_Ohm	0.3949452

6. When done experimenting, set gMotorVars.Flag_Run_Identify 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.

Lab 7a – Using Rs Online Recalibration, with fpu32

Abstract

This lab runs Lab 7 with floating point unit enabled. This lab only applies to 6x devices, as it has a floating point unit.

Objectives Learned

Running Rs Online recalibration with fpu32 enabled.

Lab Procedure

Follow the exact same procedure as in Lab 7.

Conclusion

We conclude that the libraries in ROM also work when fpu32 is enabled in 6x devices.

Lab 9 – An Example in Automatic Field Weakening

Abstract

A simple procedure in automatic field-weakening is explored. The current voltage space vector is always compared to the maximum space vector. A voltage “head room” is maintained by controlling the negative Id current of the FOC controller.

Introduction

This automatic field weakening is done with a field weakening module. This module can be thought of a simple integral controller, in the sense that it increases or decreases its outputs depending on an error signal, but without a step (without a proportional gain).

Prerequisites

It assumes knowledge of up to proj_lab05b.

Objectives Learned

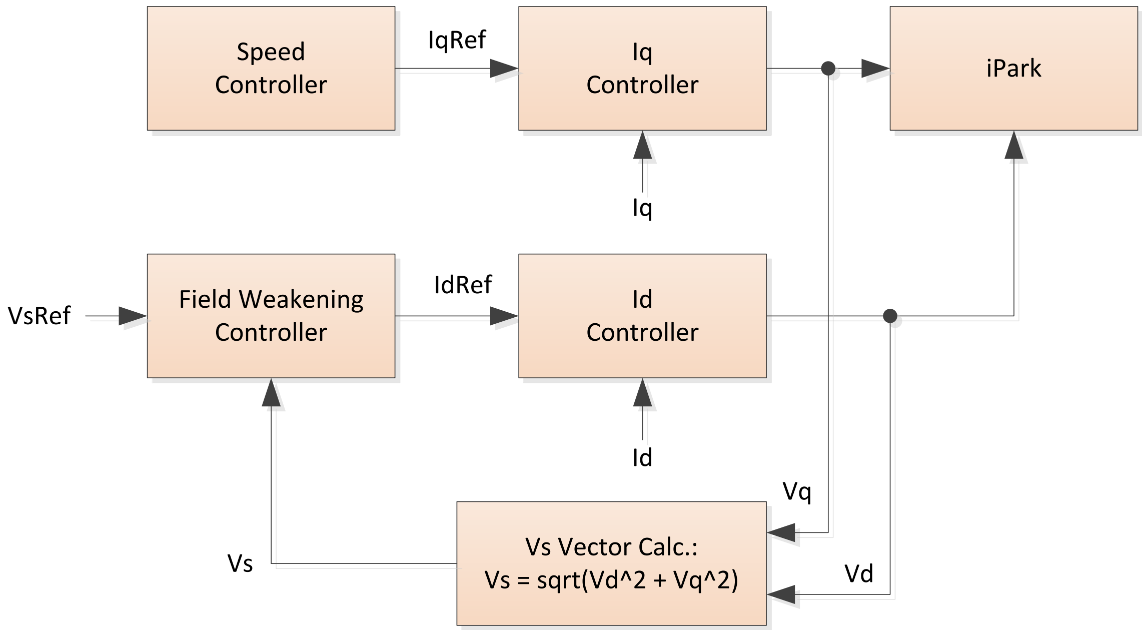
This project is used to introduce users to automatic field weakening.

Detailed Description

The automatic field weakening example shown in this project works as follows. First, the output vector, generated by the Id and Iq control loops, is calculated. That gives us a complete output vector, which we will call Vs. Then we compared that value against a reference, which is a user defined reference, of what is the maximum allowed output vector, VsRef. What that means is that if the output vector Vs is greater than VsRef, then field weakening will be applied.

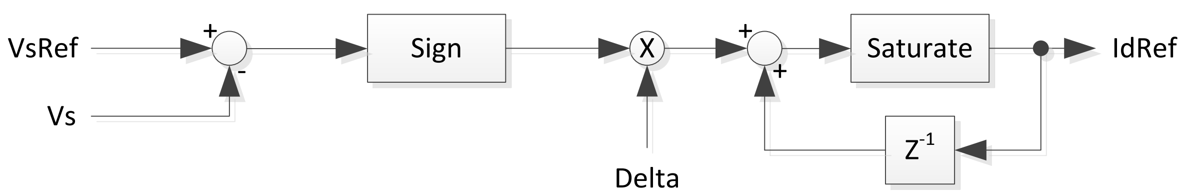
The resulting behavior will be that as the motor’s speed increases, Vs will grow beyond VsRef, and starting from that point, Id reference will start to grow negative, producing field weakening, and maintaining Vs controlled to VsRef.

The following block diagram shows two new blocks. The first one is called “Field Weakening Controller” which performs an adjustment of Id reference depending on an error signal. The second block is called “Vs Vector Calculation” which calculates the output vector from the two current controller modules.



The field weakening controller block diagram is an example of the Id reference adjustment. The intention of this project is to show an example of an Id reference adjustment depending on a reference signal, and users are encouraged to do their own algorithm for field weakening as per their specific requirements.

The following detailed field weakening controller is implemented in lab 9.



Lab Procedure

Step 1.

In user.h, a new #define is added to allow users to configure the limits of IdRef being produced by the automatic field weakening controller. This #define is shown here, as can be seen, the default value is set to 50% of the maximum current specified in the motor parameters section:

```
#define USER_MAX_NEGATIVE_ID_REF_CURRENT_A      (-0.5 * USER_MOTOR_MAX_CURRENT)
```

Make sure this value is a negative value, so that Id reference grows negatively.

Step 2.

Open project lab 9, build and load

Step 3.

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

- gMotorVars.Flag_enableFieldWeakening. This variable will be used to enable and disable field weakening.
- gMotorVars.VsRef. This is the field weakening reference. The units of this value would be from _IQ(0.0) to _IQ(2.0/3.0)
- gMotorVars.Vs. This is the output of both Id and Iq controllers combined in a vector, so Vs = sqrt(Vd^2 + Vq^2).
- gMotorVars.IdRef_A. You can monitor this variable to see the output of the field weakening algorithm as it adjusts IdRef.

Step 4.

Run motor by setting thest two flags to true (1): gMotorVars.Flag_enableSys = 1 and gMotorVars.Flag_Run_Identify = 1

(x)= gMotorVars.Flag_enableSys	1 (Decimal)
(x)= gMotorVars.Flag_Run_Identify	1 (Decimal)

Step 5.

Enable field weakening by setting this flag to true (1): gMotorVars.Flag_enableFieldWeakening = 1

(x)= gMotorVars.Flag_enableFieldWeakening	1 (Decimal)

Step 6.

Lower speed controller gains to conservative values, as the motor will be sped up to a high speed, and the speed controller does not need to be very responsive. This will also help with stability of the speed controller while in field weakening region

(x)= gMotorVars.Kp_spd	1.0 (Q-Value(24))
(x)= gMotorVars.Ki_spd	0.009999990463 (Q-Value(24))

Step 7.

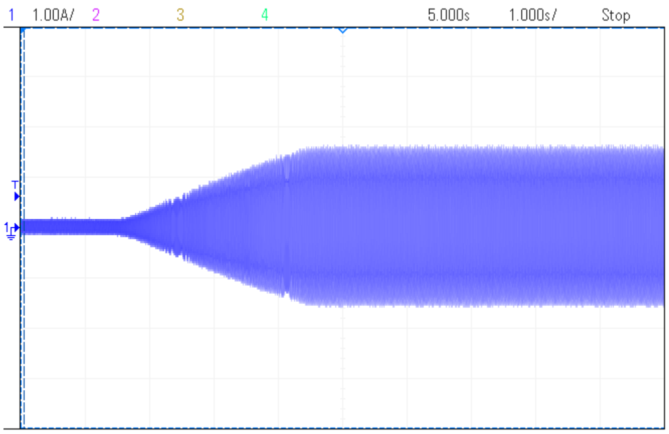
Increase the speed reference, so that the field weakening algorithm starts creating a negative output on Id reference. Keep in mind that if Vs is still lower than VsRef, Id reference won't change:

(x)= gMotorVars.Vs	0.481069684 (Q-Value(24))
(x)= gMotorVars.VsRef	0.5333333611 (Q-Value(24))
(x)= gMotorVars.IdRef_A	0.0 (Q-Value(24))

As speed increases, Id reference will start growing negative to control Vs:

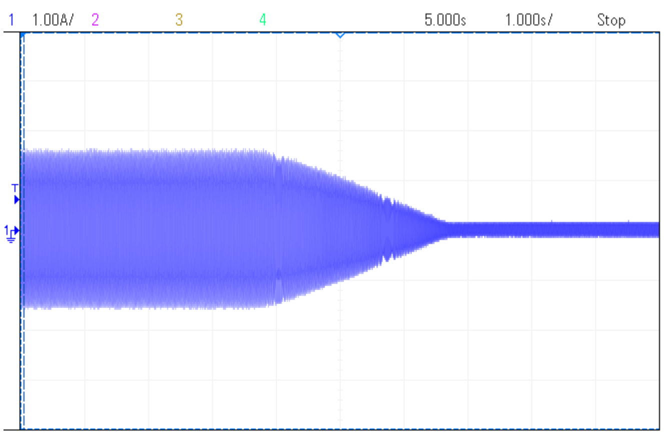
(x)= gMotorVars.Vs	0.5394797325 (Q-Value(24))
(x)= gMotorVars.VsRef	0.5333333611 (Q-Value(24))
(x)= gMotorVars.IdRef_A	-0.7057189941 (Q-Value(24))

This can also be seen in the currents. This scope shot was taken when the speed is entering the region where the automatic field weakening algorithm starts creating a negative Id reference.



Step 8.

Decrease the speed reference, so that the field weakening algorithm goes back to a zero Id reference, which is shown in the next scope plot:



Conclusion

A simple automatic field weakening algorithm is shown in this project. It is recommended that users implement and modify the automatic field weakening algorithm according to their requirements.

Lab 9a – An Example in Automatic Field Weakening, with fpu32

Abstract

This lab runs Lab 9 with floating point unit enabled. This lab only applies to 6x devices, as it has a floating point unit.

Objectives Learned

Running field weakening with fpu32 enabled.

Lab Procedure

Follow the exact same procedure as in Lab 9.

Conclusion

We conclude that the libraries in ROM also work when fpu32 is enabled in 6x devices.

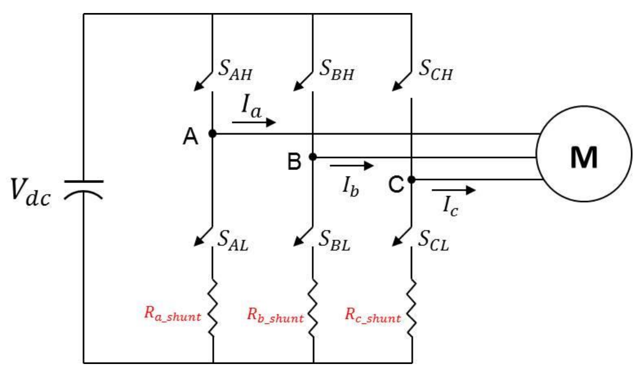
Lab 10a – An Example in Space Vector Over-Modulation

Abstract

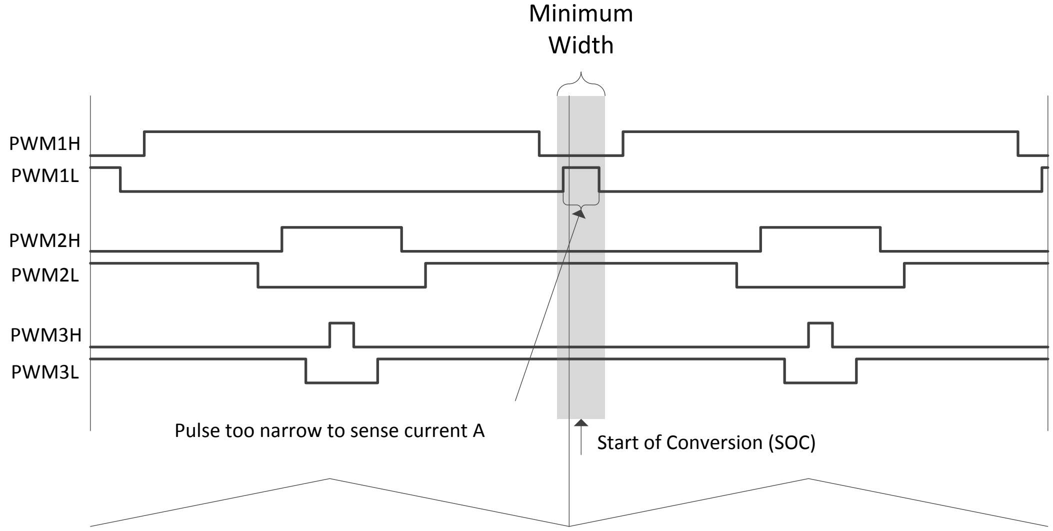
The SVM that is used by InstaSPIN is capable of saturating to a pre-specified duty cycle. When using a duty cycle over 100.0%, the SVM is considered to be in the over-modulation region. When in the over-modulation region, current shunt measurement windows become small or even disappear. This lab will 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. This provides an economic solution since the reference of the current measurement is the same as the microcontroller ground. At the same time though, it introduces a limitation, since the low side shunt resistor carries current only when the low side PWM is ON.



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 limits, and causes extensive periods of time where the low side PWM ON time basically disappears. The following time diagram 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 allows 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 the by using current symmetry, $I_a + I_b + I_c = 0$. Therefore, we can know all three phases current 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 for during over-modulation region, line-to-line voltage can be kept because the phase compensation method changes all three output voltages with same offset. Being able to reconstruct currents while doing over-modulation allows a field oriented control system to work even during heavy over-modulation, or trapezoidal control.

Prerequisites

It assumes knowledge of up to proj_lab05b.

Objectives Learned

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

Detailed Description

In lab 10a, a new approach to implementing over-modulation is done. 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 voltage compensation, this algorithm analyses about

how many currents are able to be measured with comparing to the user defined limit voltage that is the maximum output voltage to guarantee minimum duty. When two phase voltages are bigger than the limit voltage, find the middle magnitude of voltage and calculate the difference between middle magnitude of voltage and the user defined limit voltage to define offset. 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 analyses the PWM duty cycles and the ignore shunt value previously calculated to properly set the trigger for next ADC start of conversion signal.

Output Voltage Generation

The implementation of space vector modulation (SVM) allows inputs that go up to 2.0/3.0. So the amplitude of the inputs to SVM in Alpha and Beta coordinates can be from _IQ(0.0) up to _IQ(2.0/3.0). In order to make sure SVM can create outputs up to 2/3, CTRL_setParams() function should have the following:

```
// set the maximum modulation for the SVGEN module
maxModulation = _IQ(MATH_TWO_OVER_THREE);
SVGEN_setMaxModulation(obj->svgenHandle,maxModulation);
```

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:

```
// convert the ADC data
HAL_readAdcData(halHandle,&gAdcData);
```

With that function call, all currents are stored in the gAdcData.I structure. However, some of the currents stored 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, which we named SVGENCURRENT module. This module reconstructs phase currents in a simple way, depending on the state of an enumeration called: IgnoreShunt. The following logic is implemented as part of the SVGENCURRENT module in order to reconstruct the currents:

```
// select valid shunts and ignore one when needed
if (svgencurrent->IgnoreShunt==ignore_a)
{           // repair a based on b and c
  Ia = -Ib - Ic;       //Ia = -Ib - Ic;
}
else if (svgencurrent->IgnoreShunt==ignore_b)
{           // repair b based on a and c
  Ib = -Ia - Ic;       //Ib = -Ia - Ic;
}
else if (svgencurrent->IgnoreShunt==ignore_c)
{           // repair c based on a and b
```

```
    Ic = -Ia - Ib;        //Ic = -Ia - Ib;
  }
```

A second stage of current reconstruction is added to this lab, where we take care of corner case conditions when two out of the three currents 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 following code listing shows how this is done in lab 10:

```
  gIavg.value[0] += (gAdcData.I.value[0] - gIavg.value[0])>>gIavg_shift;
  gIavg.value[1] += (gAdcData.I.value[1] - gIavg.value[1])>>gIavg_shift;
  gIavg.value[2] += (gAdcData.I.value[2] - gIavg.value[2])>>gIavg_shift;

  if(measurableShuntThisCycle > two_phase_measurable)
  {
        gAdcData.I.value[0] = gIavg.value[0];
        gAdcData.I.value[1] = gIavg.value[1];
        gAdcData.I.value[2] = gIavg.value[2];
  }
```

After this second stage of current reconstruction, the measured currents and estimated currents are as close as we can have in order to have an FOC system running during extreme over-modulation conditions.

Output Voltage Compensation

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

- *Define compensation mode*
  In this function, all phase voltages are compared with $V_{LIM,}$ which is a maximum phase output voltage to guarantee the minimum duty for shunt current measurement, to check if the current can be measured at each voltage. If all phase voltages are less than $V_{LIM,}$ the compensation mode is defined as "*all_phase_measurable*" for all three currents sampling. If two phase voltages are less than $V_{LIM,}$ the compensation mode is defined as "*two_phase_measurable*". 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, the voltage compensation is carried out to be able to measure two phase currents at least through finding middle voltage V_{MIDDLE}.

- *Phase voltage compensation*
  In "*one_phase_measurable*" mode, all voltage reconstruction with offset voltage should be carried out to guarantee two phases measureable. The offset voltage is calculated as following equation and PwmData is recalculated with offset voltage.

```
      //phase voltage compensator
      if(svgencurrent->Mode == one_phase_measurable)
      {
            Voffset = (Vmid + Vmid_prev) - (dutyLimit <<1);
```

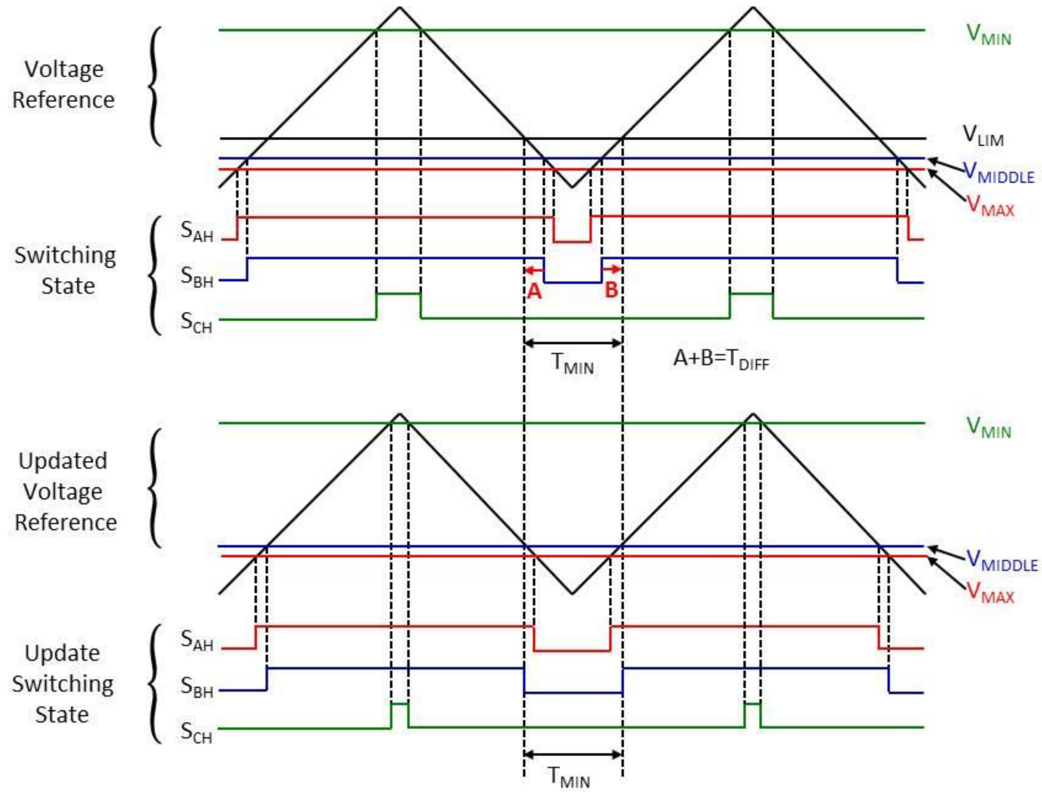
```
        pPwmData->value[0] -= Voffset;
        pPwmData->value[1] -= Voffset;
        pPwmData->value[2] -= Voffset;

}
```

- *Ignore Shunts*
  The third key module relates to knowing what 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:

  - *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*. This is simply 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.

```
// get ignore current
if(((pPwmData->value[0] + pPwmData_prev->value[0])>>1) > Vlimit)
{
        svgencurrent->IgnoreShunt = ignore_a;
}
else if(((pPwmData->value[1] + pPwmData_prev->value[1])>>1) > Vlimit)
{
        svgencurrent->IgnoreShunt = ignore_b;
}
else if(((pPwmData->value[2] + pPwmData_prev->value[2])>>1) > Vlimit)
{
        svgencurrent->IgnoreShunt = ignore_c;
}
else
{
        svgencurrent->IgnoreShunt = use_all;

}
```

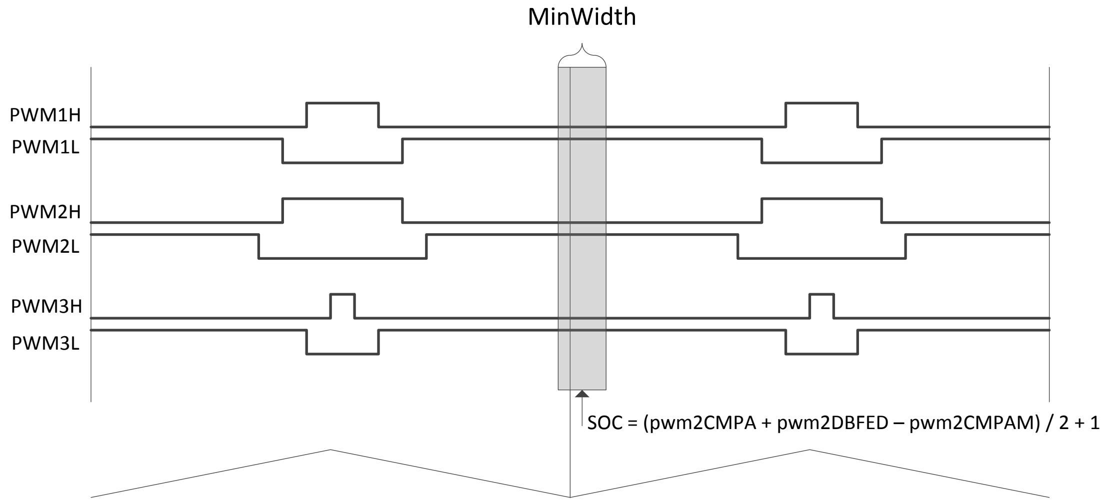


Setting the Start of Conversion (SOC) trigger

The last aspect of over-modulation is setting the start of conversion trigger in the right spot, so the best possible current measurement is taken in 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:

- IgnoreShunts. Depending on what shunts are ignored, the trigger changes to accommodate the best shunt.
- NextPulse1, 2 and 3 values. When all shunts are used, the setTrigger function needs to know what pulse is the narrowest ones, so the trigger is placed in the center of pulse.
- midVolShunt. When ignoring 1 or 2 shunts, the setTrigger function also needs to know what pulse is a middle length out of three pulses because the middle length pulse is the minimum duty, so the trigger is placed in the center of 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:



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.

In user.h, we need to have maximum phase voltage magnitude of 2/3 of Vbus as follows:

```
#define USER_MAX_VS_MAG_PU          (2.0/3.0)
```

Step 2.

Open project lab 10a, build and load

Step 3.

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

- gMotorVars.OverModulation. 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.
- 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 measureable.

Step 4.

Run motor by setting these two flags to true (1): gMotorVars.Flag_enableSys = 1 and gMotorVars.Flag_Run_Identify = 1

(x)= gMotorVars.Flag_enableSys	1 (Decimal)
(x)= gMotorVars.Flag_Run_Identify	1 (Decimal)

Step 5.

Increase the speed reference, gMotorVars.SpeedRef_krpm, until the ignore shunt value shows that shunts are being ignored as the motor spins:

(x)= svgencurrent.IgnoreShunt	ignore_b

Step 6.

Change maximum modulation value, and monitor the maximum speed you can reach. For example, using the booster pack with a DRV8305, and driving an Anaheim motor with 24V, these were the top speeds with each modulation value.

Maximum Output Vs Vector	Top Speed
0.5	5130 RPM
1/sqrt(3) = 0.5774	6000 RPM
2/3 = 0.6666	6340 RPM

Conclusion

In this lab, several aspects of overmodulation were discussed, allowing use 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.

Lab 10b – An Example in Space Vector Over-Modulation using InstaSPIN-MOTION

This example adds the InstaSPIN-MOTION speed controller and profile generator when doing over modulation.

Lab 10c – An Example in Space Vector Over-Modulation, with fpu32

Abstract

This lab runs Lab 10a with floating point unit enabled. This lab only applies to 6x devices, as it has a floating point unit.

Objectives Learned

Running over modulation with fpu32 enabled.

Lab Procedure

Follow the exact same procedure as in Lab 10a.

Conclusion

We conclude that the libraries in ROM also work when fpu32 is enabled in 6x devices.

Lab 10d – Dual Motor Sensorless Velocity InstaSPIN-FOC

Abstract

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

Introduction

In proj_lab10d, sensorless InstaSPIN-FOC is implemented to control two motors independently by one MCU.

Prerequisites

It assumes knowledge of proj_lab10a

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

In lab 10d, implement two motors controlled by one C2000 MCU using InstaSPIN-FOC. The same as controlling a single motor, define all variables for each of the two motors, set two independent interrupt subroutines **motor1_ISR()** and **motor2_ISR()** to run InstaSPIN-FOC time critical code.

To setup common hardware and each motor board parameters as below,

```
// set the common hardware abstraction layer parameters
HAL_setParams(halHandle,&gUserParams[HAL_MTR1]);

// Setup each motor board to its specific setting
HAL_setParamsMtr(halHandleMtr[HAL_MTR1],halHandle, &gUserParams[HAL_MTR1]);

// Setup each motor board to its specific setting
HAL_setParamsMtr(halHandleMtr[HAL_MTR2],halHandle, &gUserParams[HAL_MTR2]);
```

To improve dual motors' current sample at the right time point, using phase control between motor_1 PWM module and motor_2 PWM module, motor_1 as a master module and motor_2 as a slave module with a phase relationship of 180º. Configuring PWM modules of dual motors for phase control in `HAL_setupPwms()` which is defined in hal_2motors.c. The PWM output waveform is as Figure 59.

```
// HAL_setupPwms() for motor drive
void HAL_setupPwms(HAL_Handle_mtr handleMtr,HAL_Handle handle,const USER_Params
*pUserParams)
{

   if(objMtr->mtrNum == HAL_MTR1)
  {
      PWM_disableCounterLoad(objMtr->pwmHandle[0]);  // disable phase syncronization
  }
  else if(objMtr->mtrNum == HAL_MTR2)
  {
      PWM_enableCounterLoad(objMtr->pwmHandle[0]);   // enable phase syncronization
     PWM_setPhase(objMtr->pwmHandle[0],(halfPeriod_cycles>>1));   // half PWM period
  }
```

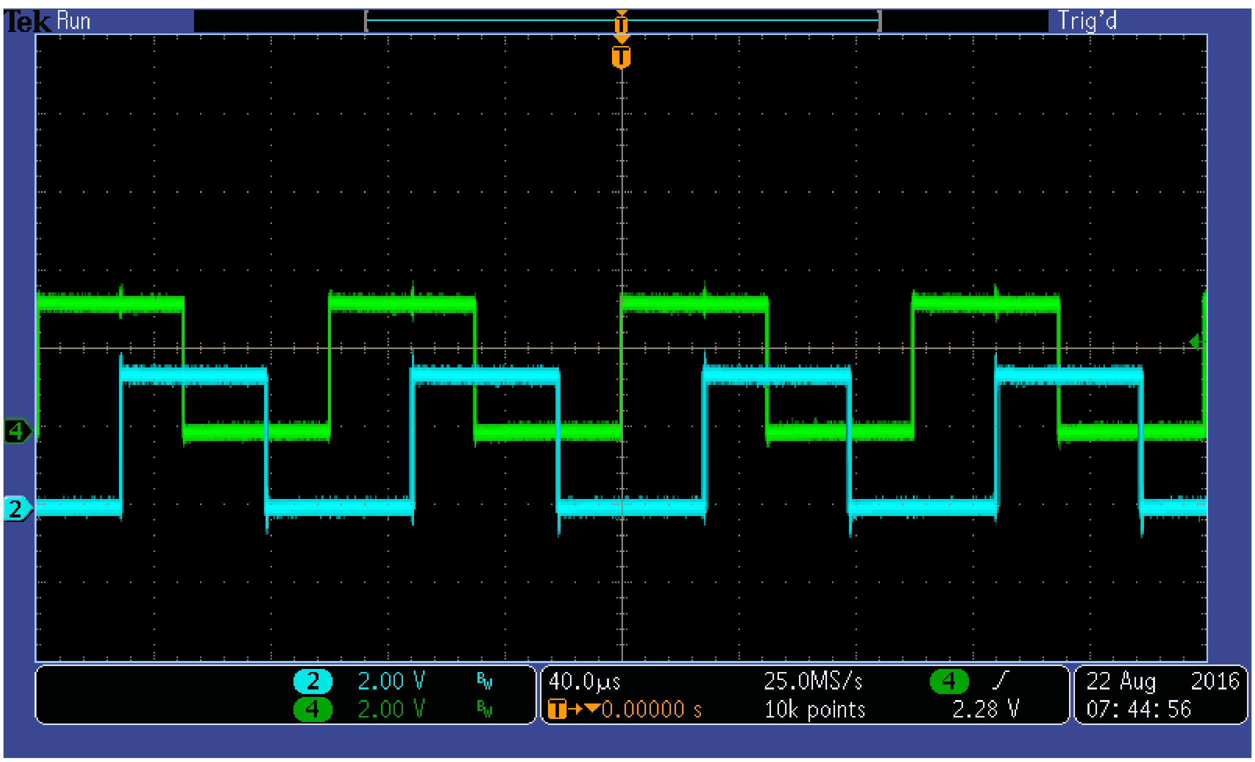


Figure 59: The PWM output for dual motors with phase of 180° (ch2 PWM_UH for motor_1, ch4 PWM_UH for motor_2)

CPU Usage Time Calculation

The **motor1_ISR()** and **motor2_ISR()** are time critical. When integrating your code into this ISR, it is important to verify that these two ISRs run in real-time.

This feature allows measuring time of CPU 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.

Step 1, add "\sw\modules\cpu_time\src\32b\cpu_time.c" to project, and include "sw/modules/cpu_time/src/32b/cpu_time.h" in "main_2motors.h".

Step 2, declare object and handle for this CPU_USAGE measurement as follows:

```
// define cpu_time object and handle for CPU usage time calculation
CPU_TIME_Handle  cpu_timeHandle[2];
CPU_TIME_Obj     cpu_time[2];
```

Step 3, initialize this module, so that timers are configured, and global variables are zeroed out.

```
// initialize the CPU usage module
cpu_timeHandle[motorNum] = CPU_TIME_init(&cpu_time[motorNum],sizeof(cpu_time[motorNum]));
CPU_TIME_setParams(cpu_timeHandle[motorNum], PWM_getPeriod(halHandleMtr[motorNum]-
>pwmHandle[0]));
```

Step 4, in order to measure the cycles that it takes to execute the ISR with this module, we add this at the very 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(cpu_timeHandle[HAL_MTR1],timer1Cnt);
```

Step 5, we need to call this at the end to get total number of cycles:

```
 // read the timer 1 value and update the CPU usage module
 timer1Cnt = HAL_readTimerCnt(halHandle,2);
 CPU_TIME_run(cpu_timeHandle[HAL_MTR1],timer1Cnt);
```

We can monitor the maximum, minimum and average usage time of CPU from watch window as below, you need to ensure the value of "cpu_time[n].timer_delta_max" is less than "cpu_time[n].pwm_period" – 100, to avoid ISR time overflow.

cpu_time[0]	struct _CPU_TIM...	{...}
pwm_period	unsigned short	4500
timer_cnt_now	unsigned long	278932729
timer_cnt_prev	unsigned long	278681272
timer_delta_now	unsigned long	534
timer_delta_prev	unsigned long	0
timer_delta_min	unsigned long	533
timer_delta_max	unsigned long	2865
timer_delta_avg	unsigned long	545
timer_band_max	unsigned long	3724539237
timer_delta_CntAcc	unsigned long	143
timer_delta_AccNum	unsigned long	80836
flag_resetStats	unsigned char	0 (Decimal)
cpu_time[1]	struct _CPU_TIM...	{...}
pwm_period	unsigned short	4500
timer_cnt_now	unsigned long	282098301
timer_cnt_prev	unsigned long	281918714
timer_delta_now	unsigned long	416
timer_delta_prev	unsigned long	0
timer_delta_min	unsigned long	415
timer_delta_max	unsigned long	2747
timer_delta_avg	unsigned long	426
timer_band_max	unsigned long	1239657554
timer_delta_CntAcc	unsigned long	210
timer_delta_AccNum	unsigned long	58262
flag_resetStats	unsigned char	0 (Decimal)

The function calculates a maximum, minimum and average of ISR CPU usage time. If users want to reset these values and restart the calculation, set this flag “cpu_time[n].flag_resetStatus” to 1.

PWMDAC

This module converts any s/w variables into the PWM signals in EPWMxA/B for C2000 MCU. Thus, it can be used to view the signal, represented by the variable, at the outputs of the PWMxA, PWMxB, pins through the external low-pass filters.

Step 1, declare object for this PWMDAC module.

```
// the PWMDAC variable
HAL_DacData_t gDacData;
```

Step 2, set the right offset and gain for each PWMDAC channel in proj_lab10d.c and hal_2motor.c

```
// set DAC parameters
HAL_setDacParameters(halHandle, &gDacData);

// set PWMDAC parameters for each channel to ensure the output waveform
void HAL_setDacParameters(HAL_Handle handle, HAL_DacData_t *pDacData)

{

   HAL_Obj *obj = (HAL_Obj *)handle;
   pDacData->PeriodMax = PWMDAC_getPeriod(obj->pwmDacHandle[PWMDAC_Number_1]);

   pDacData->offset[0] = _IQ(0.0);
   pDacData->offset[1] = _IQ(0.5);
   pDacData->offset[2] = _IQ(0.0);
   pDacData->offset[3] = _IQ(0.5);

   pDacData->gain[0] = _IQ(1.0);
   pDacData->gain[1] = _IQ(20.0);
   pDacData->gain[2] = _IQ(1.0);
   pDacData->gain[3] = _IQ(20.0);

} // end of HAL_setDacParameters() function
```

Step 3, connect inputs of the PWMDAC module.

```
 // connect inputs of the PWMDAC module
 gDacData.value[0] = gMotorVars[HAL_MTR1].angle_est_pu;
 gDacData.value[1] = gAdcData[HAL_MTR2].I.value[0];
 gDacData.value[2] = gMotorVars[HAL_MTR2].angle_est_pu;
 gDacData.value[3] = gAdcData[HAL_MTR2].I.value[0];

// run PWMDAC to output signal
 HAL_writeDacData(halHandle,&gDacData);
```

In lab, we connect the rotor angle and phase current of motor_1 and motor_2 to PWMDAC module, the output waveform on DACs of LAUNCHXL-F28069M is as Figure 60.

Ch1->DAC1: motor_1 rotor angle of the estimator, Ch2-> DAC2: motor_1 U phase sample current
Ch3->DAC3: motor_2 rotor angle of the estimator, Ch4-> DAC4: motor_2 U phase sample current

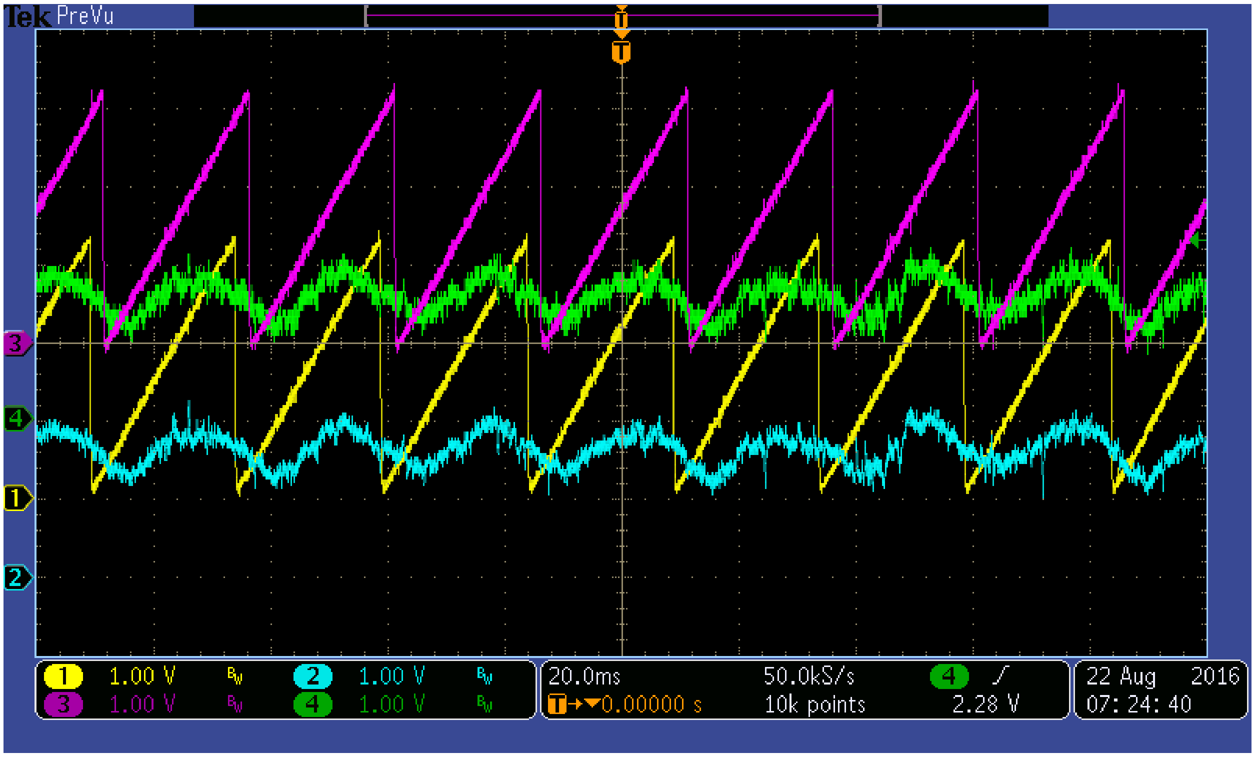


Figure 60: Rotor Angle and Phase Current Waveform Using PWMDAC Module

Project Files

Compare to proj_lab10a, we need to replace some of the original files with new files in proj_lab10d.

Table 51: New files that must be included in the project for dual motors control.

proj_lab10d		
	hal_2motors.c	Contains the various functions related to the HAL object (everything outside the CTRL system)
	user_2motors.c	Contains the function for setting initialization data to the CTRL, HAL, and EST modules

Includes

A description of the included files for proj_lab10d is shown in the below tables. Note that main_2motors.h is common across the project so there will be more included files than needed for this lab.

Table 52: Important header files needed for the dual motors control.

main_2motors.h	Header file containing all included files used in proj_lab10d.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	

	ctrl_2motors.h	Contains the public interface, object and function definitions for various functions related to the CTRL object.
	ctrl_obj_2motors.h	Defines the structures for the CTRL object
	hal_2motors.h	Contains public interface to various functions related to the HAL object
	hal_obj_2motors.h	Defines the structures for the HAL object
	user_2motors.h	Contains the public interface for user initialization data for the CTRL, HAL, and EST modules
	userParams_2motors.h	brief Defines the structures for the USER_Params object
	user_motor1.h	Contains the motor1 control initialization data for the CTRL, HAL, and EST modules
	user_motor2.h	Contains the motor2 control initialization data for the CTRL, HAL, and EST modules
	user_motor_on_j1.h	Contains the motor1 initialization parameters for the CTRL, HAL, and EST modules
	user_motor_on_j5.h	Contains the motor2 initialization parameters for the CTRL, HAL, and EST modules

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.

Table 53: Global object and variable declarations important for the setup

globals			
	CTRL_Handle	ctrlHandle[2]	the handle to a controller object (CTRL). The controller object implements all of the FOC algorithms and calls the FAST observer functions.
	MOTOR_Vars_t	gMotorVars[2]	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.
	HAL_PwmData_t	gPwmData[2]	the pwm voltage values for the three phases.
	HAL_AdcData_t	gAdcData[2]	the voltage and current adc values for 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	gSystemVars	this struct contains all of the flags and variables to control dual motors synchronously

Lab Procedure

Step 1.
Use one set of LAUNCHXL-F28069M and two sets of BOOSTXL-DRV8301 or BOOSTXL-DRV8305 to set up lab kit, connect the motor and power supply to kit.

Step 2.
In user_2motors.h, user_motor1.h, user_motor2.h, user_motor_on_j1.h and user_motor_on_j5.h, make sure motor parameters are known and correctly set. Lab 10d only works with dual PM motors.

Step 3.
In Code Composer, build proj_lab10d, connect to the target and load the .out file.

- Open the command file “sw\solutions\instaspin_foc\src\proj_lab10d.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 dual motors synchronously using the same flag and speed

- To start the project, set the variable “gSystemVars.Flag_enableSystem” equal to 1. In this lab, the variable will be set to 1 automatically.
- To enable synchronous control, set the variable “gSystemVars.Flag_enableSynControl” equal to 1.
- To start the current and speed loop controller, set the variable “gSystemVars.Flag_enableRun” equal to 1.
- The acceleration can be modified by adjusting the value in “gSystemVars.MaxAccelSet_krpmps”.
- Set a reference speed to “gSystemVars.SpeedSet_krpm” in order to run the motor at a target speed.

To run the dual motors independently using different flag and speed.

- To run the motor_1.
  - To disable synchronous control, set the variable “gSystemVars.Flag_enableSynControl” equal to 0.
  - To turn on the pwms to the motor, set the variable “gMotorVars[0].Flag_Run_Identify” equal to 1.
  - The acceleration can be modified by adjusting the value in “gMotorVars[0].MaxAccel_krpmps”.
  - Set a reference speed to “gMotorVars[0].SpeedRef_krpm” in order to run the motor_1 at a target speed.
- To run the motor_2.
  - To disable synchronous control, set the variable “gSystemVars.Flag_enableSynControl” equal to 0.
  - To turn on the controller and pwms of motor, set the variable “gMotorVars[1].Flag_Run_Identify” equal to 1.
  - The acceleration can also be modified by adjusting the value in “gMotorVars[1].MaxAccel_krpmps”.
  - Set a reference speed to “gMotorVars[1].SpeedRef_krpm” in order to run the motor_2 at a target speed.

Step 5.

When finished experimenting to stop the motor

- Set the variable "gSystemVars.Flag_enableRun" or "gMotorVars[n].Flag_Run_Identify" 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 showed how to use InstaSPIN-FOC in a dual motors system. It showed that it is easy to setup a dual motors sensorless control based on single MCU.

Lab 10e – Flying Start

Abstract

The Flying Start feature is used to start a rotating motor, as quick as possible and resume normal operation with a minimal impact on load or speed. The lab shows how to use the flying start function in InstaSPIN-FOC.

Introduction

In proj_lab10e, provide guidelines for applying flying start features 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 proj_lab10a

Objectives Learned

- The objective of this lab is to learn how to control motor using InstaSPIN-FOC with Flying Start function.

Detailed Description

The flying start is the capacity to start at any speed different to 0, which is an important function in some applications, for example, traction, washing machine, fan, e-bike, e-scooters.

When a motor is started in its normal mode it 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, 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 motors speed (frequency and phase) and voltage. The motor then accelerates to the commanded frequency. This process prevents an over current trip and significantly reduce 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 is present.

The sensorless flying start function implements a frequency search algorithm that searches for the rotor speed. The frequency search 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. As the stator voltage is measured in InstaSPIN drive, the speed and position are 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.

Initializing the Flying Start Module

The following code snippet shows how this Flying Start module is initialized and configured with default values.

```
// Initialize Flying Start (FS)
fsHandle = FS_init(&fs,sizeof(fs));

// Disable Flying Start (FS)
FS_setFlag_enableFs(fsHandle, false);

// Clear Flying Start(FS) check time count
FS_clearCntCheckTime(fsHandle);

// Set Flying Start(FS) minimum transition speed
FS_setSpeedFsMin_krpm(fsHandle, ctrlHandle, FS_SPEED_MIN);

// set Flying Start(FS) maximum check time
FS_setMaxCheckTime(fsHandle, FS_MAX_CHECK_TIME);

gMotorVars.Flag_enableSpeedCtrl = true;    // enable speed close loop control
gMotorVars.Flag_enableFlyingStart = true;  // enable Flying Start
```

As showed in following Flying Start control program flowchart, Flying Start module output a flag to enable or disable speed close loop control. A zero reference torque current is set and speed PI output is disabled when Flying Start works.

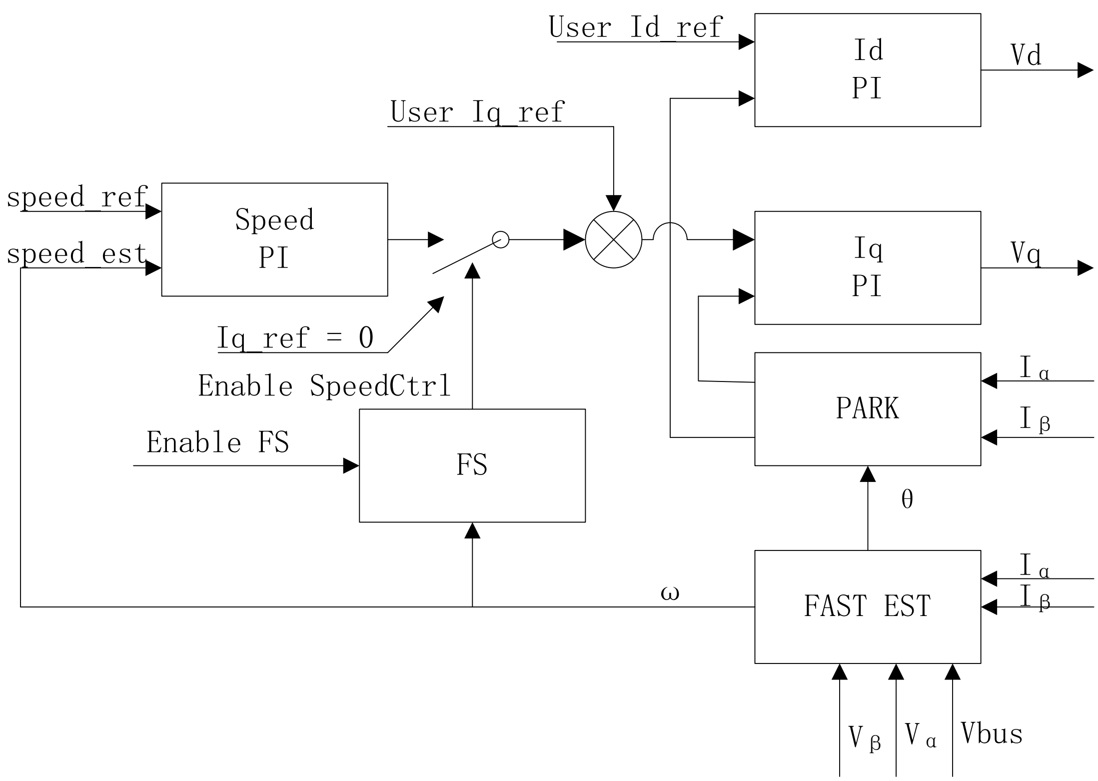


Figure 61: Flying Start Control Program Flowchart

The Flying Start algorithm is called in ISR.

```
// run the flying start function
FS_run(ctrlHandle, fsHandle);
```

The Flying Start variables and flags are updated in the background loop (outside of the ISR).

```
// Control motor Start or Stop with Flying Start
motor_RunCtrl(ctrlHandle);
```

As showed in following Flying Start module program flowchart, disable speed close loop, set reference Iq to Zero, then enable InstaSPIN-FOC module. Measure motor phase currents and voltages, run

InstaSPIN-FOC, the real motor speed can be estimated by InstaSPIN-FOC. Enable speed close loop and set reference value of speed PI when Flying start running time is up.

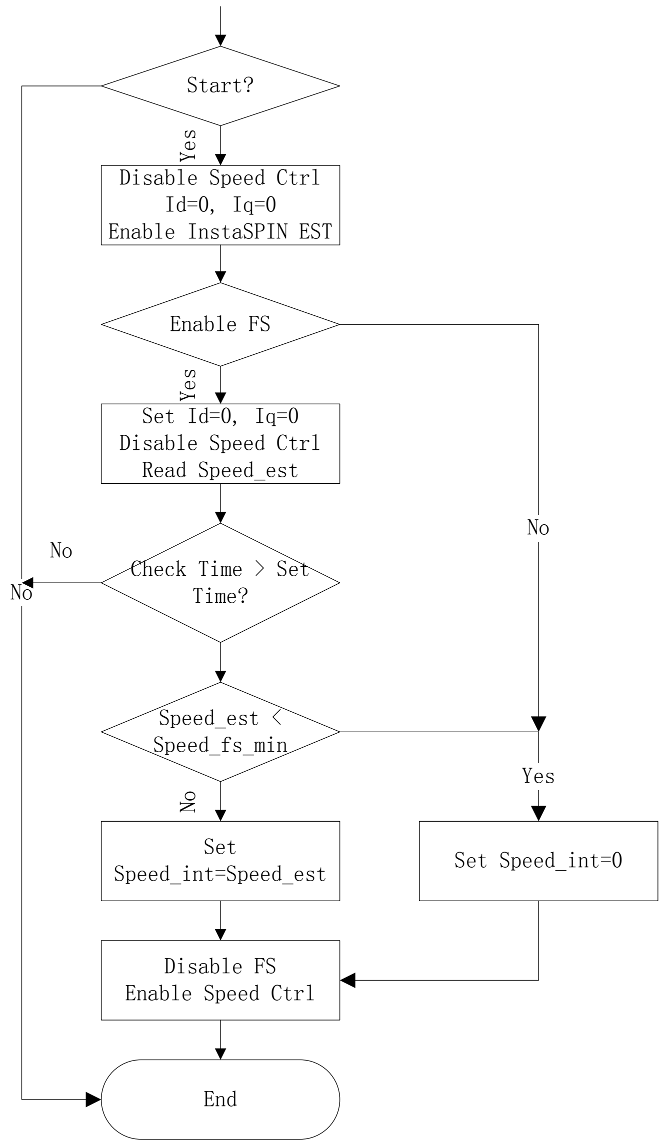


Figure 62: Flying Start Module Program Flowchart

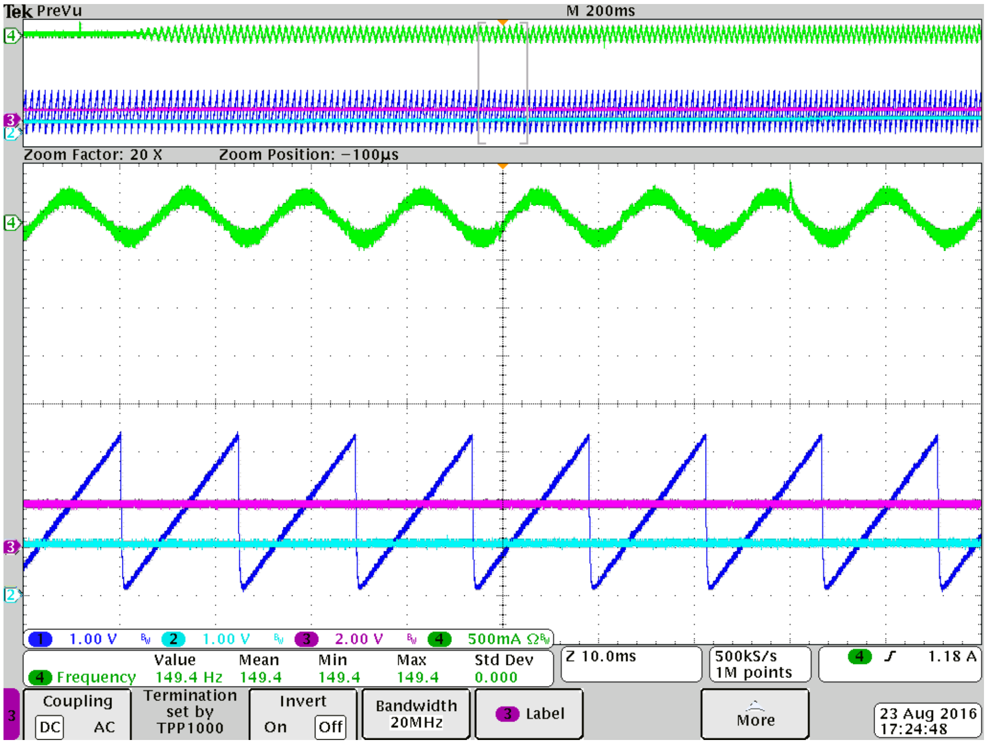


Figure 63: Motor Restart from non-Zero speed with Flying Start Function

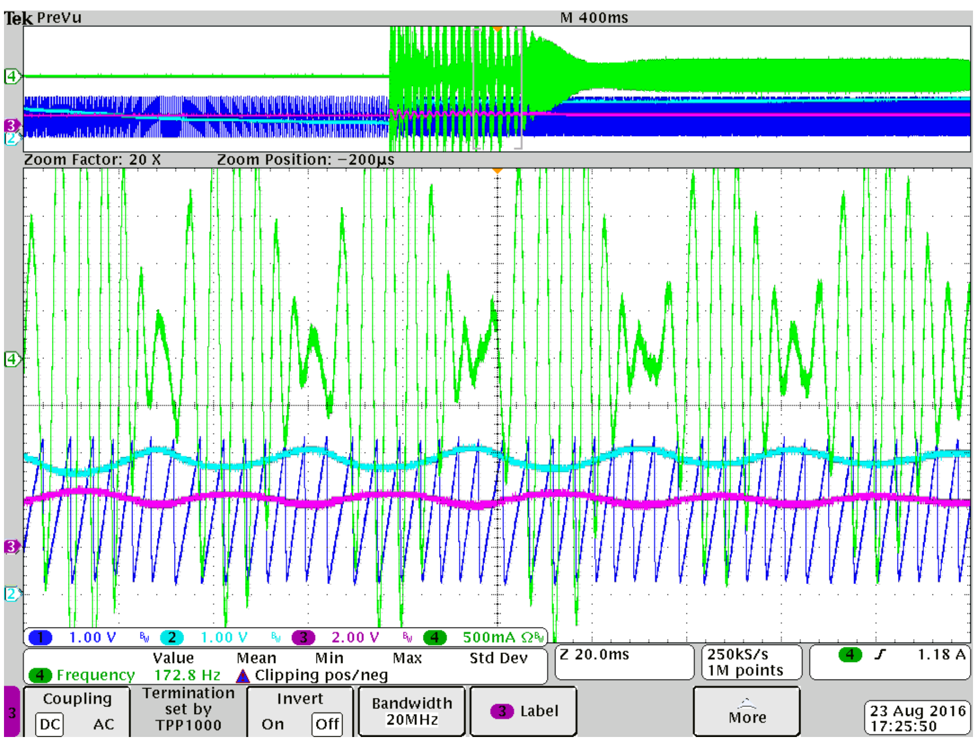


Figure 64: Motor Restart from non-Zero speed without Flying Start Function

Project Files

Compare to proj_lab10a, need to add some new files to proj_lab10d.

Table 54. New files that must be included in the project for Flying Start.

Proj_lab10e		
	flyingStart.c	Define the Flying Start (FS) module routines
	cpu_time.c	Define the CPU usage time (CPU_TIME) module routines

Includes

A description of the included files for proj_lab10e is shown in the below tables. Note that main.h is common across the project so there will be more included files than needed for this lab.

Table 55. Important header files needed for the Flying Start.

main.h	Header file containing all included files used in proj_lab10e.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
	flyingStart.h	Contains the public interface to the Flying Start (FS) module routines
	platforms	
	ctrl.h	Contains the public interface, object and function definitions for various functions related to the CTRL object.
	ctrl_obj.h	Defines the structures for the CTRL object
	user.h	Contains the motor control initialization data for the CTRL, HAL, and EST modules

Global Object and Variable Declarations

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

Table 56: Global object and variable declarations are important for the setup

globals			
	CTRL_Handle	ctrlHandle	the handle to a controller object (CTRL). The controller object implements all of the FOC algorithms and calls the FAST observer functions.
	MOTOR_Vars_t	gMotorVars	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.
	FS_Obj FS_Handle	fs fsHandle	The object and handle of a Flying Start struct.
	CPU_TIME_Obj CPU_TIME_Handle	cpu_time cpu_timeHandle	The object and handle of a CPU usage time struct.
	HAL_DacData_t	gDacData	The object of a PWMDAC.

CPU Usage Time Calculation

The mainISR() are time critical. When integrating your code into this ISR, it is important to verify that this ISR runs in real-time.

This feature allows measuring time of CPU 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.

Step 1, add “\sw\modules\cpu_time\src\32b\cpu_time.c” to project, and include "sw/modules/cpu_time/src/32b/cpu_time.h" in “main.h”.

Step 2, declare object and handle for this CPU_USAGE measurement as follows:

```
// define cpu_time object and handle for CPU usage time calculation
CPU_TIME_Handle  cpu_timeHandle;
CPU_TIME_Obj     cpu_time;
```

Step 3, initialize this module, so that timers are configured, and global variables are zeroed out.

```
// initialize the CPU usage module
cpu_timeHandle = CPU_TIME_init(&cpu_time,sizeof(cpu_time));
CPU_TIME_setParams(cpu_timeHandle, PWM_getPeriod(halHandle->pwmHandle[0]));
```

Step 4, in order to measure the cycles that it takes to execute the ISR with this module, we add this at the very 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(cpu_timeHandle,timer1Cnt);
```

Step 5, call this at the end to get total number of cycles:

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

We can monitor the maximum, minimum and average usage time of CPU form watch window as below, you need to ensure the value of “cpu_time.timer_delta_max” is less than “cpu_time.pwm_period”*2 –100, to avoid ISR time overflow.

cpu_time	struct _CPU_TIM...	{...}
pwm_period	unsigned short	3000
timer_cnt_now	unsigned long	2777293329
timer_cnt_prev	unsigned long	2777128250
timer_delta_now	unsigned long	2904
timer_delta_prev	unsigned long	0
timer_delta_min	unsigned long	0
timer_delta_max	unsigned long	3140
timer_delta_avg	unsigned long	2931
timer_band_max	unsigned long	2208588596
timer_delta_CntAcc	unsigned long	123
timer_delta_AccNum	unsigned long	426779
flag_resetStatus	unsigned char	0 (Decimal)

The function calculates a maximum, minimum and average of ISR CPU usage time. If users want to reset these values and restart the calculation, set this flag “cpu_time.flag_resetStatus” to 1.

PWMDAC

This module converts any s/w variables into the PWM signals in EPWMxA/B for C2000 MCU. Thus, it can be used to view the signal, represented by the variable, at the outputs of the PWMxA, PWMxB, pins through the external low-pass filters.

Step 1, declare object for this PWMDAC module.

```
 // the PWMDAC variable
HAL_DacData_t gDacData;
```

Step 2, set the right offset and gain for each PWMDAC channel in proj_lab10e.c and hal.c

```
 // set DAC parameters
HAL_setDacParameters(halHandle, &gDacData);

// set PWMDAC parameters for each channel to ensure the output waveform
void HAL_setDacParameters(HAL_Handle handle, HAL_DacData_t *pDacData)
{
   HAL_Obj *obj = (HAL_Obj *)handle;
   pDacData->PeriodMax = PWMDAC_getPeriod(obj->pwmDacHandle[PWMDAC_Number_1]);


   pDacData->offset[0] = _IQ(0.5);
   pDacData->offset[1] = _IQ(0.0);
   pDacData->offset[2] = _IQ(0.0);
   pDacData->offset[3] = _IQ(0.0);


   pDacData->gain[0] = _IQ(10.0);
   pDacData->gain[1] = _IQ(10.0);
   pDacData->gain[2] = _IQ(2.0);
   pDacData->gain[3] = _IQ(1.0);

} // end of HAL_setDacParameters() function
```

Step 3, connect inputs of the PWMDAC module.

```
  // get the estimator angle and frequency values
  gMotorVars.angle_est_pu = EST_getAngle_pu(ctrlHandle->estHandle);
  gMotorVars.speed_est_pu = EST_getFm_pu(ctrlHandle->estHandle);

  _iq Iq_pu = CTRL_getIq_in_pu(ctrlHandle);

  // connect inputs of the PWMDAC module.
  gDacData.value[0] = gAdcData.I.value[0];
  gDacData.value[1] = Iq_pu;
  gDacData.value[2] = gMotorVars.speed_est_pu;
  gDacData.value[3] = gMotorVars.angle_est_pu;

  HAL_writeDacData(halHandle,&gDacData);
```

In lab, we connect related variables to PWMDAC module, the output waveform on DACs of LAUNCHXL-F28069M is as Figure 65.
Ch1-> DAC4: rotor angle of the estimator
Ch2-> DAC3: rotor speed of the estimator,
Ch3-> DAC2: q-axis reference current
Ch4-> DAC1: U phase sample current,

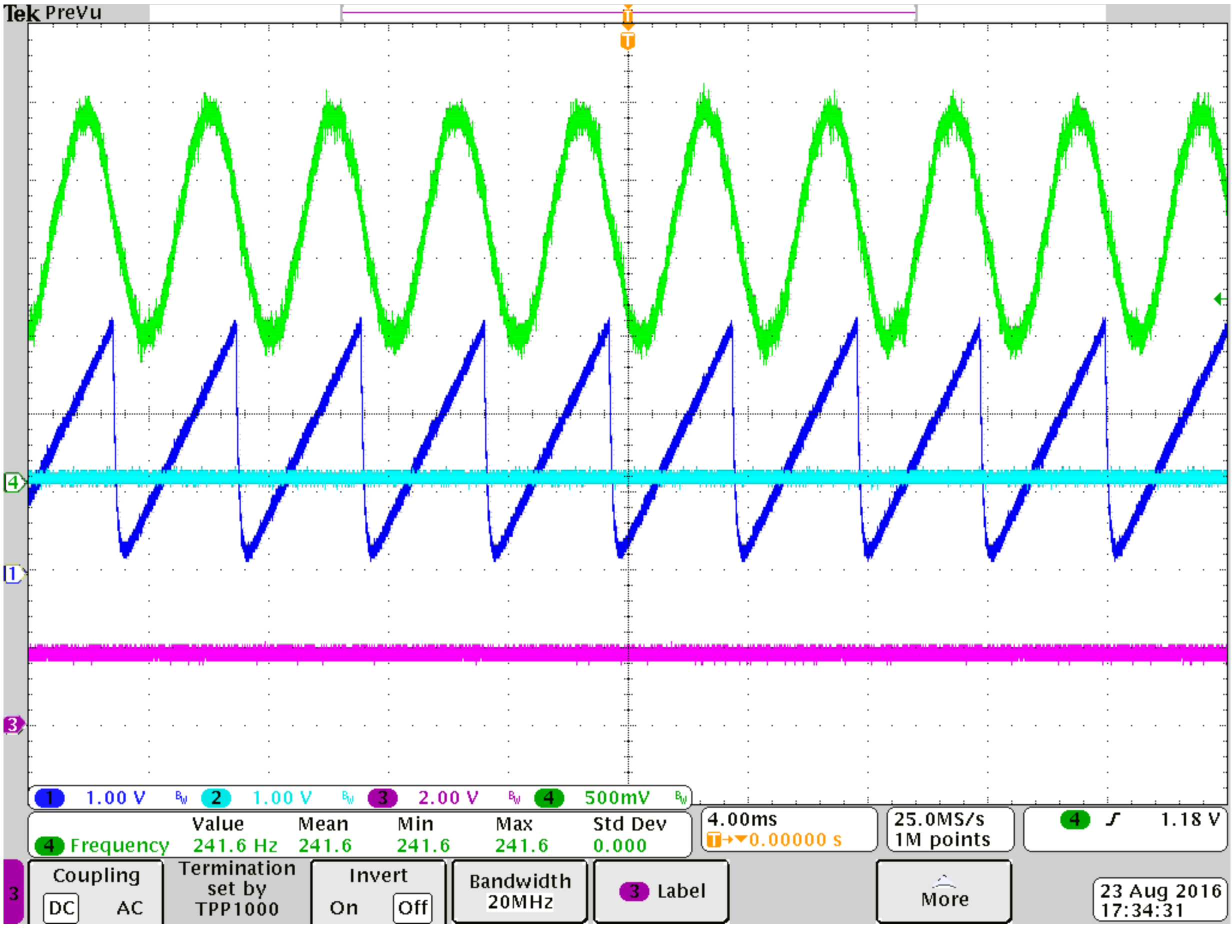


Figure 65: Motor Current, Speed, Angle Waveform Using PWMDAC Module

Lab Procedure

Step 1.
Use LAUNCHXL-F28069M and BOOSTXL-DRV8301 or BOOSTXL-DRV8305 to set up lab kit, connect correct and motor and power supply to kit.

Step 2.
In user.h, make sure motor parameters are known and correctly set. Lab 10e only works with PM motors.

Step 3.
In Code Composer, build proj_lab10e, connect to the target and load the .out file.

- Open the command file “sw\solutions\instaspin_foc\src\proj_lab10e.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 realtime 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 run the motor with flying start function

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To enable flying start function, set the variable “gMotorVars.Flag_enableFlyingStart” equal to 1.
- To turn on the pwms to the motor, set the variable “gMotorVars.Flag_enableRun” equal to 1.
- The acceleration can be modified by adjusting the value in “gMotorVars.MaxAccel_krpmps”.
- Set a reference speed to “gMotorVars.SpeedSet_krpm” in order to run the motor at a target speed.

Step 5.

When finished experimenting to stop the motor

- Set the variable “gMotorVars.Flag_enableRun” 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 flying start function in InstaSPIN-FOC. The flying start feature allows motor start at any speed different to zero without over current trips occur on drive.

Lab 11 – A Simplified Example without Controller Module

Abstract

This lab utilizes a simplified approach, so that users can see the entire field oriented control system, spelled out in the interrupt service routine. This can be thought of an approach that will be combined with user's code to create a production type of project.

Introduction

In a typical production software development related to MotorWare and InstaSPIN, no motor identification is required, and a simplified approach of the main ISR is needed. Also, the ROM function calls should be reduced to the minimum in order to have as much control of the software as possible. This project shows how to accomplish that. If Motor ID is required, please refer to previous labs, where the controller module is used, since motor ID requires a state machine that is implemented in the controller module.

Prerequisites

It assumes knowledge of up to proj_lab05b.

Objectives Learned

The objective of this lab is to learn how to have the most simplified interface to the functions in ROM, without the need of a controller object.

Detailed Description

Lab 11 can be divided into three main sections.

- Initialization of the estimator
- Background loop
- Main ISR

Initialization of the estimator

****Please note** – Rs online recalculation is *NOT* available in this lab. Changing the input parameter of `EST_setFlag_enableRsRecalc()` from `false` to `true` will cause runtime errors******

In this section, two different approaches are implemented depending on the version of InstaSPIN.

For version 1.7 implemented in 2xF and 5xF/M devices, the following estimator initialization is done in lab 11:

```
// initialize the estimator
estHandle = EST_init((void *)USER_EST_HANDLE_ADDRESS, 0x200);

// initialize the user parameters
USER_setParams(&gUserParams);

// set the hardware abstraction layer parameters
HAL_setParams(halHandle,&gUserParams);
```

```
// initialize the estimator
EST_setEstParams(estHandle,&gUserParams);
EST_setupEstIdleState(estHandle);
```

Those function calls will configure the estimator and will put it in a state ready to be enabled. Keep in mind that lab 11 does not support motor ID.

Two functions in this code example (EST_setEstParams() and EST_setupEstIdleState()) do not exist in ROM, and they have been created to support labs that do not use a controller module.

For version 1.6, implemented in 6xF/M devices, the following code example configures the estimator. Notice that we do have a CTRL_ function call, but this is actually a function call to ROM, and it is only needed for initialization of the estimator, hence no global controller object or global controller handle is needed in the project:

```
// initialize the estimator
estHandle = EST_init((void *)USER_EST_HANDLE_ADDRESS, 0x200);

// initialize the user parameters
USER_setParams(&gUserParams);

// set the hardware abstraction layer parameters
HAL_setParams(halHandle,&gUserParams);

{
  CTRL_Handle ctrlHandle = CTRL_init((void *)USER_CTRL_HANDLE_ADDRESS, 0x200);
  CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;
  obj->estHandle = estHandle;

  // initialize the estimator through the controller
  CTRL_setParams(ctrlHandle,&gUserParams);
  CTRL_setUserMotorParams(ctrlHandle);
  CTRL_setupEstIdleState(ctrlHandle);
}
```

Background Loop

The following background loop is needed to enable or disable the estimator, as well as to turn on or off the PWM

```
  // loop while the enable system flag is true
  while(gMotorVars.Flag_enableSys)
    {
      if(gMotorVars.Flag_Run_Identify)
        {
          // update estimator state
          EST_updateState(estHandle,0);

          // enable the PWM
          HAL_enablePwm(halHandle);
```

```
        }
      else
        {
          // set estimator to Idle
          EST_setIdle(estHandle);

          // disable the PWM
          HAL_disablePwm(halHandle);

          // clear integral outputs
          PID_setUi(pidHandle[0],_IQ(0.0));
          PID_setUi(pidHandle[1],_IQ(0.0));
          PID_setUi(pidHandle[2],_IQ(0.0));

          // clear Id and Iq references
          gIdq_ref_pu.value[0] = _IQ(0.0);
          gIdq_ref_pu.value[1] = _IQ(0.0);
        }
    } // end of while(gFlag_enableSys) loop
```

For simplicity purposes, other tasks used in the background loop are not shown here. Those additional functions are related to updating global variables for the watch window, enable and disable forced angle, etc. Only the critical functions are shown here.

Main ISR

The interrupt is generated by the end of conversion as the other labs do. The difference here in this main ISR is that there is no controller object. So even though the ISR looks more complex, it actually follows a standard field oriented control (FOC) implementation using source files from MotorWare that can be updated and tweaked by users. The main ISR is divided into several sections:

- *Forward FOC.* This includes getting ADC conversions, offset compensation of the converted values, and Clarke transforms of currents and voltages

```
  // acknowledge the ADC interrupt
  HAL_acqAdcInt(halHandle,ADC_IntNumber_1);

  // convert the ADC data
  HAL_readAdcDataWithOffsets(halHandle,&gAdcData);

  // remove offsets
  gAdcData.I.value[0] = gAdcData.I.value[0] - gOffsets_I_pu.value[0];
  gAdcData.I.value[1] = gAdcData.I.value[1] - gOffsets_I_pu.value[1];
  gAdcData.I.value[2] = gAdcData.I.value[2] - gOffsets_I_pu.value[2];
  gAdcData.V.value[0] = gAdcData.V.value[0] - gOffsets_V_pu.value[0];
  gAdcData.V.value[1] = gAdcData.V.value[1] - gOffsets_V_pu.value[1];
  gAdcData.V.value[2] = gAdcData.V.value[2] - gOffsets_V_pu.value[2];

  // run Clarke transform on current
  CLARKE_run(clarkeHandle_I,&gAdcData.I,&Iab_pu);
```

```
// run Clarke transform on voltage
CLARKE_run(clarkeHandle_V,&gAdcData.V,&Vab_pu);
```

- *Estimator Run.* This is the main function to ROM that executed the estimator

```
// run the estimator
EST_run(estHandle,
        &Iab_pu,
        &Vab_pu,
        gAdcData.dcBus,
        gMotorVars.SpeedRef_pu);
```

- *Extracting Estimated Variables.* A few function calls are needed in order to get variables to be controlled by a field oriented control system. Those variables are Angle, Speed, Id and Iq.

```
// generate the motor electrical angle
angle_pu = EST_getAngle_pu(estHandle);
speed_pu = EST_getFm_pu(estHandle);

// get Idq from estimator to avoid sin and cos
EST_getIdq_pu(estHandle,&gIdq_pu);
```

Notice that Id and Iq are read with EST_getIdq_pu() function call. First of all, that function was created to support projects without a controller object, so it is not in ROM, but a linked in pre-built library. It is useful to extract Id and Iq from the estimator object because it saves us execution cycles of sin(), cos() and a Park transform. However, if users want full control of how Id and Iq are calculated, the following code example also works, although it consumes more CPU cycles than just reading Id and Iq from the estimator:

```
// generate the motor electrical angle
angle_pu = EST_getAngle_pu(estHandle);
speed_pu = EST_getFm_pu(estHandle);

// compute the sin/cos phasor
phasor.value[0] = _IQcosPU(angle_pu);
phasor.value[1] = _IQsinPU(angle_pu);

// set the phasor in the Park transform
PARK_setPhasor(parkHandle,&phasor);

// run the Park module
PARK_run(parkHandle,&Iab_pu,&gIdq_pu);
```

- *Speed Control.* The first control action to be done is a speed controller. This gets a reference from a global variable, and an actual value from the estimated speed. A decimation rate defined in user.h is also used, to execute the speed controller every so often.

```
    // when appropriate, run the PID speed controller
    if(pidCntSpeed++ >= USER_NUM_CTRL_TICKS_PER_SPEED_TICK)
      {
        // clear counter
```

```
            pidCntSpeed = 0;

            // run speed controller
            PID_run_spd(pidHandle[0],
                        gMotorVars.SpeedRef_pu,
                        speed_pu,
                        &(gIdq_ref_pu.value[1]));
          }
```

- *Id Control.* Next, Id is controlled.

```
        // get the reference value
        refValue = gIdq_ref_pu.value[0];

        // get the feedback value
        fbackValue = gIdq_pu.value[0];

        // run the Id PID controller
        PID_run(pidHandle[1],refValue,fbackValue,&(gVdq_out_pu.value[0]));
```

- *Iq Control.* Lastly, Iq is controller.

```
        // get the Iq reference value
        refValue = gIdq_ref_pu.value[1];

        // get the feedback value
        fbackValue = gIdq_pu.value[1];

        // calculate Iq controller limits, and run Iq controller
        outMax_pu = _IQsqrt(_IQ(USER_MAX_VS_MAG_PU * USER_MAX_VS_MAG_PU)
                            -_IQmpy(gVdq_out_pu.value[0],gVdq_out_pu.value[0]));
        PID_setMinMax(pidHandle[2],-outMax_pu,outMax_pu);
        PID_run(pidHandle[2],refValue,fbackValue,&(gVdq_out_pu.value[1]));
```

Before actually running the controller of Iq, a PID output limit is calculated based on available voltage from an output vector, and a maximum limit specified in user.h.

- ***Inverse FOC.*** Once a new Vd and Vq vectors are generated by the Id and Iq controllers respectively, inverse transforms and space vector modulation is done. For this inverse FOC stage, the first thing to calculate is the output angle. This is based on the most recent angle estimation, and a compensation factor based on the estimated speed and the frequency of the PWM module. This step essentially compensates for the double buffered delay that exists in any digital control using mirrored PWM modules.

```
        // compensate angle for PWM delay
        angle_pu = angleDelayComp(speed_pu, angle_pu);
```

Once that angle is compensated, a new phasor is calculated to execute inverse Park transform:

```
        // compute the sin/cos phasor
        phasor.value[0] = _IQcosPU(angle_pu);
```

```
        phasor.value[1] = _IQsinPU(angle_pu);

        // set the phasor in the inverse Park transform
        IPARK_setPhasor(iparkHandle,&phasor);

        // run the inverse Park module
        IPARK_run(iparkHandle,&gVdq_out_pu,&Vab_pu);
```

Up until this point, Valpha and Vbeta are known and stored in variable Vab_pu. The next step is to compensate for any drop in Vbus. This is done by reading the 1/Vbus calculation done inside the estimator, and compensating Vab_pu with that value. Then SVM is executed using the 1/Vbus compensated values:

```
        // run the space Vector Generator (SVGEN) module
        oneOverDcBus = EST_getOneOverDcBus_pu(estHandle);
        Vab_pu.value[0] = _IQmpy(Vab_pu.value[0],oneOverDcBus);
        Vab_pu.value[1] = _IQmpy(Vab_pu.value[1],oneOverDcBus);
        SVGEN_run(svgenHandle,&Vab_pu,&(gPwmData.Tabc));
```

- ***Writing PWM Values.*** At the end of the ISR, the new calculated values are written to the PWM module through the HAL as follows:

```
    // write the PWM compare values
    HAL_writePwmData(halHandle,&gPwmData);
```

Lab Procedure

Step 1.

In user.h, make sure offsets and motor parameters are known and correctly set. Lab 11 only works with PM motors:

```
#define   I_A_offset (1.210729778)
#define   I_B_offset (1.209441483)
#define   I_C_offset (1.209092796)

#define   V_A_offset (0.5084558129)
#define   V_B_offset (0.5074239969)
#define   V_C_offset (0.5065535307)
```

```
#elif (USER_MOTOR == Anaheim_BLY172S)
#define USER_MOTOR_TYPE                 MOTOR_Type_Pm
#define USER_MOTOR_NUM_POLE_PAIRS       (4)
#define USER_MOTOR_Rr                   (NULL)
#define USER_MOTOR_Rs                   (0.4)
#define USER_MOTOR_Ls_d                 (0.00067)
#define USER_MOTOR_Ls_q                 (0.00067)
#define USER_MOTOR_RATED_FLUX           (0.034)
#define USER_MOTOR_MAGNETIZING_CURRENT  (NULL)
#define USER_MOTOR_RES_EST_CURRENT      (1.0)
```

```
#define USER_MOTOR_IND_EST_CURRENT      (-1.0)
#define USER_MOTOR_MAX_CURRENT          (5.0)
#define USER_MOTOR_FLUX_EST_FREQ_Hz     (20.0)
```

Step 2.

Open project lab 11, build and load

Step 3.

Load the variables by opening the script: proj_lab11.js. In this particular lab, the following variables will be used:

- gMotorVars.SpeedRef_krpm. This variable will be used to set the speed reference in kilo RPM. **Keep in mind that in this project there is no ramp, so changes to the speed reference will be effective immediately, causing a possible overcurrent condition, or if decelerating a motor this can cause an overvoltage.**
- gMotorVars.Speed_krpm. This variable is used to monitor the speed reference in kilo RPM.
- pid[0].Kp, pid[0].Ki. These two variables are used for the speed controller gains.
- pid[1].Kp, pid[1].Ki. These two variables are used for the Id current controller gains.
- pid[2].Kp, pid[2].Ki. These two variables are used for the Iq current controller gains.

Step 4.

Run motor by setting thest two flags to true (1): gMotorVars.Flag_enableSys = 1 and gMotorVars.Flag_Run_Identify = 1

(x)= gMotorVars.Flag_enableSys	1 (Decimal)
(x)= gMotorVars.Flag_Run_Identify	1 (Decimal)

Step 5.

Experiment with speed controller reference and gains. Change the speed reference with this variable: gMotorVars.SpeedRef_krpm, and monitor the estimated speed with variable: gMotorVars.Speed_krpm. Change speed controller gains with these variables: pid[0].Kp and pid[0].Ki.

Conclusion

After experimenting with this lab, users can modify the source files to accommodate their own requirements, without a controller object that encapsulates a lot of these modules. This lab shows all the FOC blocks individually called in the ISR, without other abstraction layers.

Lab 11a – A Feature Rich Simplified Example without Controller Module

Abstract

Since the inception of InstaSPIN, users have been looking for an example with the least amount of ROM function calls, very straight forward ISR, and with all the features that InstaSPIN provides. Also, users are interested in not having a high level controller module, so that users have the flexibility to modify the project without too many levels of abstraction. This lab provides users both benefits of not having a highly integrated controller module, and at the same time having all the features InstaSPIN brings to sensorless motor control.

Introduction

In lab 11 we showed a bare bones simplified project without a controller that only supports PM motors. Lab 11a adds all the features that make InstaSPIN a complete solution for motor control. The features included in lab 11a are:

- Simplified approach by not having a controller object. Same feature as lab 11.
- Offset recalculation
- Rs recalculation
- Speed and Id ramps for smooth reference changes
- Both motors ACIM and PM/IPM are supported
- PowerWarp
- Rs OnLine
- Overmodulation
- Field Weakening
- 1/Vbus compensation
- CPU usage calculation
- No Motor ID

Prerequisites

It assumes knowledge of proj_lab11.

Objectives Learned

The objective of this lab is to learn how to have the most simplified interface to the functions in ROM, and at the same time have a feature rich example that can be used by users to go to production with.

Detailed Description

Taking lab 11 as a starting point, the following sections will describe each additional feature in detail.

Offset Recalculation

This feature allows recalculating voltages and currents offsets as desired while the motor is at standstill. The approach taken to calculate the offsets in this lab is by using 6 first order filters declared at the top of lab 11a:

```
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
```

Then, the filters are initialized using the cutoff frequency specified in user.h with: #define USER_OFFSET_POLE_rps

```
  // initialize and configure offsets using filters
  {
    uint16_t cnt = 0;
    _iq b0 = _IQ(gUserParams.offsetPole_rps/(float_t)gUserParams.ctrlFreq_Hz);
    _iq a1 = (b0 - _IQ(1.0));
    _iq b1 = _IQ(0.0);

    for(cnt=0;cnt<6;cnt++)
      {
        filterHandle[cnt] = FILTER_FO_init(&filter[cnt],sizeof(filter[0]));
        FILTER_FO_setDenCoeffs(filterHandle[cnt],a1);
        FILTER_FO_setNumCoeffs(filterHandle[cnt],b0,b1);
        FILTER_FO_setInitialConditions(filterHandle[cnt],_IQ(0.0),_IQ(0.0));
      }

    gMotorVars.Flag_enableOffsetcalc = false;
  }
```

The actual offsets are calculated only when the enable flag is set, and this logic is checked in the ISR as follows:

```
  else if(gMotorVars.Flag_enableOffsetcalc == true)
    {
      runOffsetsCalculation();
    }
```

That function is explained next. As can be seen, the duty cycles are all set to 50% by writing _IQ(0.0) to the Tabc values, and the filters are run. Once a time has elapsed, the calculated offsets are stored in global variables.

```
void runOffsetsCalculation(void)
{
  uint16_t cnt;

  // enable the PWM
  HAL_enablePwm(halHandle);

  for(cnt=0;cnt<3;cnt++)
    {
      // Set the PWMs to 50% duty cycle
      gPwmData.Tabc.value[cnt] = _IQ(0.0);
```

```
      // reset offsets used
      gOffsets_I_pu.value[cnt] = _IQ(0.0);
      gOffsets_V_pu.value[cnt] = _IQ(0.0);

      // run offset estimation
      FILTER_FO_run(filterHandle[cnt],gAdcData.I.value[cnt]);
      FILTER_FO_run(filterHandle[cnt+3],gAdcData.V.value[cnt]);
    }

  if(gOffsetCalcCount++ >= gUserParams.ctrlWaitTime[CTRL_State_OffLine])
    {
      gMotorVars.Flag_enableOffsetcalc = false;
      gOffsetCalcCount = 0;

      for(cnt=0;cnt<3;cnt++)
        {
          // get calculated offsets from filter
          gOffsets_I_pu.value[cnt] = FILTER_FO_get_y1(filterHandle[cnt]);
          gOffsets_V_pu.value[cnt] = FILTER_FO_get_y1(filterHandle[cnt+3]);

          // clear filters
          FILTER_FO_setInitialConditions(filterHandle[cnt],_IQ(0.0),_IQ(0.0));
          FILTER_FO_setInitialConditions(filterHandle[cnt+3],_IQ(0.0),_IQ(0.0));
        }
    }

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

Rs Recalculation

This feature allows recalculating the stator resistance. The way this works in lab 11a is different than the Rs Recalculation used in previous labs. In lab 11a, setting the enable flag will automatically run the recalculation, and once it is done, the estimator will be placed in Idle, so the motor won't run as it does in other labs right after Rs recalculation. First, we need to have a trajectory in Id, since we will use Id reference to recalculate Rs:

```
TRAJ_Handle      trajHandle_Id;  //!< the handle for the id reference trajectory
TRAJ_Obj         traj_Id;        //!< the id reference trajectory object
```

Then, the trajectory is initialized, with a delta of Rs per second

```
  // initialize the Id reference trajectory
  trajHandle_Id = TRAJ_init(&traj_Id,sizeof(traj_Id));

  // configure the Id reference trajectory
  TRAJ_setTargetValue(trajHandle_Id,_IQ(0.0));
  TRAJ_setIntValue(trajHandle_Id,_IQ(0.0));
  TRAJ_setMinValue(trajHandle_Id,_IQ(-USER_MOTOR_MAX_CURRENT /
                                    USER_IQ_FULL_SCALE_CURRENT_A));
  TRAJ_setMaxValue(trajHandle_Id,_IQ(USER_MOTOR_MAX_CURRENT /
```

```
                                   USER_IQ_FULL_SCALE_CURRENT_A));
  TRAJ_setMaxDelta(trajHandle_Id,_IQ(USER_MOTOR_RES_EST_CURRENT /
                                   USER_IQ_FULL_SCALE_CURRENT_A /
                                   USER_ISR_FREQ_Hz));
```

In the background loop, the Rs recalculation enable flag is checked, so that if this is enabled, the estimator is placed into the proper state by enabling this internal flag, and then running the update state function:

```
        else if(gMotorVars.Flag_enableRsRecalc)
          {
            // set angle to zero
            EST_setAngle_pu(estHandle,_IQ(0.0));

            // enable or disable Rs recalculation
            EST_setFlag_enableRsRecalc(estHandle,true);

            // update estimator state
            EST_updateState(estHandle,0);

            #ifdef FAST_ROM_V1p6
              // call this function to fix 1p6
              softwareUpdate1p6(estHandle);
            #endif

            // enable the PWM
            HAL_enablePwm(halHandle);

            // set trajectory target for speed reference
            TRAJ_setTargetValue(trajHandle_spd,_IQ(0.0));

            // set trajectory target for Id reference
            TRAJ_setTargetValue(trajHandle_Id,_IQ(USER_MOTOR_RES_EST_CURRENT /
                                                  USER_IQ_FULL_SCALE_CURRENT_A));

            // if done with Rs recalculation, disable flag
            if(EST_getState(estHandle) == EST_State_OnLine)
              gMotorVars.Flag_enableRsRecalc = false;
          }
```

As shown above, the speed trajectory is set to zero, and the Id trajectory is set to the motor resistance estimation current specified in user.h. Also notice from above that as soon as the state of the estimator changes to OnLine, the Rs Recalculation is disabled.

This trajectory is run and updated in the ISR:

```
  // run a trajectory for Id reference
  TRAJ_run(trajHandle_Id);
```

In the ISR, the additional code when doing Rs Recalculation is to set the Iq reference to zero, as follows:

```
    // set Iq reference to zero when doing Rs recalculation
    if(gMotorVars.Flag_enableRsRecalc) gIdq_ref_pu.value[1] = _IQ(0.0);
```

Also in the ISR, to avoid moving around the trigger while recalculating Rs, the following line is needed:

```
  // run function to set next trigger
  if(!gMotorVars.Flag_enableRsRecalc) runSetTrigger();
```

Speed and Id ramps for smooth reference changes

This feature allows changing the speed references and Id references smoothly with a ramp instead of steps. This is very useful in speed reference changes to avoid overcurrents or overvoltages if the motor is decelerated by a step in speed reference. Also, Id ramps are useful when changing the reference, or when driving an induction motor where the magnetizing current is fed through Id reference. The following code changes allow ramps in the speed reference. Id trajectory is the same as the one used for Rs recalculation, with the exception that the target value of Id trajectory is user provided when not doing Rs recalculation. For the speed trajectory, this is the additional code:

```
TRAJ_Handle     trajHandle_spd;  //!< the handle for the speed reference trajectory
TRAJ_Obj        traj_spd;        //!< the speed reference trajectory object
```

Initialization of the speed trajectory:

```
  // initialize the speed reference trajectory
  trajHandle_spd = TRAJ_init(&traj_spd,sizeof(traj_spd));

  // configure the speed reference trajectory
  TRAJ_setTargetValue(trajHandle_spd,_IQ(0.0));
  TRAJ_setIntValue(trajHandle_spd,_IQ(0.0));
  TRAJ_setMinValue(trajHandle_spd,_IQ(-1.0));
  TRAJ_setMaxValue(trajHandle_spd,_IQ(1.0));
  TRAJ_setMaxDelta(trajHandle_spd,_IQ(USER_MAX_ACCEL_Hzps /
                                      USER_IQ_FULL_SCALE_FREQ_Hz /
                                      USER_ISR_FREQ_Hz));
```

In the background loop, the speed trajectory target value is set from the user global variable:

```
            // set trajectory target for speed reference
            TRAJ_setTargetValue(trajHandle_spd,_IQmpy(gMotorVars.SpeedRef_krpm,
                                                      gSpeed_krpm_to_pu_sf));
```

In the ISR, the speed trajectory is updated:

```
  // run a trajectory for speed reference
  TRAJ_run(trajHandle_spd);
```

Also in the ISR, when using the speed reference for the speed controller, the value is pulled from the intermediate value of the trajectory:

```
          // run speed controller
```

```
        PID_run_spd(pidHandle[0],
                    TRAJ_getIntValue(trajHandle_spd),
                    speed_pu,
                    &(gIdq_ref_pu.value[1]));
```

Both motors ACIM and PM/IPM are supported

In order to support ACIM, magnetizing current needs to be handled smoothly through a ramp (or trajectory as this is called in MotorWare). The Id reference trajectory configuration for ACIM support will have the rated magnetizing current as a set point as shown here:

```
      // configure the Id reference trajectory
      TRAJ_setTargetValue(trajHandle_Id,_IQ(0.0));
      TRAJ_setIntValue(trajHandle_Id,_IQ(0.0));
      TRAJ_setMinValue(trajHandle_Id,_IQ(0.0));
      TRAJ_setMaxValue(trajHandle_Id,_IQ(USER_MOTOR_MAGNETIZING_CURRENT /
                                         USER_IQ_FULL_SCALE_CURRENT_A));
      TRAJ_setMaxDelta(trajHandle_Id,_IQ(USER_MOTOR_MAGNETIZING_CURRENT /
                                         USER_IQ_FULL_SCALE_CURRENT_A /
                                         USER_ISR_FREQ_Hz));
```

As can be seen, the minimum value is set to zero to be able to start the motor with a ramp in Id starting from zero.

PowerWarp

This mode only applies to ACIM motors. In order to allow this feature to work without a controller object, a new library was created and added to MotorWare: EST_runPowerWarp.lib. This library runs PowerWarp algorithm with simple input and output parameters, and an example is done here in lab 11a. The following code listing shows how the background loop adds Id trajectory configuration when PowerWarp is enabled. When it is enabled, PowerWarp is run by calling: EST_runPowerWarp, and the trajectory is configured in such a way that limits the minimum Id reference to 30% of the rated magnetizing current, and it also limits the rate of change, so the Id reference ramp changes slowly. When disabled, the Id trajectory is configured with default values, so Id reference target is set to the rated magnetizing current.

```
if(gMotorVars.Flag_enablePowerWarp)
  {
    _iq Id_target_pw_pu = EST_runPowerWarp(estHandle,
                          TRAJ_getIntValue(trajHandle_Id),
                          gIdq_pu.value[1]);

    TRAJ_setTargetValue(trajHandle_Id,Id_target_pw_pu);
    TRAJ_setMinValue(trajHandle_Id,_IQ(USER_MOTOR_MAGNETIZING_CURRENT * 0.3 /
                                       USER_IQ_FULL_SCALE_CURRENT_A));
    TRAJ_setMaxDelta(trajHandle_Id,_IQ(USER_MOTOR_MAGNETIZING_CURRENT * 0.3 /
                                       USER_IQ_FULL_SCALE_CURRENT_A /
                                       USER_ISR_FREQ_Hz));
  }
else
  {
    // set trajectory target for Id reference
```

```
   TRAJ_setTargetValue(trajHandle_Id,_IQ(USER_MOTOR_MAGNETIZING_CURRENT /
                                       USER_IQ_FULL_SCALE_CURRENT_A));
   TRAJ_setMinValue(trajHandle_Id,_IQ(0.0));
   TRAJ_setMaxDelta(trajHandle_Id,_IQ(USER_MOTOR_MAGNETIZING_CURRENT /
                                    USER_IQ_FULL_SCALE_CURRENT_A /
                                    USER_ISR_FREQ_Hz));
 }
```

Rs OnLine

There are a few changes when supporting Rs OnLine. To start up with, we need configuration variables declared as global variables, so we can change them through a watch window. Those variables are:

```
volatile bool gFlag_enableRsOnLine = false;
volatile bool gFlag_updateRs = false;
volatile _iq gRsOnLineFreq_Hz = _IQ(0.2);
volatile _iq gRsOnLineId_mag_A = _IQ(0.5);
volatile _iq gRsOnLinePole_Hz = _IQ(0.2);
```

- *gFlag_enableRsOnLine* is an enable flag of RsOnLine
- *gFlag_updateRs* enables updates from the estimated RsOnLine to the actual internal Rs used by the estimator. It is recommended to change the update flag to true only when the RsOnLine estimator (*gMotorVars.RsOnLine_Ohm*) has settled.
- *gRsOnLineFreq_Hz* defines the frequency of the slowly rotating angle, as explained in user's guide: SPRUHJ1F, Chapter 15.
- *gRsOnLineId_mag_A* is the current amplitude to be injected for RsOnLine as explained in above user's guide.
- *gRsOnLinePole_Hz* is the cutoff frequency of the RsOnLine low pass filters as explained in the user's guide.

The following initialization of RsOnLine is required, so that the user provided configuration is scaled to per unit values, and the correct filter coefficients are calculated:

```
  // configure RsOnLine
  EST_setFlag_enableRsOnLine(estHandle,gFlag_enableRsOnLine);
  EST_setFlag_updateRs(estHandle,gFlag_updateRs);
  EST_setRsOnLineAngleDelta_pu(estHandle,_IQmpy(gRsOnLineFreq_Hz,
                                         _IQ(1.0/USER_ISR_FREQ_Hz)));
  EST_setRsOnLineId_mag_pu(estHandle,_IQmpy(gRsOnLineId_mag_A,
                                  _IQ(1.0/USER_IQ_FULL_SCALE_CURRENT_A)));

  // Calculate coefficients for all filters
  {
    _iq b0 = _IQmpy(gRsOnLinePole_Hz, _IQ(1.0/USER_ISR_FREQ_Hz));
    _iq a1 = b0 - _IQ(1.0);
    EST_setRsOnLineFilterParams(estHandle,EST_RsOnLineFilterType_Current,
                                b0,a1,_IQ(0.0),b0,a1,_IQ(0.0));
    EST_setRsOnLineFilterParams(estHandle,EST_RsOnLineFilterType_Voltage,
                                b0,a1,_IQ(0.0),b0,a1,_IQ(0.0));
  }
```

In the background loop, we enable or disable RsOnLine according to user's input through the watch window. We also change configuration on the fly, allowing users to experiment with different RsOnLine settings:

```
// enable or disable RsOnLine
EST_setFlag_enableRsOnLine(estHandle,gFlag_enableRsOnLine);

// set slow rotating frequency for RsOnLine
EST_setRsOnLineAngleDelta_pu(estHandle,_IQmpy(gRsOnLineFreq_Hz,
                                    _IQ(1.0/USER_ISR_FREQ_Hz)));

// set current amplitude for RsOnLine
EST_setRsOnLineId_mag_pu(estHandle,_IQmpy(gRsOnLineId_mag_A,
                              _IQ(1.0/USER_IQ_FULL_SCALE_CURRENT_A)));

// set flag that updates Rs from RsOnLine value
EST_setFlag_updateRs(estHandle,gFlag_updateRs);

// clear Id for RsOnLine if disabled
if(!gFlag_enableRsOnLine) EST_setRsOnLineId_pu(estHandle,_IQ(0.0));
```

Finally, in the ISR, in order to output the commanded value by the RsOnLine algorithm, this value is taken from the estimator, and it is added to the trajectory Id reference.

```
// get the reference value from the trajectory module
refValue = TRAJ_getIntValue(trajHandle_Id) + EST_getRsOnLineId_pu(estHandle);
```

Overmodulation

Similar to lab 10a, some especial considerations and needed for overmodulation. Please refer to lab 10a description to know the details of overmodulation.

Field Weakening

Similar to lab 9, a simple automatic field weakening example is added to lab 11a. Please refer to lab 9 description to know the details of field weakening.

1/Vbus Compensation

This feature is also included in lab11, as this is always required to amplify output voltage depending on how much Vbus has dropped. This is done by reading the 1/Vbus calculation done inside the estimator, and compensating Vab_pu with that value. Then SVM is executed using the 1/Vbus compensated values:

```
// run the space Vector Generator (SVGEN) module
oneOverDcBus = EST_getOneOverDcBus_pu(estHandle);
Vab_pu.value[0] = _IQmpy(Vab_pu.value[0],oneOverDcBus);
Vab_pu.value[1] = _IQmpy(Vab_pu.value[1],oneOverDcBus);
SVGEN_run(svgenHandle,&Vab_pu,&(gPwmData.Tabc));
```

CPU Usage Calculation

This feature allows measuring percentage of CPU 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. First, we declare object and handle for this CPU_USAGE measurement as follows:

```
CPU_USAGE_Handle  cpu_usageHandle;
CPU_USAGE_Obj     cpu_usage;
```

Then, we initialize this module, so that timers are configured, and global variables are zeroed out.

```
  // initialize the CPU usage module
  cpu_usageHandle = CPU_USAGE_init(&cpu_usage,sizeof(cpu_usage));
  CPU_USAGE_setParams(cpu_usageHandle,
                     (uint32_t)USER_SYSTEM_FREQ_MHz * 1000000, // timer period, cnts
                     (uint32_t)USER_ISR_FREQ_Hz);  // average over 1 second of ISRs
```

In order to measure the cycles that it takes to execute the ISR with this module, we add this at the very beginning of the ISR:

```
  // read the timer 1 value and update the CPU usage module
  timer1Cnt = HAL_readTimerCnt(halHandle,1);
  CPU_USAGE_updateCnts(cpu_usageHandle,timer1Cnt);
```

And we also call this at the end to get total number of cycles:

```
  // read the timer 1 value and update the CPU usage module
  timer1Cnt = HAL_readTimerCnt(halHandle,1);
  CPU_USAGE_updateCnts(cpu_usageHandle,timer1Cnt);

  // run the CPU usage module
  CPU_USAGE_run(cpu_usageHandle);
```

In order to report CPU usage in percentage, we call this function in the background loop, and will display % from 0 to 100.

```
void updateCPUusage(void)
{
  uint32_t minDeltaCntObserved = CPU_USAGE_getMinDeltaCntObserved(cpu_usageHandle);
  uint32_t avgDeltaCntObserved = CPU_USAGE_getAvgDeltaCntObserved(cpu_usageHandle);
  uint32_t maxDeltaCntObserved = CPU_USAGE_getMaxDeltaCntObserved(cpu_usageHandle);
  uint16_t pwmPeriod = HAL_readPwmPeriod(halHandle,PWM_Number_1);
  float_t  cpu_usage_den = (float_t)pwmPeriod *
                           (float_t)USER_NUM_PWM_TICKS_PER_ISR_TICK * 2.0;

  // calculate the minimum cpu usage percentage
  gCpuUsagePercentageMin = (float_t)minDeltaCntObserved / cpu_usage_den * 100.0;

  // calculate the average cpu usage percentage
  gCpuUsagePercentageAvg = (float_t)avgDeltaCntObserved / cpu_usage_den * 100.0;
```

```
  // calculate the maximum cpu usage percentage
  gCpuUsagePercentageMax = (float_t)maxDeltaCntObserved / cpu_usage_den * 100.0;

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

This calculation involved an average of CPU usage in one second. If users want to reset the value of this average and restart the accumulation of cycles for this average calculation, set this flag to 1:

```
cpu_usage. flag_resetStats = 1
```

Lab Procedure

Step 1.

Open lab 11a, build and load project, and load variables in the .js file (refer to previous projects for instructions on how to do this).

Step 2.

Run the application, and the first thing to do is to calculate offsets. So set gMotorVars.Flag_enableOffsetcalc = 1. Wait until the variable changes back to 0 automatically

(x)= gMotorVars.Flag_enableOffsetc	0 (Decimal)

At that point, all offsets will be calculated and store in these variables:

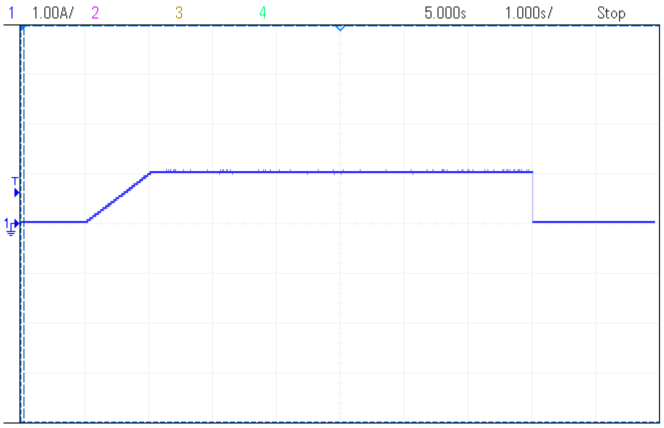
gOffsets_I_pu	{...}
value	0x00008852@Data
(x)= [0]	1.211008966 (Q-Value(24))
(x)= [1]	1.209482312 (Q-Value(24))
(x)= [2]	1.209160745 (Q-Value(24))
gOffsets_V_pu	{...}
value	0x00008858@Data
(x)= [0]	0.5112561584 (Q-Value(24))
(x)= [1]	0.5103011131 (Q-Value(24))
(x)= [2]	0.5095147491 (Q-Value(24))

Step 3.

Enable Rs recalculation by setting gMotorVars.Flag_enableRsRecalc = 1. Wait until the variable changes back to 0 automatically

(x)= gMotorVars.Flag_enableRsRecalc	0 (Decimal)

Also notice that while doing Rs recalculation, DC currents will be injected to the motor windings as shown here:



Rs recalculation in lab 11a only recalculates the stator resistance, and does not spin the motor when Rs recalculation is complete

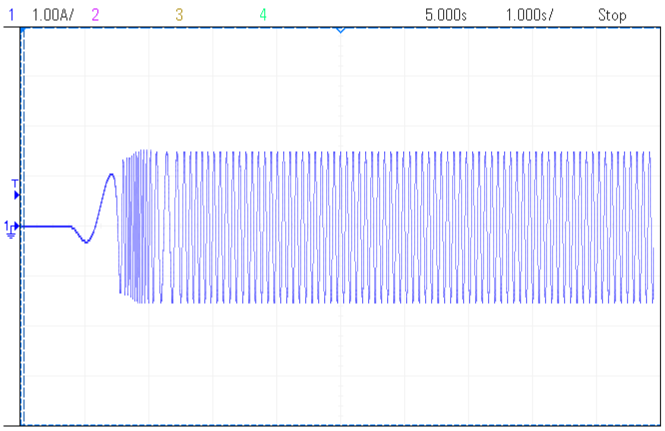
Step 4.

Once offsets are Rs are calibrated, the motor can be run by setting gMotorVars.Flag_Run_Identify = 1.

Step 5.

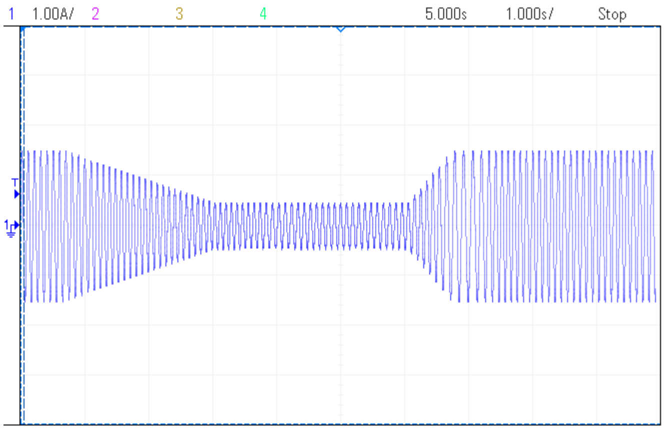
Change speed reference and speed controller gains. This step is used to experiment with the speed controller, by changing the reference with this variable: gMotorVars.SpeedRef_krpm, and speed controller gains: pid[0].Kp and pid[0].Ki.

If the motor is an induction motor, you will see that when starting up the motor, a magnetizing current will build up as shown in the following phase current scope plot:



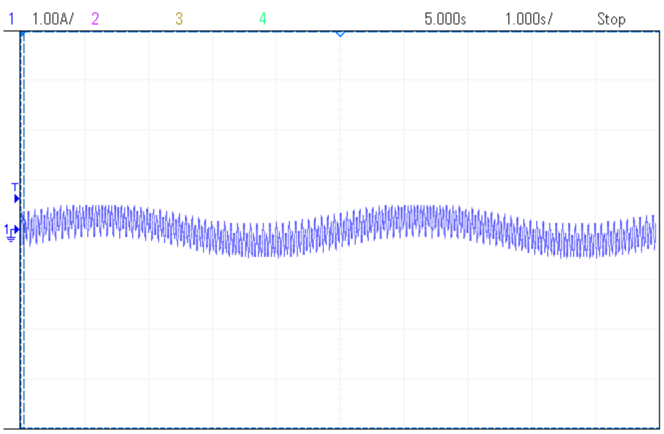
Step 6.

Enable and disable PowerWarp. This feature only applies to ACIM. By setting and clearing this flag, users can enable or disable PowerWarp: gMotorVars.Flag_enablePowerWarp. The following current waveform from one of the motor phases shows a transition when PowerWarp is enabled (for energy efficiency improvements), and then disabled (for faster load changes response):

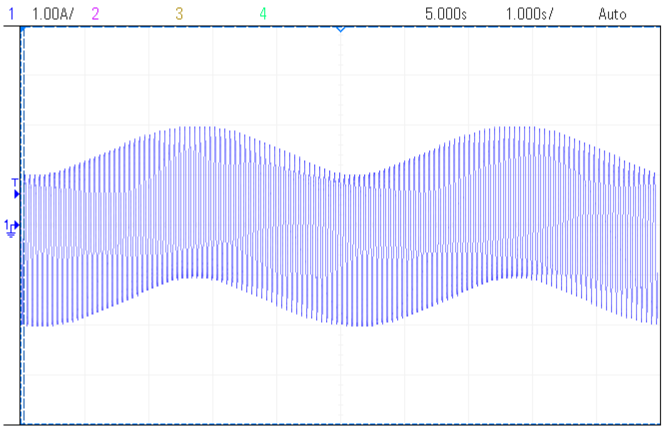


Step 7.

Enable Rs Online. By setting and clearing this flag, users can turn Rs Online on or off: gFlag_enableRsOnLine. When Rs Online is enabled, Id current is injected, producing a current waveform as follows:



If there is already Id reference provided by the user, or the motor is an ACIM which already has a magnetizing current component, Rs Online will still work, although the AC waveform would look like this for an ACIM:



If both Rs and Rs Online values are monitored, notice how they are different because the update flag has not been set yet:

(x)= gMotorVars.Rs_Ohm	0.3895107
(x)= gMotorVars.RsOnLine_Ohm	0.3898349

If you set the update flag: gFlag_updateRs = 1, then Rs Online value will be updated into Rs

(x)= gMotorVars.Rs_Ohm	0.3913409
(x)= gMotorVars.RsOnLine_Ohm	0.3913409

Now you can turn off Rs online: gFlag_enableRsOnLine = 0, gFlag_updateRs = 0

Step 8.

Lower speed controller gains, since for step 8 we will go into overmodulation, and it is better to have soft speed controller gains:

(x)= pid[0].Kp	1.0 (Q-Value(24))
(x)= pid[0].Ki	0.009999990463 (Q-Value(24))

Now speed up the motor by changing the speed reference: gMotorVars.SpeedRef_krpm

In the case of the booster pack with DRV8305, the Anaheim motor, 24V input and with a maxed out output vector of:

(x)= gMotorVars.Vs	0.6666666269 (Q-Value(24))

The maximum speed reached is:

(x)= gMotorVars.Speed_krpm	6.263718128 (Q-Value(24))

If purely sinusoidal waveforms are generated (so that there is no overmodulation), then change this parameter in user.h:

```
#define USER_MAX_VS_MAG_PU          (0.5)
```

And try again. This value won't require current reconstruction or recalculating the trigger point. But the maximum output vector would be limited to:

(x)= gMotorVars.Vs	0.5 (Q-Value(24))

And the maximum speed with this output vector would be:

(x)= gMotorVars.Speed_krpm	5.050895393 (Q-Value(24))

So essentially we gain more than 1,200 RPM with overmodulation in this particular case.

Step 9.

In Step 9 we will experiment with field weakening. This lab only applies to PMSM and IPM motors, since for ACIM PowerWarp can be used as a form of field weakening to reach higher speeds.

Lower speed controller gains, since high speed experimenting usually works better with software speed controller gains.

(x)= pid[0].Kp	1.0 (Q-Value(24))
(x)= pid[0].Ki	0.009999990463 (Q-Value(24))

Speed up the motor and enable field weakening: gMotorVars.Flag_enableFieldWeakening = 1

The output voltage variable Vs will start growing, until it reaches VsRef:

(x)= gMotorVars.IdRef_A	0.0 (Q-Value(24))
(x)= gMotorVars.Vs	0.3982147574 (Q-Value(24))
(x)= gMotorVars.VsRef	0.5333333611 (Q-Value(24))

When the commanded speed is increased more, and Vs is forced to be greater than VsRef, then the field weakening algorithm will start producing a negative Id reference in order to keep Vs <= VsRef

(x)= gMotorVars.IdRef_A	-0.8010864258 (Q-Value(24))
(x)= gMotorVars.Vs	0.5277536511 (Q-Value(24))
(x)= gMotorVars.VsRef	0.5333333611 (Q-Value(24))

Conclusion

In addition to the conclusions of lab 11, this lab adds the main features of InstaSPIN, and without high lever abstraction modules like the CTRL object, users can modify this lab to accommodate their own requirements, so that a project can be productize.

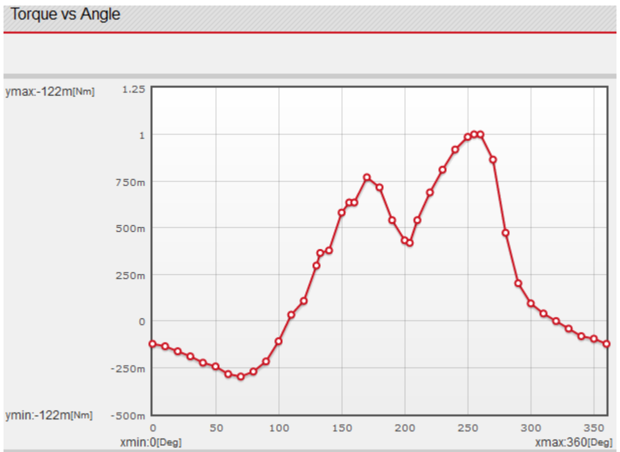
Lab 11b – Vibration Compensation Example

Abstract

In applications where the load is dependent on the mechanical angle such as air conditioning compressors, it is desirable to have a control loop that compensates for the known load. TI created a new library that implements an algorithm that compensates load that causes vibration. This lab shows an example on how to use the vibration compensation library.

Introduction

In lab 11a we showed a feature rich example without a controller module. Lab 11b builds on top of lab 11a, adding a vibration compensation module. In order to understand the problem, let's take a look at the load profile of a typical compressor application, where load is dependent of the mechanical angle, as the piston compresses and decompresses.



As can be seen, in one mechanical cycle, the load goes from a negative load (load actually helps with motor motion) up to maximum load, in this case of 1 Nm. This kind of load profile is extremely challenging for a conventional PI speed controller, since it would require a very responsive controller, that if it is only based on feedback, it would be very oscillatory or unstable. In the detailed section of this lab, we will talk about the vibration module, and how it is used to compensate for these kinds of loads.

Prerequisites

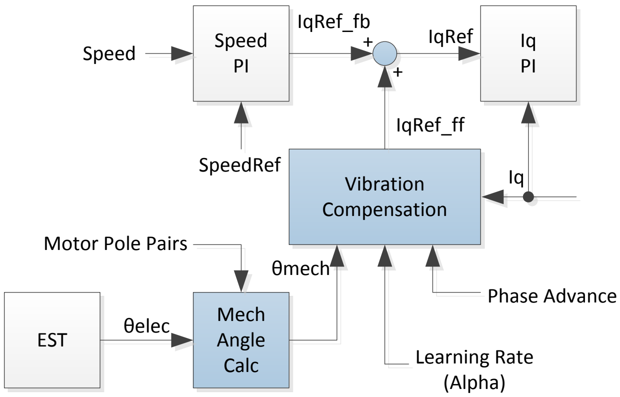
It assumes knowledge of proj_lab11a.

Objectives Learned

The objective of this lab is to learn how to interface the vibration compensation library.

Detailed Description

What we have created is an algorithm that learns the load profile as the motor runs, and as the speed controller tries to correct for these load changes, and once the load is learned, the algorithm is used to extract load information relative to the mechanical angle, and uses that information as a feedforward in the speed controller. The blue blocks are added to the FOC system, to allow adding a feedforward term to the speed controller, in the form of a summing point to the output generated by the speed controller.



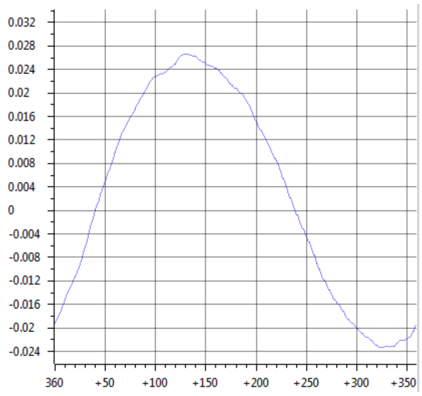
Starting from the lower left, the first input that is required by the vibration compensation module is the mechanical angle. This is calculated based on the electrical angle and the number of pole pairs. It is not required for the mechanical angle to be synchronized with the electrical angle. In other words, the zero of the mechanical angle, physically, doesn't need to be the zero of the electrical angle. This is because the vibration compensation module will learn the load according to the mechanical angle provided, independently of what the mechanical angle is compared to the physical position of the shaft.

Then the vibration compensation module is implemented. This module requires the mechanical angle, a learning rate, which is essentially how fast (less noise immune) or how slow (more noise immune) the load learning happens, and a phase advance, which is how the learned load will be accessed with respect to the mechanical angle. If zero phase advance, the loaded value on IqRef_ff will correspond to the provided mechanical angle. If phase advance is 10, the loaded value on IqRef_ff will correspond to the mechanical angle plus 10 mechanical degrees (in a scale from 0 to 360 degrees).

Then the summation point in between the speed controller and the Iq controller. This is where the output of the vibration compensation module is used, to help the speed controller with this term. This technique is also known as feedforward, since the load is known in advance, according to the mechanical angle provided.

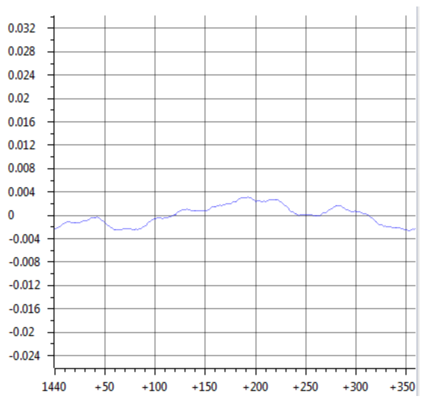
Once the load has been learned by the vibration compensation module, the speed controller will correct for transients in load change, that don't relate to the natural mechanical load vs. mechanical angle, which is already compensated by the vibration compensation module.

In order to illustrate how the vibration compensation module helps, let's take a look at the following plot, where we show the output of the speed controller with vibration compensation disabled. It is obvious that the speed controller gains need to be high in order to track the load changes as the motor spins every cycle.



The Y axis is Iq reference from the speed controller in per units, and X axis is mechanical degrees, from 0 to 360, so it shows a complete mechanical cycle. It can be seen that the speed controller needs high bandwidth to simply catch up with a load change during each mechanical cycle.

Once the vibration compensation module has been enabled, the output of the speed controller is:



The rest of the Iq reference is being provided by the vibration compensation algorithm.

Now let's take a look at the software. Taking lab 11a as a starting point, the following sections will describe each additional features of lab 11b in detail.

Variables Needed for Vibration Compensation

- vib_comp_reserved[]. This buffer is required to reserve memory used by the vibration compensation module.
- gSpeed_fbk_out[]. This buffer is used to store the output of the speed controller vs mechanical angle
- gSpeed_array_index. Index used for gSpeed_fbk_out[] buffer
- gSpeed_max_pu. Variable used to store the maximum speed for speed variation calculation
- gSpeed_min_pu. Variable used to store the minimum speed for speed variation calculation
- gSpeed_delta_krpm. Variable used to store the speed variation calculation
- gFlag_speedStatsReset. This flag is used to reset the speed variation calculation
- gAbsAngle_elec_pu. Variable used for electrical angle from _IQ(0.0) to _IQ(1.0)
- gAbsAngle_mech_pu. Variable used for mechanical angle from _IQ(0.0) to _IQ(1.0)

- gAngle_mech_poles. Variable used for mechanical angle from _IQ(0.0) to _IQ(USER_MOTOR_NUM_POLE_PAIRS)
- gAngle_z1_pu. Variable used for electrical angle in previous sample
- gAlpha. Learning rate of the vibration compensation module from _IQ(0.0) to _IQ(1.0)
- gAdvIndexDelta. Phase advance of the vibration compensation module from 0 to 360
- gFlag_enableOutput. Flag that enables the output from the vibration compensation module to the Iq reference
- gFlag_enableUpdates. Flag that enables learning of the load, and storing the load profile internally in the vibration compensation module
- gFlag_resetVibComp. Flag that resets the learned load curve

Initializing the Vibration Compensation Module

The following code snippet shows how this module is initialized and configured with default values

```
// initialize the handle for vibration compensation
vib_compHandle = VIB_COMP_init(&vib_comp_reserved, VIB_COMP_getSizeOfObject());

VIB_COMP_setParams(vib_compHandle, gAlpha, gAdvIndexDelta);

VIB_COMP_reset(vib_compHandle);

for(cnt=0;cnt<360;cnt++)
  {
    gSpeed_fbk_out[cnt] = _IQ(0.0);
  }
```

Updating Vibration Compensation Variables in the Background Loop

The following code shows how the internal variable and flags are updated in the background loop (outside of the ISR) in case users change these values in the watch window:

```
VIB_COMP_setAlpha(vib_compHandle, gAlpha);

VIB_COMP_setAdvIndexDelta(vib_compHandle, gAdvIndexDelta);

VIB_COMP_setFlag_enableOutput(vib_compHandle, gFlag_enableOutput);

VIB_COMP_setFlag_enableUpdates(vib_compHandle, gFlag_enableUpdates);

if(gFlag_resetVibComp)
  {
    gFlag_resetVibComp = false;
    VIB_COMP_reset(vib_compHandle);
  }

if(gFlag_speedStatsReset)
  {
    gFlag_speedStatsReset = false;
    gSpeed_max_pu = _IQ(0.0);
    gSpeed_min_pu = _IQ(1.0);
```

```
            }
```

Starting from the top, gAlpha is used as the learning speed. The higher this value (with a maximum of _IQ(1.0)) the slowest it learns the algorithm. A high value is desirable though, since it provides noise immunity.

The purpose of the gAdvIndexDelta variable is to advance the output waveform into the future by a little bit so that the resulting current will be very close to the desired value by the time the mechanical angle reaches that point. A typical value of 10 is recommended, but ultimately needs to be fine-tuned by the user.

Calculating the Mechanical Angle in the ISR

As soon as the angle is estimated inside the ISR, we call these two new functions, which are public and source code available in the project file itself, to calculate the mechanical angle. The first function call removes the sign of the electrical angle, to have an absolute value from _IQ(0.0) to _IQ(1.0) with any direction of rotation. The second function call is used to convert this angle into mechanical angle.

```
  // calculate absolute electrical angle
  gAbsAngle_elec_pu = getAbsElecAngle(angle_pu);

  // calculate absolute mechanical angle
  gAbsAngle_mech_pu = getAbsMechAngle(&gAngle_mech_poles,
                                      &gAngle_z1_pu,
                                      gAbsAngle_elec_pu);
```

Then the vibration compensation algorithm is called and its output is added to the Iq reference

```
      // get the Iq reference value plus vibration compensation
      refValue = gIdq_ref_pu.value[1] +
                 VIB_COMP_run(vib_compHandle, gAbsAngle_mech_pu, gIdq_pu.value[1]);
```

Lab Procedure

Step 1.

Repeat steps 1, 2, 3 and 4 from lab 11a.

Step 2.

Change speed reference (gMotorVars.SpeedRef_krpm) and speed controller gains (pid[0].Kp and pid[0].Ki). This step is used to take the motor and load to where the motor vibrates due to the pulsating load. For vibration compensation to work better, increase the values of the speed controller gains. Make sure the speed controller is still stable though.

Step 3.

Get an initial assessment of the speed variation, by setting this bit: gFlag_speedStatsReset = 1. This will force a new calculation of speed variation. In the tested example we get the following speed variation after resetting that flag:

(x)= gSpeed_delta_krpm	0.08618044853 (Q-Value(24))

So before enabling vibration compensation, our speed variation is about 86 RPM.

Step 4.

Now enable the output by setting this flag: gFlag_enableOutput = 1.Then let it run for a 5 to 10 seconds, and then get the new speed variation by setting this bit: gFlag_speedStatsReset = 1. In our example, now the speed variation is about 12 RPM:

(x)= gSpeed_delta_krpm	0.01241111755 (Q-Value(24))

Step 5.

If the vibration was not reduced, try increasing the speed controller gains. Also try increasing the learning speed of the vibration compensation algorithm by decreasing the value of gAlpha in decrements of _IQ(0.02), so try: _IQ(0.99), _IQ(0.97), _IQ(0.95), etc. each time you change gAlpha, let it run for a few seconds and get a reading of the speed variation by resetting that calculation: gFlag_speedStatsReset = 1.

Conclusion

In addition to the conclusions of lab 11a, this lab adds vibration compensation. This new feature allows having a traditional PI speed controller, and by simply adding this algorithm, speed variations that cause motor vibration are significantly reduced. Another important advantage of this vibration compensation algorithm is that the mechanical angle does not need to be synchronized to any physical mechanical position.

Lab 11d – Dual Motor Sensorless Velocity InstaSPIN-FOC

Abstract

A writeup for dual motor control using InstaSPIN-FOC is provided in the proj_lab10d writeup.

Writeups will not be provided for InstaSPIN-FOC proj_lab11d, and neither for InstaSPIN-MOTION proj_lab6e, proj_lab12c nor proj_lab13f. Please review the code to understand the technique and important variables employed in these labs.

Proj_lab11d is a valid dual motor control project for InstaSPIN-FOC based on the simplified structure of proj_lab11. Labs 6e, 12c, and 13f are InstaSPIN-MOTION projects that build upon the lab 11d code base.

Please note that proj_lab11d does not include a trajectory module; as such, there is no acceleration variable, and any changes the speed reference are applied as a step input. Use caution. InstaSPIN-MOTION labs 6e, 12c, and 13f do not have this limitation.

Lab 11e – Hall Start with Transition to FAST

Abstract

To improve the torque drive performance of PMSM/BLDC motor control for zero speed start up, an example is shown here by using Hall sensors for zero speed start-up and InstaSPIN-FOC for high speed running. The two driving modes switch from one to another smoothly at a speed point specified by users based on the according application needs. It is assumed the system is expected to run properly with speed closed loop control when a target speed is set higher than the switching speed point, because the slow speed feedback from Hall sensor will lead to a very limited bandwidth for speed closed loop and make the speed controller difficult to perform as expected at the low speed range.

Introduction

A sensor-and-sensorless hybrid control of PMSM motor is presented in Lab 11e. A postion Hall-sensor-based BLDC control is to seve as a motor starting algorithm to support zero to low speed running. InstaSPIN FOC sensorless control won't be implemented until a high speed running is reached. By BLDC control, a square wave current control in synchronous with the position information from Hall sensors is implemented. The position information from Hall sensors will be synced up with estimated angle output from InstaSPIN observer module to facilitate the transition from Hall sensor based mode to sensorless instaSPIN mode.

Prerequisites

It assumes knowledge of proj_lab11.

****PLEASE NOTE: If using TMDSHVMTRINSPIN kit for this lab you will need to change the pullup resistors for the hall inputs (R20, R21, R22) or (R17, R18, R19) to 1.0 kohm.**

Objectives Learned

The objective of this lab is to learn to implement the trapezoidal BLDC control in an InstaSPIN project and make a smooth switch between the sensored BLDC drive and the sensorless InstaSPIN-FOC drive.

Detailed Description

While starting up from zero speed, the motor is fed with current in square waveform for each phase winding, in synchronous with the position information from hall sensors, just as the conventional common BLDC motor drives do. This helps for applications which require full torque for motor start up or very low speed running, which is usually challenging for sensorless motor drive algorithms. The control mode switches to FOC control when motor runs into high speed mode. If the speed is running down to low speed zone from high, the control mode will switch back to BLDC control again. A Hysteresis control is implemented for the mode switching to avoid system chattering.

Main Run-Time loop (forever loop)

No changes have been made in the forever loop in this lab.

Main ISR

The new functions added for BLDC control using Hall sensor are listed as below:

HALLBLDC_Ctrl_Run(). This function implements BLDC control using Hall sensors
HALLBLDC_State_Check(). This function check Hall sensor state and calculate motor speed
HALLBLDC_Ctrl_Stop(). This function resets motor state
HALLBLDC_Ctrl_PwmSet(). This function sets PWM switching state

Project Files

There are no new project files.

Includes

There are no new includes.

Global Object and Variable Declarations

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

gHall_BLDC_Is_fdb_pu. Variable used to store the sense current of BLDC
gHall_BLDC_Is_ref_pu. Variable used for setting the reference current of BLDC
gHall_PwmDuty. Variable used to store the setting PWM duty of BLDC
gHall_speed_fdb_pu. Variable used to store the feedback speed of BLDC
gHall_speed_FastToBldc_low_pu. Variable used for setting the low speed of hysteresis control convert mode
gHall_speed_BldcToFast_high_pu. Variable used for setting the high speed of hysteresis control convert mode
gHall_State. Variable used to store the hall sensor state
gHall_PwmState. Variable used for setting the PWM switching state
gHall_BLDC_Flag_Is_fdb. Variable used for setting active phase current
gHall_PwmIndex[8]. This buffer used to the output pwm switch state of motor run vs hall sensor state.
Flag_EnableBldc. Flag enables BLDC mode
gHall_Flag_State_Change. Flag shows motor control mode change

CPU Usage Time Calculation

The mainISR() is time critical. When integrating your code into this ISR, it is important to verify that this ISR runs in real-time.

This feature allows measuring time of CPU 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.

Step 1, add “\sw\modules\cpu_time\src\32b\cpu_time.c” to project, and include "sw/modules/cpu_time/src/32b/cpu_time.h" in “main.h”.

Step 2, declare object and handle for this CPU_USAGE measurement as follows:

```
// define cpu_time object and handle for CPU usage time calculation
CPU_TIME_Handle  cpu_timeHandle;
CPU_TIME_Obj     cpu_time;
```

Step 3, initialize this module, so that timers are configured, and global variables are zeroed out.

```
// initialize the CPU usage module
cpu_timeHandle = CPU_TIME_init(&cpu_time,sizeof(cpu_time));
```

Step 4, in order to measure the cycles that it takes to execute the ISR with this module, we add this at the very 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(cpu_timeHandle,timer1Cnt);
```

Step 5, call this at the end to get total number of cycles:

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

We can monitor the maximum, minimum and average usage time of CPU from watch window as below, you need to ensure the value of “cpu_time.timer_delta_max” is less than “cpu_time.pwm_period”*2-100, to avoid ISR time overflow.

The function calculates a maximum, minimum and average of ISR CPU usage time. If users want to reset these values and restart the calculation, set this flag “cpu_time.flag_resetStatus” to 1.

PWMDAC

This module converts any s/w variables into the PWM signals in EPWMxA/B for C2000 MCU. Thus, it can be used to view the signal, represented by the variable, at the outputs of the PWMxA, PWMxB, pins through the external low-pass filters.

Step 1, declare object for this PWMDAC module.

```
// the PWMDAC variable
HAL_DacData_t gDacData;
```

Step 2, set the right offset and gain for each PWMDAC channel in proj_lab11e.c and hal.c

```
// set DAC parameters
HAL_setDacParameters(halHandle, &gDacData);


// set PWMDAC parameters for each channel to ensure the output waveform
void HAL_setDacParameters(HAL_Handle handle, HAL_DacData_t *pDacData)

{

  HAL_Obj *obj = (HAL_Obj *)handle;
  pDacData->PeriodMax = PWMDAC_getPeriod(obj->pwmDacHandle[PWMDAC_Number_1]);
  pDacData->offset[0] = _IQ(0.5);
  pDacData->offset[1] = _IQ(0.5);
  pDacData->offset[2] = _IQ(0.5);
  pDacData->offset[3] = _IQ(0.0);

  pDacData->gain[0] = _IQ(1.0);
  pDacData->gain[1] = _IQ(1.0);
  pDacData->gain[2] = _IQ(1.0);
  pDacData->gain[3] = _IQ(1.0);

} // end of HAL_setDacParameters() function
```

Step 3, connect inputs of the PWMDAC module.

```
     // get the estimator angle and frequency values
    gMotorVars.angle_est_pu = angle_pu;
    gMotorVars.speed_est_pu = speed_est_pu;

    // connect inputs of the PWMDAC module.
    gDacData.value[0] = gAdcData.I.value[0];
    gDacData.value[1] = gIdq_ref_pu.value[0];
    gDacData.value[2] = gMotorVars.speed_est_pu;
    gDacData.value[3] = gMotorVars.angle_est_pu;

    // run PwmDAC
   HAL_writeDacData(halHandle,&gDacData);
```

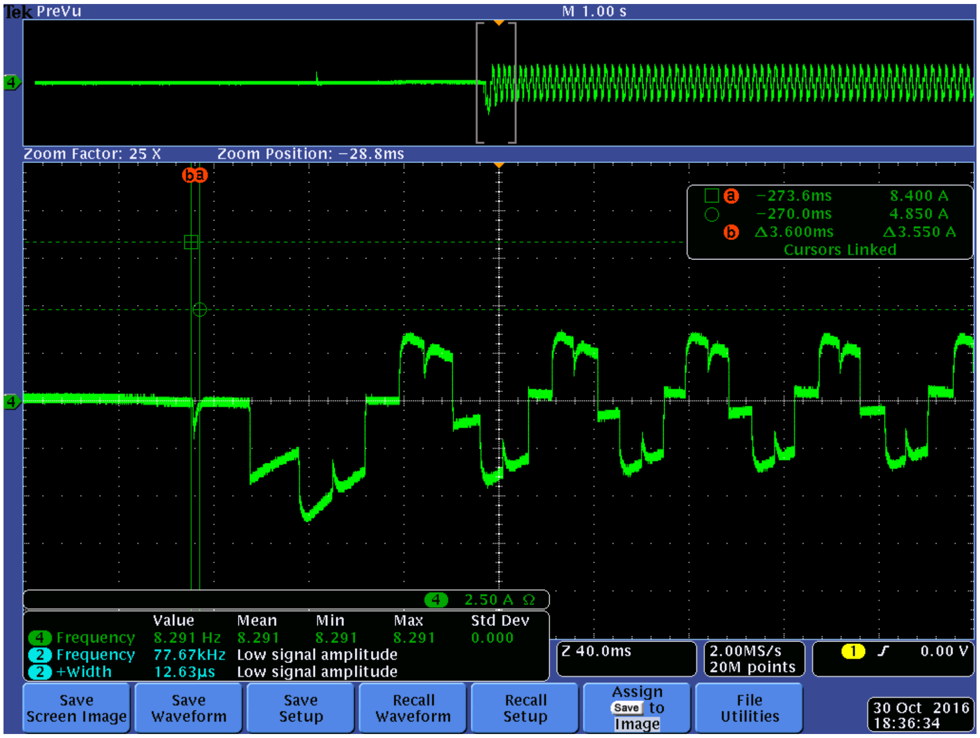


Figure 66: Hall startup from zero speed with full load

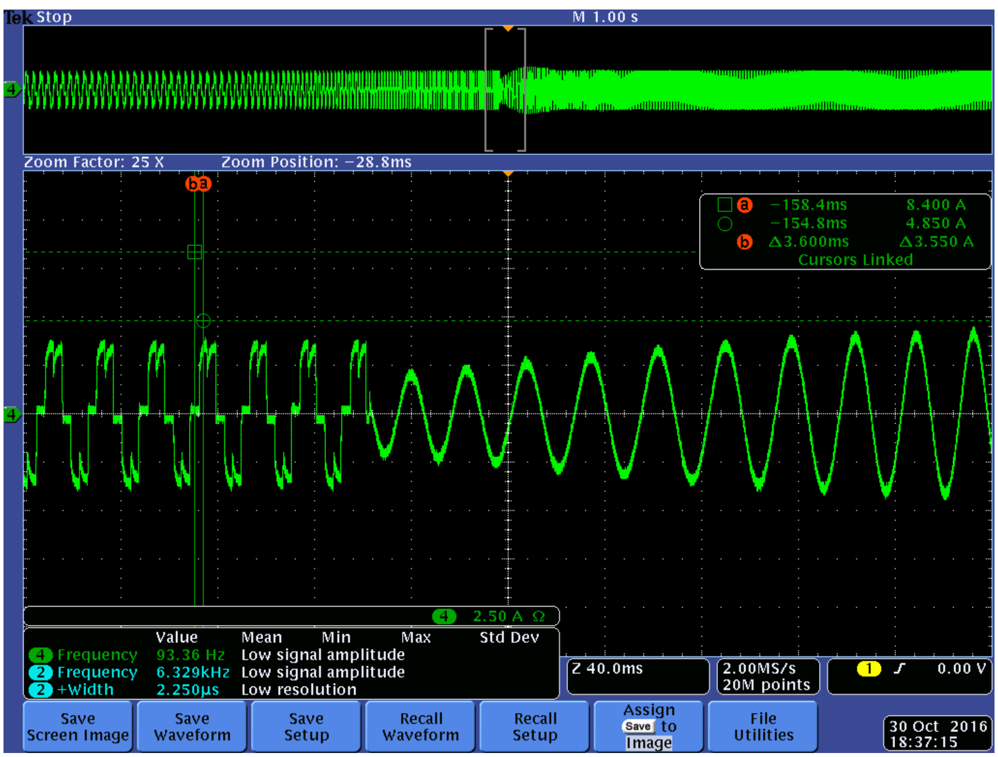


Figure 67: Hall sensor run switch to Fast sensorless run with full load

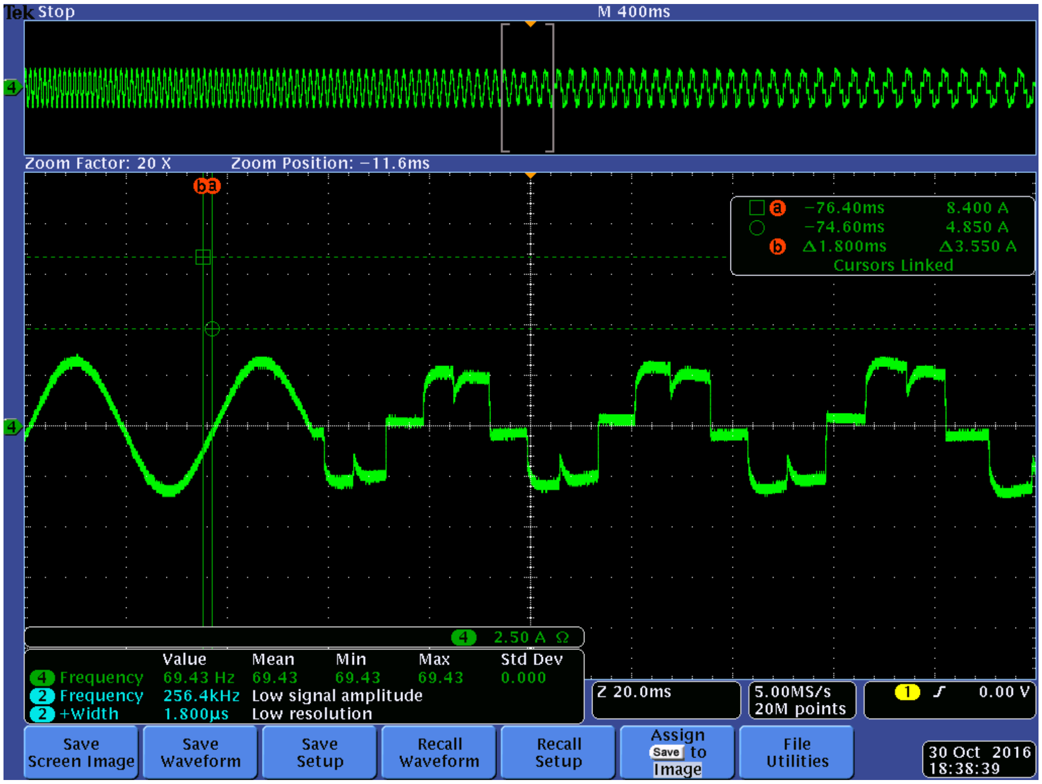


Figure 68: Fast sensorless run switch to Hall sensor run with full load

Lab Procedure

Step 1.

****PLEASE NOTE: If using TMDSHVMTRINSPIN kit for this lab you will need to change the pullup resistors for the hall inputs (R20, R21, R22) or (R17, R18, R19) to 1.0 kohm.**

If using **LAUNCHXL-F28069M** and **BOOSTXL-DRV8301** or **BOOSTXL-DRV8305** to set up the lab kit, connect the motor and power supply to the kit with a reference as below. Need to select whether to use the hall input on connector EQEP1 or EQEP2 of the LaunchPad by set “`#define JH_QEP_1`” or “`#define JH_QEP_2`” in hal.h.

BLY172S-24V			JH_QEP_1	JH_QEP_2
Wire Color	Description	BOOSTXL-DRV8301/5	LaunchXL-F28069M	
Yellow	Phase A	A		
Red	Phase B	B		
Black	Phase C	C		
Black	Hall Power Gnd		GND	GND
Red	Hall Power Supply		+5V	+5V
White	Hall A		EQEP1A-GPIO20	EQEP2A-GPIO54
Green	Hall B		EQEP1B-GPIO21	EQEP2B-GPIO55
Blue	Hall C		EQEP1I-GPIO23	EQEP2I-GPIO56

If using **F28069M Control Card** and **DRV8301-EVM** or **DRV8312-EVM** to set up the lab kit, connect the motor and power supply to the kit with a reference as below. Need to select whether to use the hall input on connector J4 or J10 of the EVM by set `“#define JH_J4” or “#define JH_J10”` in hal.h.

BLY172S-24V		JH_J4	JH_J10
Wire Color	Description	DRV8301-EVM DRV8312-EVM	
Yellow	Phase A	A	
Red	Phase B	B	
Black	Phase C	C	
Black	Hall Power Gnd	GND	GND
Red	Hall Power Supply	+5V	+5V
White	Hall A	QEPA-GPIO20	CAP1-GPIO24
Green	Hall B	QEPB-GPIO21	CAP2-GPIO25
Blue	Hall C	QEPI-GPIO23	CAP3-GPIO26

If using **F28069M Control Card** and **High Voltage Kit** to set up the lab kit, connect the motor and power supply to the kit with a reference as below. Need to select whether to use the hall input on connector QEP or CAP of the LaunchPad by set `“#define JH_QEP” or`
`“#define JH_CAP”` in hal.h.

BLWS235D-160V-3000		QEP	HALL/CAP
Wire Color	Description	BOOSTXL-DRV8301/5	
Yellow	Phase A	A	
Red	Phase B	B	
Black	Phase C	C	
Black	Hall Power Gnd	GND	GND
Red	Hall Power Supply	+5V	+5V
White	Hall A	QEP1-A	HALL-1
Green	Hall B	QEP1-B	HALL-2
Blue	Hall C	QEP1-I	HALL-3

F28027F Control Card only supports **High Voltage Kit**, connect the motor and power supply to the kit and use the hall input on HALL/CAP connector of the kit. Set SW2(4-5) and SW3(1-2, 4-5) on F28027F Control Card.

Step 2.

In user.h, make sure motor parameters are known and correctly set. Lab 11e only works on PM motors with 3 hall sensors.

Step 3.

In Code Composer, build proj_lab11e, connect to the target and load the .out file.

- Open the command file “sw\solutions\instaspin_foc\src\proj_lab11e.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
  - 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 run the motor with Hall sensors

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- Set low and high speed for changing the control method in “gHall_speed_FastToBldc_low_pu” and “gHall_speed_BldcToFast_high_pu”.
- To turn on the pwms to the motor, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.
- The acceleration can be modified by adjusting the value in “gMotorVars.MaxAccel_krpmps”.
- Set a reference speed to “gMotorVars.SpeedRef_krpm” in order to run the motor at a target speed.

Step 5.

When finished experimenting to stop the motor

- Set the variable “gMotorVars.Flag_Run_Identify” to 0 to shut down the motor.
- Turn off real-time control and stop the debugger.
- Turn off power supply of drive kit.

Conclusion

This lab suggests a sensor & sensorless hybrid control method for PMSM drive: hall-sensor based drive is adopted to boost the performance for start or low speed running and InstaSPIN-FOC based drive for high speed. This solution makes the traditional sensor based solution less challenging for high speed running due to the imperfection of hall senor signal. It also enables senorless solution for low speed running down to zero without sacrifice of performance.

Lab 12a – Sensored Inertia Identification

Abstract

For applications where a sensor is required InstaSPIN-MOTION can provide the same advantage it provides to sensorless applications. InstaSPIN-MOTION currently supports quadrature encoders; Additional information about sensored systems can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section 21 Sensored Systems).

Introduction

This lab discusses how to configure a quadrature encoder to provide feedback for the speed and angle information needed by SpinTAC™ and the FOC in order to control the motor. This lab will also identify the inertia for sensored applications.

Prerequisites

The user motor settings have been identified and populated in the InstaSPIN-MOTION user.h file. See the prerequisites for lab 05c for additional information

Objectives Learned

- Connect a quadrature encoder to your motor
- Use the quadrature encoder to replace the FAST estimator
- Estimate the system inertia

Background

This lab has a number of new API calls. Figure 69 shows the block diagram for this project. This lab builds from Lab 5c - InstaSPIN-MOTION Inertia Identification in order to present the simplest implementation of the quadrature encoder.

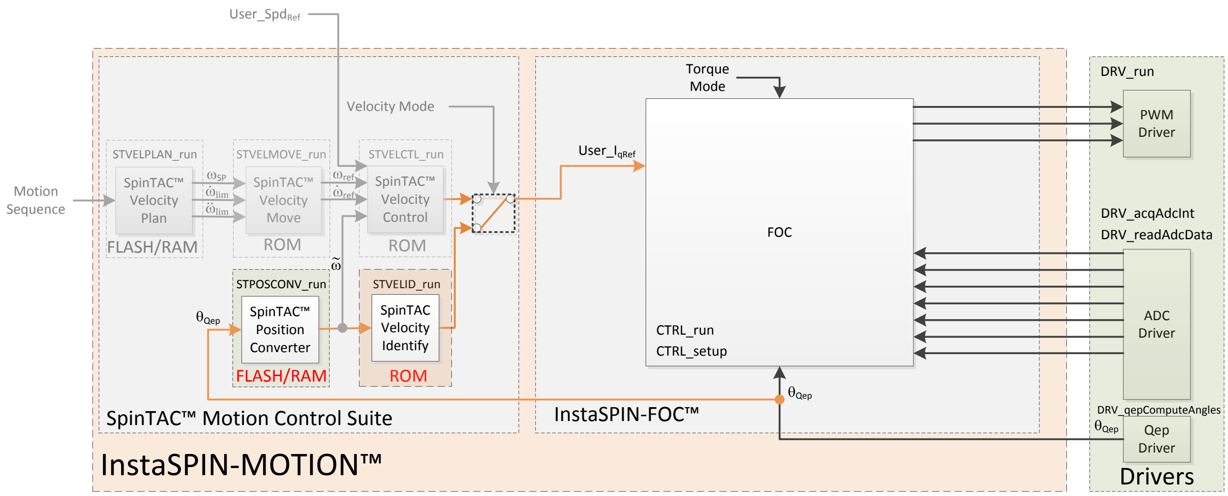


Figure 69: InstaSPIN-MOTION™ block Diagram for lab 12a

This lab adds the SpinTAC Position Converter into the project. This component takes the electrical angle from the ENC module and outputs a speed feedback that is used by SpinTAC Velocity Identify. This module will also calculate the slip required for AC Induction motors.

The additional API calls are designed to convert the raw encoder counts from the QEP driver into a usable electrical angle via the ENC module and the electrical angle into speed feedback that can be used in the rest of the system.

For this lab it is important to ensure that the physical connections are done correctly. If the motor or encoder wires are connected in the wrong order, the lab will not function properly. It will result in the motor being unable to move. For the motor it is important to ensure that the motor phases are connected to the right phase on the controller. Phase connections for the motors that are provided with the TI Motor Control Reference Kits are provided in Table 57.

Table 57: Motor phase connections for reference kit motors

Motor Phases	Anaheim BLY172S-24V-4000	Teknic M-2310P-LN-04K	Estun EMJ-04APB22
A / U	Yellow	T (White)	Red
B / V	Red	R (White-Black	Blue
C / W	Black	S (White-Red)	White

For the encoder it is important to ensure that A is connected to A, B to B, and I to I. Often +5V dc and ground connections are required as well. Please refer to the information for your board in order to wire your encoder correctly.

This project will setup the calibrated angle of the encoder. This ensures that both the encoder and the motor are aligned on a zero degree electrical angle. This step is done when the FAST™ estimator is recalibrating the Rs value of the motor. For AC Induction motors, this calibration is not required, since the motor will always start at a zero degree electrical angle.

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

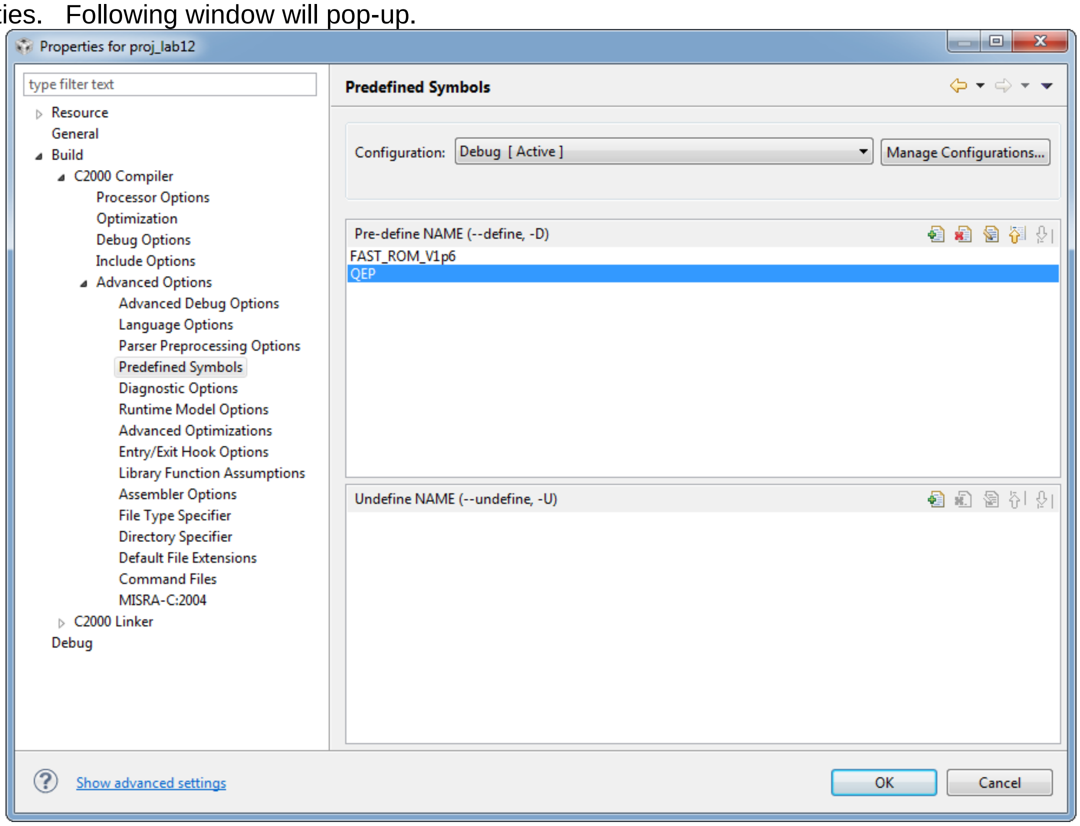
Project Files

To include the quadrature encoder into the project, a number of platform related files need to be modified to include the ENC module and to provide functions to access the values it produces. If you compare the non-QEP version of these platform files with the QEP version you will see the specific differences that have been made. Table 58 describes the files that have been modified to support the ENC module.

Table 58: New and updated files that must be included in the project

proj_lab12a		
	ctrlQEP.c	Contains code for CTRL_run and CTRL_setup, which is the code that runs the FOC. This has been modified to support the ENC module.
	enc.c	Contains the code to configure and the ENC module.
	qep.c	Contains the code to configure and run the eQEP driver.
	slip.c	Contains the code to configure and run the SLIP module

As part of this project there is a predefined symbol that is set in order to tell the project files to use the quadrature encoder instead of the FAST software encoder. If you right-click on proj_lab12a, and select properties. Following window will pop-up.



Open Build -> C2000 Compiler -> Advanced Options, and select Predefined Symbols. The highlighted line is the new predefined symbol that informs the project that it will use a quadrature encoder for feedback.

Include the Header File

A description of the newly included files for the quadrature encoder is shown in Table 59. Note that this list only contains the files that are different from those used in the previous labs.

Table 59: Important header files needed for motor control

main.h	Header file containing all included files used in main.c	
	platforms	
	ctrlQEP.h	Function definitions for the CTRL ROM library. Contains the CTRL object declaration. Modified to use the ENC electrical angle for PMSM and ENC magnetic angle for ACIM.

The critical header file for the SpinTAC components is spintac_pos_conv.h. This file contains the code required to configure the SpinTAC Position Converter.

Table 60: Important header files needed for SpinTAC components

spintac_velocity.h	Header file containing all SpinTAC header files used in main.c	
	SpinTAC	
	spintac_pos_conv.h	SpinTAC Position Converter structures and function declarations.

Declare the Global Structure

There are no new global variables declared in this lab.

Initialize the Configuration Variables

During the initialization and setup of the project the ENC module and the SLIP module need to be configured. The SpinTAC Position Converter also requires some setup to function properly. This is done by calling the function listed in Table 61. The lab is setup to configure the ENC module, SLIP Module, and SpinTAC Position Converter correctly.

Table 61: Important setup functions needed for the motor control

main		
	ENC	
	ENC_setup	Setups all the defualt value for the ENC module and makes it ready to produce feedback.
	SLIP	
	SLIP_setup	Setups all the defualt value for the SLIP module and makes it ready to produce feedback. (ACIM Only)
	SpinTAC	
	ST_setupPosConv	Setups all the default value for SpinTAC Position Converter.

Main Run-Time loop (forever loop)

No changes have been made in the forever loop for this lab.

Main ISR

The new functions that are required for this lab include functions in order to access the encoder feedback and process that feedback to produce speed feedback that will be used by the rest of the system. The new functions are listed in Table 62.

Table 62: InstaSPIN functions used in the main ISR

mainISR		
	ENC	
	ENC_calcElecAngle	This function calculates the current electrical angle for the ENC module. The electrical angle is based on the number of counts received by the QEP driver.
	ENC_getPositionMax	This function returns the current electrical angle of the motor
	ENC_setZeroOffset	This function sets the calibrated angle to align the ENC angle with the rotor angle (PMSM Only)
	ENC_getPositionMax	This function returns the maximum count of the quadrature encoder
	ENC_getRawEncoderCounts	This function returns the raw counts from the QEP driver
	SpinTAC	
	ST_runPosConv	This function calls the SpinTAC Position Converter
	EST	
	EST_getState	This function returns the state of the FAST Estimator
	SLIP	
	SLIP_setElectricalAngle	This function provides the electrical angle to the SLIP compensation module (ACIM Only)
	SLIP_run	This function calls the SLIP compensation module
	SLIP_getMagneticAngle	This function returns the slipped electrical angle (referred to as the magnetic angle)

Call the SpinTAC Position Converter

The new functions that are required for this lab include functions to access the electrical angle produced by the ENC module and process that angle to produce speed feedback that will be used by the rest of the system. The new functions are listed in Table 62.

Table 62: InstaSPIN functions used in ST_runPosConv

ST_runPosConv		
	ENC	
	ENC_getElecAngle	This function return the electrical angle produced by the ENC module.
	CTRL	
	CTRL_getIdq_in_addr	This function returns the address of the Idq current vector. (ACIM Only)
	SpinTAC	
	STPOSCONV_setElecAngle_erev	This function set the electrical angle (Pos_erev) in SpinTAC Position Converter
	STPOSCONV_setCurrentVector	This function set the Idq current vector in SpinTAC Position Converter (ACIM Only)
	STPOSCONV_run	This function calls into the SpinTAC Position Converter in order to produce speed feedback from the electrical angle produced by the eQEP module.
	STPOSCONV_getSlipVelocity	This function gets the estiamted amount of slip velocity in electrical revolutions per second (ACIM Only)
	SLIP	
	SLIP_setSlipVelocity	This function sets the slip velocity in the motor and calculates the incremental slip for each iteration of the FOC. (ACIM Only)

Call SpinTAC™ Velocity Identify

No new API elements have been added to this lab. The one change that has been made is to the feedback source used for SpinTAC™ Velocity Identify. Previously, SpinTAC Velocity Identify was using the FAST estimator to provide speed feedback. This has been replaced with the speed feedback from the SpinTAC™ Position Converter. The change is shown in Table 63.

Table 63: InstaSPIN functions used in ST_runVelId

ST_runVelId		
	SpinTAC	
	STPOSCONV_getVelocityFiltered	This function returns the filtered speed feedback (VelLpf) from SpinTAC Position Converter

Lab Procedure

After verifying that the correct encoder line count has been set as the value for USER_MOTOR_ENCODER_LINES, use Code Composer to build lab12a. Start a Debug session and download the proj_lab12a.out file to the MCU.

- Open the command file “sw\solutions\instaspin_motion\src\proj_lab12a.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

To run the motor a couple of steps are required:

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To start the current loop controller, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

At this point the motor is using the quadrature encoder to provide the motor angle information into the FOC. This process has removed the FAST estimator from the system.

- Set “gMotorVars.SpinTAC.VelIdGoalSpeed_krpm” equal to the rated speed of your motor. This will ensure a more accurate inertia result.
- Set “gMotorVars.SpinTAC.VelIdRun” to 1. Watch how the motor spins for a few seconds. It is running the open-loop inertia identification provided by SpinTAC.
    - If the motor did not spin or spun in the wrong direction at this step there are a number of things that need to be verified:
        - The number of encoder lines in USER_MOTOR_ENCODER_LINES
        - The motor phase lines are connected in the correct order
        - The motor encoder lines are connected in the correct order
    - If the value of “gMotorVars.SpinTAC.VelIdErrorID” is set to non-zero then the inertia identification process has failed.
        - If the value is 2004 and the motor spun at a speed, most likely that the goal speed was set too high. Reduce “gMotorVars.SpinTAC.VelIdGoalSpeed_krpm” by half and try again.
        - If the value is 2003, most likely that the torque rate was set too low. Decrease “gMotorVars.SpinTAC.VelIdTorqueRampTime_sec” by 1.0 to have the torque be applied quicker.
        - If the value is 2006, this means that the motor did not spin through the entire inertia identification process. Decrease “gMotorVars.SpinTAC.VelIdTorqueRampTime_sec” by 1.0 to have the torque change quicker during the test.
    - The value of the motor inertia is placed into “gMotorVars.SpinTAC.InertiaEstimate_Aperkrpm”
    - The value of the motor friction is placed into “gMotorVars.SpinTAC.FrictionEstimate_Aperkrpm”

Open user.h following these steps:

1. Expand user.c from the Project Explorer window

2. Right-mouse click on user.h 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

Opening the Outline View will provide an outline of the user.h contents

In the section were you have defined your motor there should be two additional definitions to hold the inertia and the friction of your system. Place the values for inertia and friction from gMotorVars.SpinTAC.InertiaEstimate_Aperkrpm and gMotorVars.SpinTAC.FrictionEstimate_Aperkrpm as the values for these defines.

- USER_SYSTEM_INERTIA (gMotorVars.SpinTAC.InertiaEstimate_Aperkrpm)
- USER_SYSTEM_FRICTION (gMotorVars.SpinTAC.FrictionEstimate_Aperkrpm)

The motor inertia is now identified.

You can also estimate the motor inertia in less than 1 complete rotation. To do this, follow these guidelines

- Reduce the goal speed in gMotorVars.SpinTAC.VelIdGoalSpeed_krpm, this won’t spin the motor as quickly during the test
    - Start by setting this value to 1/10th of the rated speed of the motor
        - This value should be greater than the minimum speed resolution of your encoder
- Reduce the torque ramp in gMotorVars.SpinTAC.VelIdTorqueRampTime_sec, this will have the torque change quicker during the test
    - Start by setting this value to 1.0
        - This value can be decreased further, but will introduce more jerk since the torque is being changed faster

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how to use InstaSPIN-MOTION in a sensored system. It showed that it is easy to setup a quadrature encoder and identify the system inertia is less than one revolution.

Lab 12b - Using InstaSPIN-MOTION with Sensored Systems

Abstract

For applications where a sensor is required InstaSPIN-MOTION can provide the same advantage it provides to sensorless applications. InstaSPIN-MOTION currently supports quadrature encoders; Hall effect sensors may be available in a future release. Additional information about sensored systems can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section 21 Sensored Systems).

Introduction

This lab discusses using a physical sensor to provide feedback for the SpinTAC Speed Controller.

Prerequisites

The motor inertia value has been identified and populated in user.h. This process should be completed in Lab 12a – Sensored Inertia Identification. The quadrature encoder has also been confirmed working in that lab.

Objectives Learned

- Use the quadrature encoder to replace the FAST estimator for speed feedback
- Control the speed of a sensored motor

Background

This lab has a number of new API calls. Figure 70 shows the block diagram for this project. This lab is based on Lab 5e - Tuning the InstaSPIN-MOTION Speed Controller in order to present the simplest implementation of the quadrature encoder.

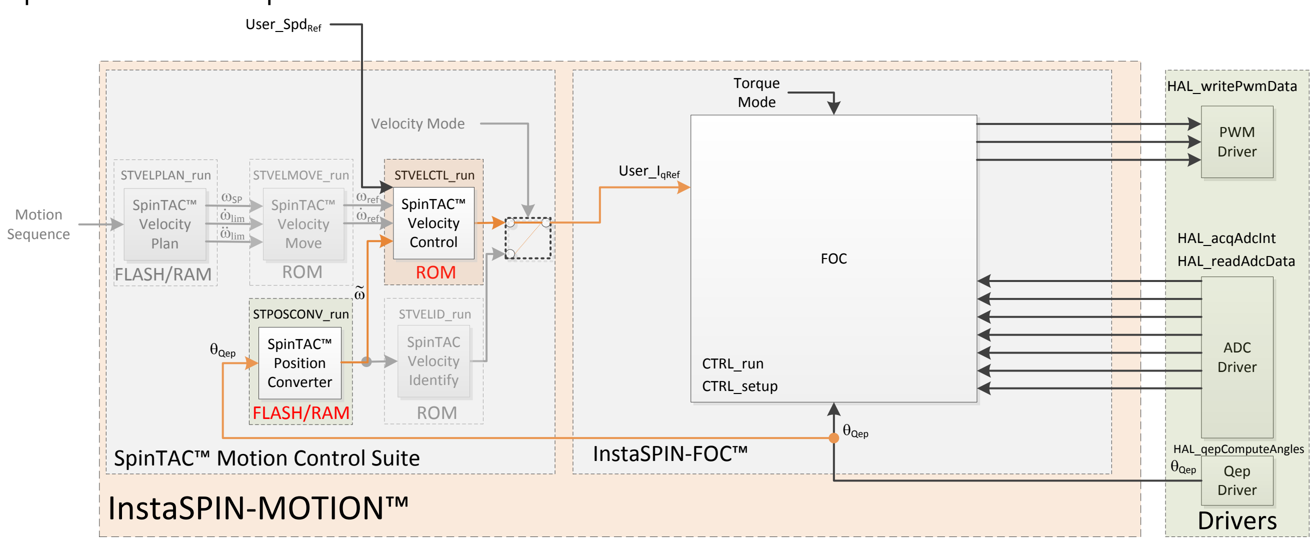


Figure 70: InstaSPIN-MOTION block diagram for lab 12

For this lab it is important to ensure that the physical connections are done correctly. If the motor or encoder wires are connected in the wrong order, the lab will not function properly. It will result in the motor being unable to move. For the motor it is important to ensure that the motor phases are connected to the right phase on the controller. Phase connections for the motors that are provided with the TI Motor Control Reference Kits are provided in Table 64.

Table 64: Motor phase connections for reference kit motors

Motor Phases	Anaheim BLY172S-24V-4000	Teknic M-2310P-LN-04K	Estun EMJ-04APB22
A / U	Yellow	T (White)	Red
B / V	Red	R (White-Black	Blue
C / W	Black	S (White-Red)	White

For the encoder it is important to ensure that A is connected to A, B to B, and I to I. Often +5V dc and ground connections are required as well. Please refer to the information for your board in order to wire your encoder correctly.

This project will setup the calibrated angle of the encoder. This ensures that both the encoder and the motor are aligned on a zero degree electrical angle. This step is done when the FAST™ estimator is recalibrating the Rs value of the motor. For AC Induction motors, this calibration is not required, since the motor will always start at a zero degree electrical angle.

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

Project Files

There are no new project files in this lab.

Include the Header File

There are no new header files in this lab.

Declare the Global Structure

There are no new global variables declared in this lab.

Initialize the Configuration Variables

No changes have been made to the configuration section of this lab.

Main Run-Time loop (forever loop)

No changes have been made in the forever loop for this lab.

Main ISR

There are no new function calls in this lab.

The function calls for SpinTAC Velocity Identify have been replaced by the function calls for the SpinTAC Speed Controller (see Lab 5e - Tuning the InstaSPIN-MOTION Speed Controller for more information).

Call the SpinTAC™ Speed Controller

No new API elements have been added to this lab. The one change that has been made is to the feedback source used for the SpinTAC™ speed controller. Previously, the SpinTAC Velocity Control was using the FAST estimator to provide speed feedback. This has been replaced with the speed feedback from the SpinTAC™ Position Converter. The change is shown in Table 65.

Table 65: InstaSPIN functions used in ST_runVelCtl

ST_runVelCtl		
	SpinTAC	
	STPOSCONV_getVelocityFiltered	This function returns the filtered speed feedback (VelLpf) from SpinTAC Position Converter

Lab Procedure

After verifying that the correct encoder line count has been set as the value for USER_MOTOR_ENCODER_LINES, use Code Composer to build lab12b. Start a Debug session and download the proj_lab12b.out file to the MCU.

- Open the command file “sw\solutions\instaspin_motion\src\proj_lab12b.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

To run the motor a couple of steps are required:

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To start the current loop controller, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

At this point the motor is under speed control using the quadrature encoder to provide both the speed feedback as well as the motor angle information into the FOC. This process has removed the FAST estimator from the system.
If the motor did not spin at this step there are a number of things that need to be verified:

- The number of encoder lines in USER_MOTOR_ENCODER_LINES
- The motor phase lines are connected in the correct order
- The motor encoder lines are connected in the correct order

Set a speed reference to “gMotorVars.SpeedRef_krpm” in order to get the motor to spin at speed.

- If the motor does not spin at this step it typically indicates a wiring problem.
  - Double check the motor wiring to ensure that it is connected in the correct order.
  - Double check the encoder wiring to ensure that it is connected correctly.

Notice that the motor should have even better speed response and control than with the FAST estimator.

Continue to update the value in “gMotorVars.SpeedRef_krpm” with the speed you would like the motor to run. The acceleration can also be modified by adjusting the value in “gMotorVars.MaxAccel_krpmps”.

You should notice smoother starts from zero speed as well as the ability to spin the motor at lower speeds than with the FAST estimator.

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how to use InstaSPIN-MOTION in a sensored system. It showed that it is easy to setup a quadrature encoder and get your motor spinning with a feedback sensor.

Lab 12c – Dual Motor Sensored Velocity InstaSPIN-MOTION

Abstract

Please see code comments for implementation details.

Writeups will not be provided for InstaSPIN-MOTION proj_lab6e, proj_lab12c nor proj_lab13f. Please review the code to understand the technique and important variables employed in these labs.

A writeup for dual motor control using InstaSPIN-FOC is provided in the proj_lab10d writeup.

Proj_lab11d is a valid dual motor control project for InstaSPIN-FOC based on the simplified structure of proj_lab11. Labs 6e, 12c, and 13f are InstaSPIN-MOTION projects that build upon the lab 11d code base.

Please note that proj_lab11d does not include a trajectory module; as such, there is no acceleration variable, and any changes the speed reference are applied as a step input. Use caution. InstaSPIN-

Lab 13a - Tuning the InstaSPIN-MOTION Position Controller

Abstract

Tuning position control applications can be very difficult and time consuming. InstaSPIN-MOTION provides a position-velocity controller that can be tuned using a single coefficient. This single gain (bandwidth) typically works across the entire range of loads and transitions in applications, reducing their complexity. This lab demonstrates how to connect the InstaSPIN-MOTION position controller and tune it for your application.

Introduction

This lab demonstrates the functionality of the position controller included in InstaSPIN-MOTION. The position controller is provided as part of the SpinTAC Motion Control Suite.

Prerequisites

The system inertia value has been identified and populated in user.h. This process should be completed in Lab 12a – Sensored Inertia Identification. A source is available to provide the electrical angle of the motor to the FOC & the SpinTAC Position Converter. An example of how to do this with a quadrature encoder is provided in Lab 12a – Sensored Inertia Identification.

Objectives Learned

- Use the InstaSPIN-MOTION Position Controller
- Tune the InstaSPIN-MOTION Position Controller for your application

Background

This lab has a number of new API calls. Figure 71 shows the block diagram for this project. This lab builds from Lab 12b - Using InstaSPIN-MOTION with Sensored Systems in order to show how to add the InstaSPIN-MOTION Position Controller once your quadrature encoder is functional.

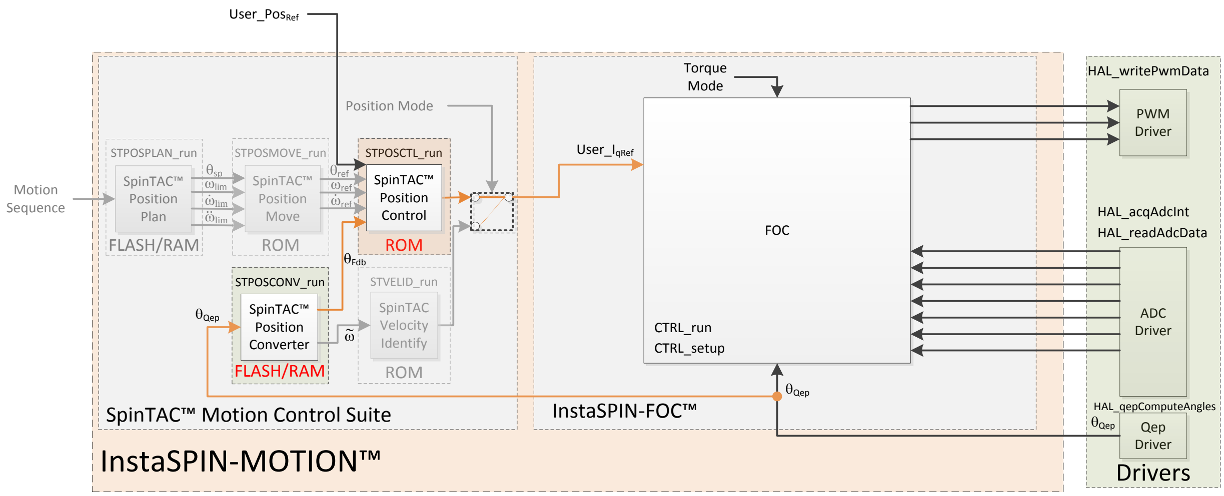


Figure 71: InstaSPIN-MOTION block diagram for lab 13a

This lab adds SpinTAC Position Control into the project. This component takes the mechanical angle from the SpinTAC Position Converter and a reference mechanical angle in order to generate a torque output that is provided to the FOC.

The SpinTAC Motion Control Suite applies a rollover bound to the position signal. It does this in order to maintain the precision of the position signal. When the position signal hits the upper bound the rollover count is increased, similarly when the position signal hits the lower bound the rollover count is decreased. This allows the SpinTAC Motion Control Suite to operate on positions that might be larger than what can be stored in a fixed-point variable. Figure 72 displays the position signal for a position move of 120 mechanical revolutions.

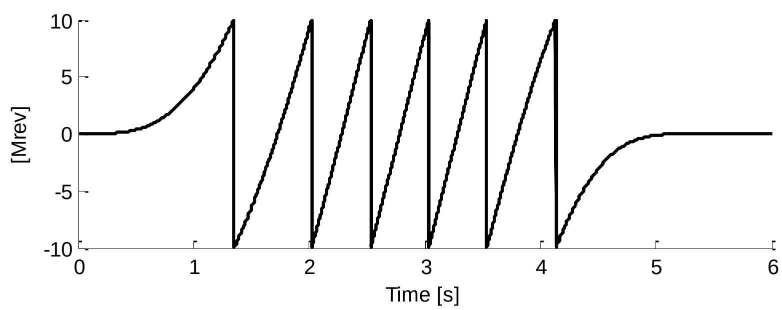


Figure 72: Position Sawtooth Signal

This project will setup the calibrated angle of the encoder. This ensures that both the encoder and the motor are aligned on a zero degree electrical angle. This step is done when the FAST™ estimator is recalibrating the Rs value of the motor.

Project Files

There are no new project files in this project.

Include the Header File

To accommodate the differences between position control and velocity control a different main.h is required. This new file is called main_position.h. This file holds the includes and global structures that are required for position control. The includes that are different between main.h & main_position.h are captured in Table 66.

Table 66: Important header files needed for the motor control

main_position.h	Header file containing all included files used in main.c	
	spintac	
	spintac_position.h	SpinTAC posiiton component setup and configuration.

The critical header file for the SpinTAC position components is spintac_position.h. This file contains the code required to configure the SpinTAC Position Control. This header file is common across all position labs so there will be more includes than are needed for this lab. The new headers are captured in Table 67.

Table 67: Important header files needed for SpinTAC components

spintac_position.h	Header file containing all SpinTAC header files used in main.c	
	SpinTAC	
	spintac_pos_ctl.h	SpinTAC Position Control structures and function declarations.

Declare the Global Structure

There are no new global variables declared in this lab. For consistency, the global SpinTAC object has the same name, but the contents have changed. The velocity components have been replaced by the position components.

Initialize the Configuration Variables

During the initialization and setup of the project the SpinTAC Position Control needs to be configured. This is done by calling the function listed in Table 68. The lab is setup to configure the SpinTAC Position Control module correctly.

Table 68: Important setup functions needed for the motor control

main		
	SpinTAC	
	ST_setupPosCtl	Setups all the defualt values for the SpinTAC Position Controller.

Main Run-Time loop (forever loop)

One change has been made to the forever loop of this lab. When the FAST estimator is not in the state EST_State_OnLine, we need to make sure that the SpinTAC Position Control is placed into reset. This is done because when performing the Rs recalibration, the system is not ready to be under position control. Once this step is complete the SpinTAC Position Control can be enabled. These functions are listed in Table 69.

Table 69: InstaSPIN functions used in the main forever loop

main		
	SpinTAC	
	STPOSCTL_setReset	Sets the reset bit inside the SpinTAC Positiion Control
	STPOSCTL_setEnable	Sets the enable bit inside the SpinTAC Positon Control

Main ISR

The new functions that are required for this lab include functions in order to run the SpinTAC Position Control. The new functions are listed in Table 70.

Table 70: InstaSPIN functions used in the main ISR

mainISR		
	SpinTAC	
	ST_runPosCtl	This function calls the SpinTAC Position Control

Call SpinTAC Position Control

The functions that are required to run SpinTAC Position Control are listed in Table 71. These functions provide the ability to provide references and feedback to SpinTAC Position Control and pass the output from SpinTAC Position Control to the FOC.

Table 71: InstaSPIN functions used in ST_runPosCtl

ST_runPosCtl		
	SpinTAC	
	STPOSCTL_setPositionReference_mrev	This function sets the position reference [Mrev] in SpinTAC Position Control
	STPOSCTL_setVelocityReference	This function sets the velocity reference [pu/s] in SpinTAC Position Control
	STPOSCTL_setAccelerationReference	This function sets the acceleration reference [pu/s^2] in SpinTAC Position Control
	STPOSCTL_setPositionFeedback_mrev	This function sets the position feedback [Mrev] in SpinTAC Position Control
	STPOSCONV_getPosition_mrev	This function gets the position [Mrev] from the SpinTAC Position Converter
	STPOSCTL_run	This function runs the SpinTAC Position Control
	STPOSCTL_getTorqueReference	This function gets the output signal from SpinTAC Position Control
	CTRL	
	CTRL_setIq_ref_pu	This function sets the Iq refence in the FOC

Lab Procedure

In Code Composer, build proj_lab13a. Start a Debug session and download the proj_lab13a.out file to the MCU.

- Open the command file "sw\solutions\instaspin_motion\src\proj_lab13a.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

To run the motor a couple of steps are required:

- To start the project, set the variable "gMotorVars.Flag_enableSys" equal to 1.
- To start the current loop controller, set the variable "gMotorVars.Flag_Run_Identify" equal to 1.

At this point the motor is under position control using the quadrature encoder to provide both the position feedback as well as the motor angle information into the FOC. It is important to note that the motor will not spin in this lab. The position reference for this lab is set to 0.

If the motor spun unexpectedly, at this step there are a number of things that need to be verified:

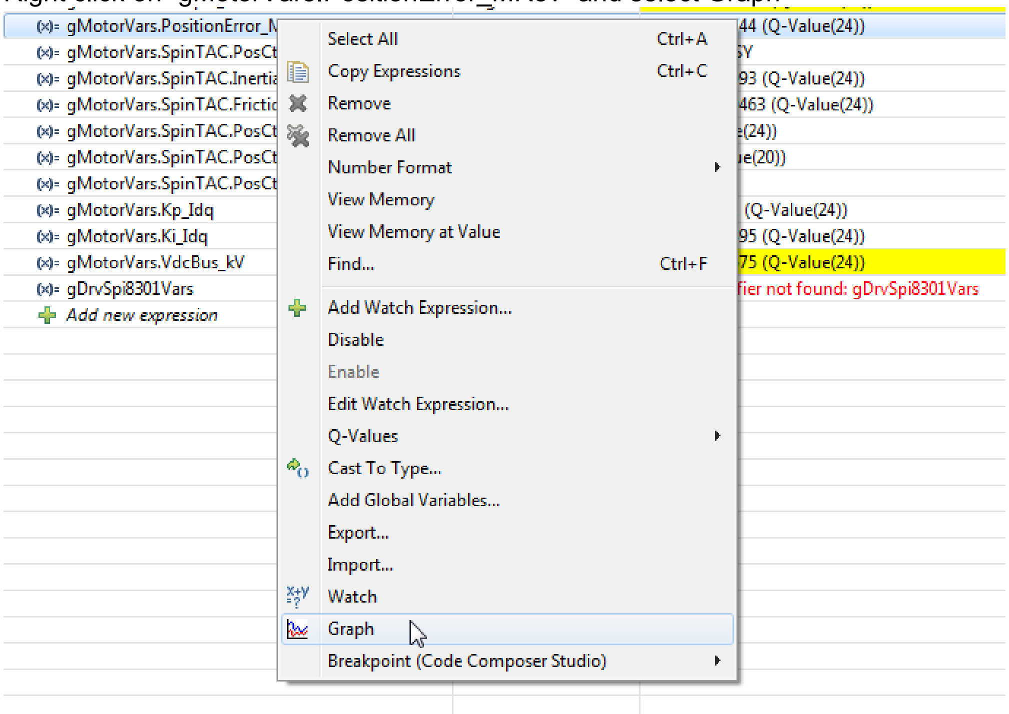
- The number of encoder lines in USER_MOTOR_ENCODER_LINES
- The motor phase lines are connected in the correct order
- The motor encoder lines are connected in the correct order

Manually rotate the motor to feel how the position controller is fighting to hold the position at 0. At this point it should not be holding zero very strongly.

The value of the position error that is being introduced is available in "gMotorVars.PositionError_MRev." This value is in mechanical revolutions. As the motor is manually rotated, the current position error can be watched here.

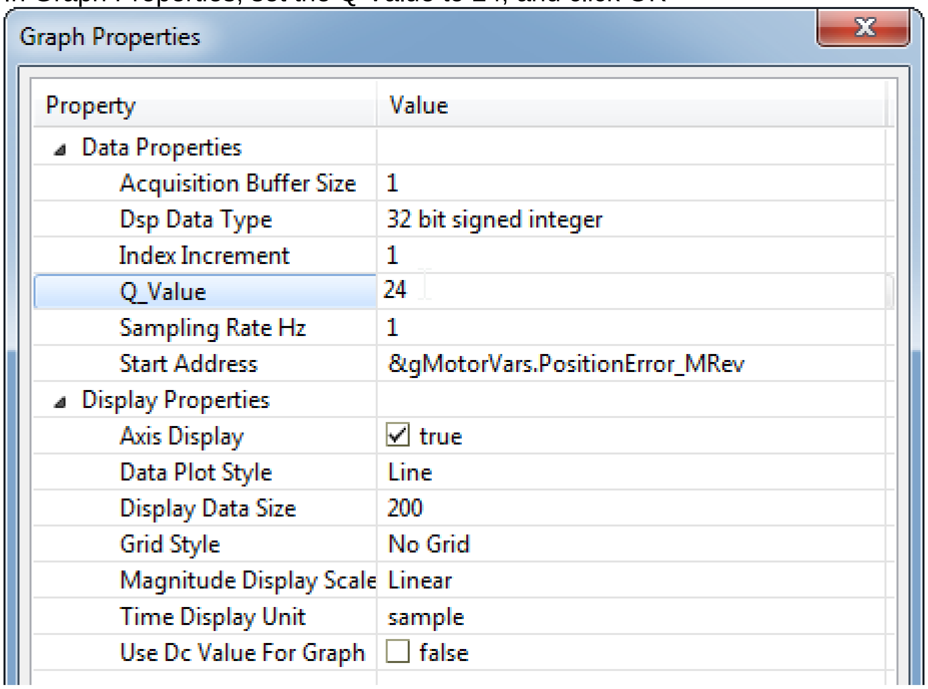
The position error variable can also be graphed to visually watch the error signal.

1. Right click on “gMotorVars.PositionError_MRev” and select Graph

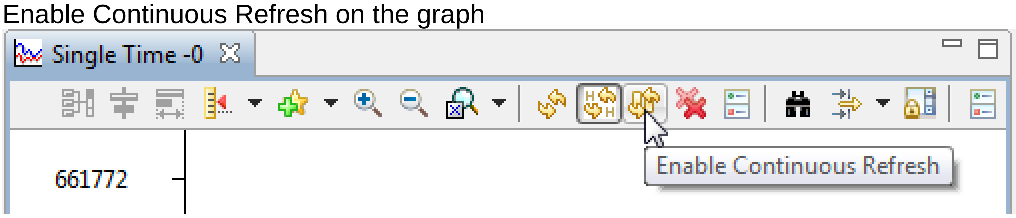


2. Open the Graph Properties

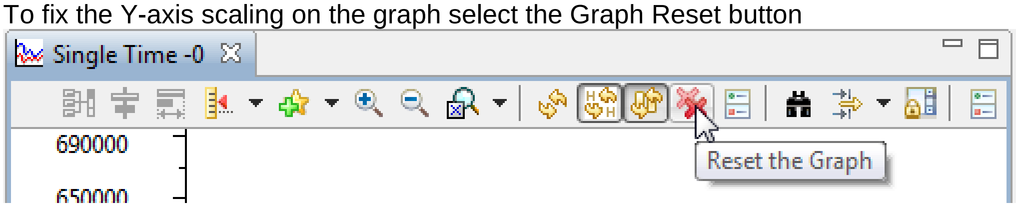
3. In Graph Properties, set the Q Value to 24, and click OK



4. Enable Continuous Refresh on the graph



5. To fix the Y-axis scaling on the graph select the Graph Reset button



Resetting the graph can be done at any time

To tune SpinTAC Position Control you need to adjust the value in “gMotorVars.SpinTAC.PosCtlBwScale.” This value sets the Bandwidth in the SpinTAC Position Control. As the Bandwidth is increased the motor will be able to hold a zero position much tighter.

- Increase the bandwidth scale “gMotorVars.SpinTAC.PosCtlBwScale” in steps of 1.0, continuing to feel how tightly the motor is holding zero speed.
- Once SpinTAC Position Control is tightly holding zero the bandwidth scale has been tuned.

If your motor starts to oscillate or vibrate, than the Bandwidth has been set too high and it needs to be reduced by 10-20%.

Once you have identified the ideal bandwidth for your system it should be stored in user.h. This will allow it to be the default bandwidth in future labs.

At the top of the USER MOTOR defines section in user.h there is a parameter called USER_SYSTEM_BANDWIDTH_SCALE. Update the value for this define with the value you identified during the tuning process.

- USER_SYSTEM_BANDWIDTH_SCALE (gMotorVars.SpinTAC.PosCtlBwScale)

When done experimenting with the motor:

- Set the variable "Flag_Run_Identify" to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how to use InstaSPIN-MOTION in a position control system. It showed how easy it is to setup the SpinTAC Position Control. It also demonstrated how quickly SpinTAC Position Control can be tuned to meet your performance requirements.

Lab 13b - Smooth Position Transitions with SpinTAC™ Move

Abstract

InstaSPIN-MOTION includes SpinTAC Move, a motion profile generator that generates constraint-based, time-optimal position trajectory curves. It removes the need for lookup tables, and runs in real time to generate the desired motion profile. This lab will demonstrate the different configurations and their impact on the final position transition of the motor.

Introduction

InstaSPIN-MOTION includes a position profile generator as part of the SpinTAC Motion Control Suite. This position profile generator is called SpinTAC Move. It is a position profile generator that computes the time-optimal curves within the user defined velocity, acceleration, deceleration, and jerk bounds. It supports basic ramp profile, as well as advanced s-curve and st-curve (Linestream Proprietary) curves. The proprietary st-curve features a continuous jerk to provide additional smoothing on the trajectory.

Prerequisites

This lab assumes that the system inertia has been identified and SpinTAC Position Control has been tuned.

Objectives Learned

- Use SpinTAC Move to transition between positions.
- Become familiar with the bounds that can be adjusted as part of SpinTAC Move
- Continue exploring how SpinTAC Position Control takes advantage of the advanced features of SpinTAC Position Move

Background

This lab adds new API functions calls to call SpinTAC Move. Figure 73 shows how SpinTAC Move connects with the rest of the SpinTAC components.

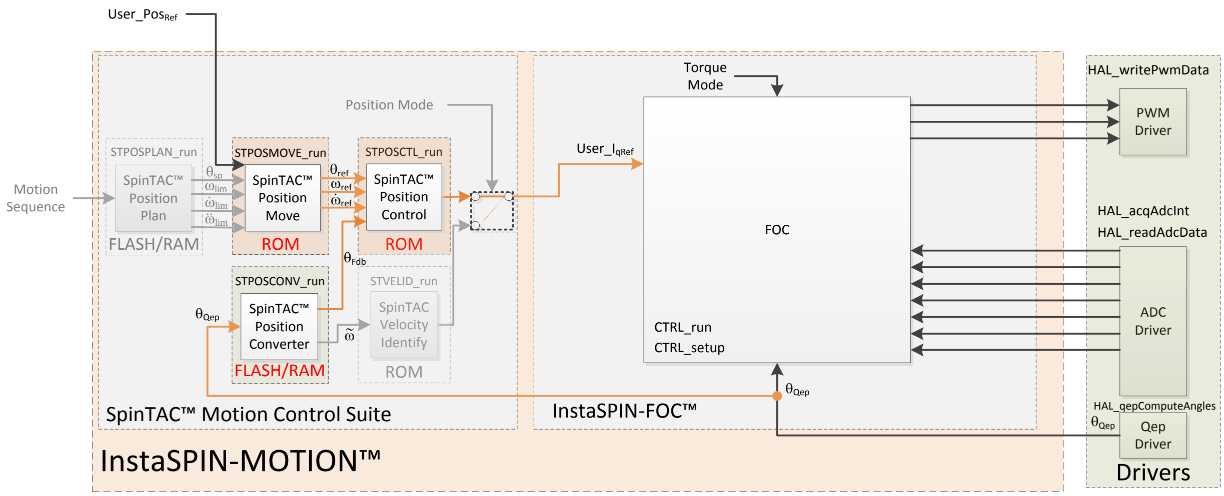


Figure 73: InstaSPIN-MOTION block diagram for lab 13b

This lab adds SpinTAC Position Move to provide trajectory, or position transition, curves to the SpinTAC Position Control. The block diagram shows the connections between the two components. SpinTAC Position Move accepts the user position step command and outputs the position, speed, and acceleration references to SpinTAC Position Control.

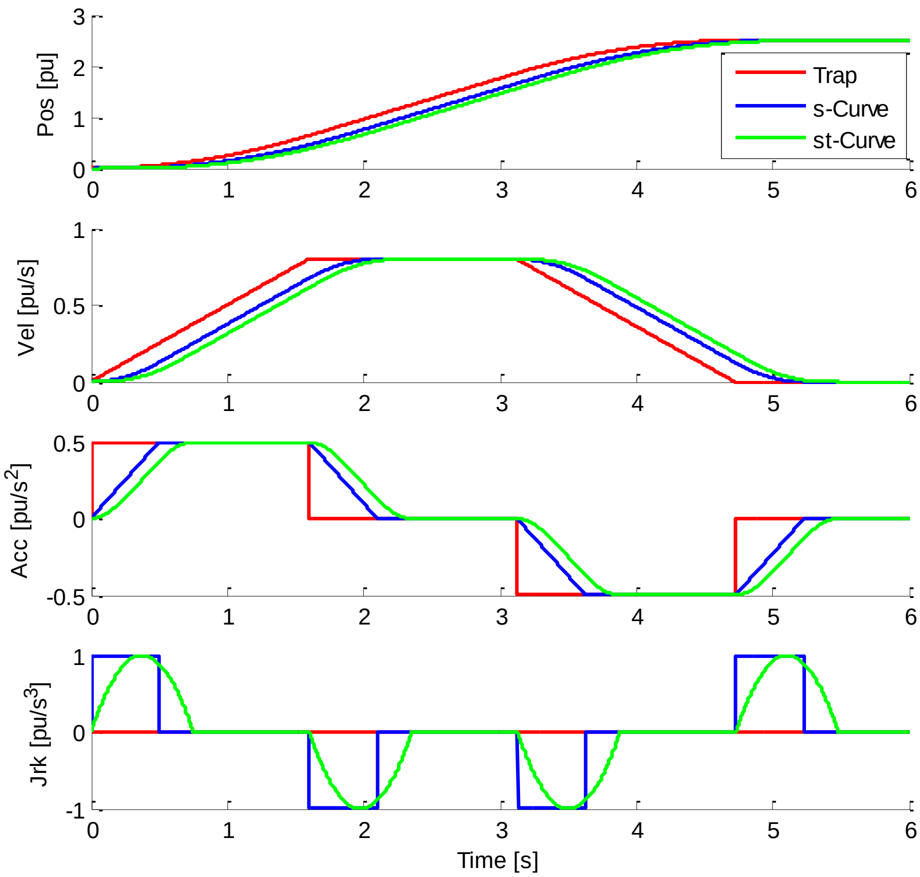


Figure 74: Trajectory Curves Available from SpinTAC Move

Figure 74 illustrates the differences between the three curve types available in SpinTAC Move. The st-curve represents the smoothest motion, which is critical for systems that are sensitive to large amounts of jerk. Jerk represents the rate of change of acceleration. A larger jerk will increase the acceleration at a faster rate. Steps, or sharp movement between two positions, can cause the system to oscillate. Control over jerk can round the velocity corners, reducing oscillation. As a result, acceleration can be set higher. Controlling the jerk in your system will lead to less mechanical stress on your system components and can lead to better reliability and less failing parts.

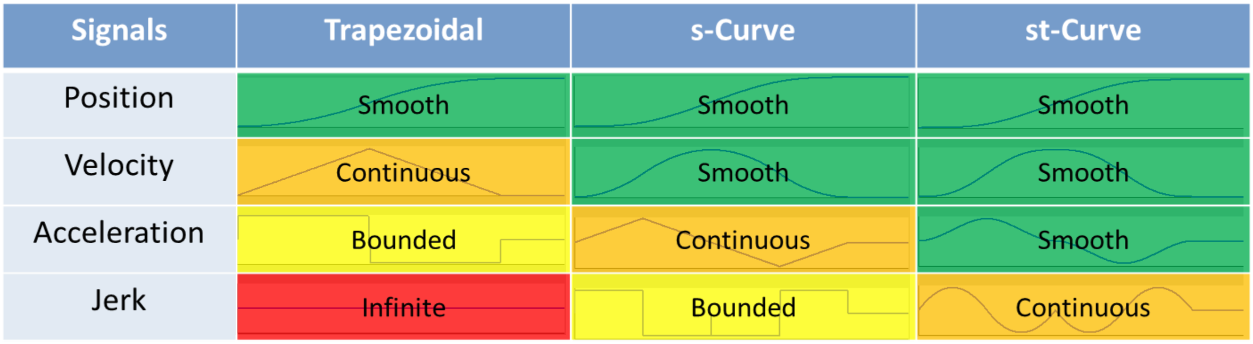


Figure 75: Chart describing curves available in SpinTAC™ Move

Figure 75 shows the different characteristics of the three curve types provided by SpinTAC Position Move. St-curve provides the smoothest motion by smoothing out the acceleration of the profile. For most applications the st-curve represents the best motion profile.

SpinTAC Position Move operates on relative position transitions. It is not based on an absolute position system. Each position command that is provided to SpinTAC Position Move is treated as a position step. Therefore, if your system were to start at a position of 0.5 mechanical revolutions and you commanded SpinTAC Position Move to move 1 mechanical revolution, the end absolute position would be 1.5 mechanical revolutions.

It is also important to note that in SpinTAC Position Move the ratio between the configured deceleration and acceleration must be with the following range: [0.1, 10].

Project Files

There are no new project files in this project.

Include the Header File

The critical header file for the SpinTAC position components is spintac_position.h. This file contains the code required to configure SpinTAC Position Move. This header file is common across all position labs so there will be more includes than are needed for this lab. The new headers are captured in Table 72.

Table 72: Important header files needed for SpinTAC components

spintac_position.h	Header file containing all SpinTAC header files used in main.c	
	SpinTAC	
	spintac_pos_move.h	SpinTAC Position Move structures and function declarations.

Declare the Global Structure

There are no new global object and variable declarations.

Initialize the Configuration Variables

During the initialization and setup of the project the SpinTAC Position Move needs to be configured. This is done by calling the function listed in Table 73. The lab is setup to configure the SpinTAC Position Move module correctly.

Table 73: Important setup functions needed for the motor control

main		
	SpinTAC	
	ST_setupPosMove	Setups all the defualt values for SpinTAC Position Move

Main Run-Time loop (forever loop)

One change has been made to the forever loop of this lab. Until the motor has been identified, the starting position needs to be provided to SpinTAC Position Move. Table 74 describes this function call.

Table 74: InstaSPIN functions used in the main forever loop

main		
	SpinTAC	
	STPOSMOVE_setPositionStart_mrev	Sets the starting position in SpinTAC Position Move

Main ISR

The new functions that are required for this lab include functions in order to run SpinTAC Position Move. The new functions are listed in Table 75.

Table 75: InstaSPIN functions used in the main ISR

mainISR		
	SpinTAC	
	ST_runPosMove	This function calls SpinTAC Position Move

Call SpinTAC Position Control

In the previous lab, the references provided to SpinTAC Position Control were fixed at 0. In this lab, these references are provided by SpinTAC Position Move. The new functions required to provide the reference from SpinTAC Position Move are listed in Table 76.

Table 76: InstaSPIN functions used in ST_runPosCtl

ST_runPosCtl		
	SpinTAC	
	STPOSMOVE_getPositionReference_mrev	This function gets the position reference [Mrev] from SpinTAC Position Move
	STPOSOVE_getVelocityReference	This function gets the velocity reference [pu/s] from SpinTAC Position Move
	STPOSMOVE_getAccelerationReference	This function gets the acceleration reference [pu/s^2] from SpinTAC Position Move

Call SpinTAC Position Move

The functions required to run SpinTAC Position Move are listed in Table 77. These functions set the user provided limits, curve type, and call SpinTAC Position Move.

Table 77: InstaSPIN functions used in ST_runPosMove

ST_runPosMove		
	SpinTAC	
	STPOSMOVE_getStatus	This function gets the status of SpinTAC Position Move
	STPOSOVE_setCurveType	This function sets the curve type in SpinTAC Position Move
	STPOSMOVE_setPositionStep_mrev	This function sets the position step [MRev] in SpinTAC Position Move
	STPOSMOVE_setVelocityLimit	This function sets the velocity limit [pu/s] in SpinTAC Position Move
	STPOSMOVE_setAccelerationLimit	This function sets the acceleration limit [pu/s^2] in SpinTAC Position Move
	STPOSMOVE_setDecelerationLimit	This function sets the deceleration limit [pu/s^2] in SpinTAC Position Move
	STPOSMOVE_setJerkLimit	This function sets the jerk limit [pu/s^3] in SpinTAC Position Move
	STPOSMOVE_setEnable	This function sets the enable bit in SpinTAC Position Move
	STPOSMOVE_run	This function runs SpinTAC Position Move

Lab Procedure

In Code Composer, build proj_lab13b. Start a Debug session and download the proj_lab13b.out file to the MCU.

- Open the command file "sw\solutions\instaspin_motion\src\proj_lab13b.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

To run the motor a couple of steps are required:

- To start the project, set the variable "gMotorVars.Flag_enableSys" equal to 1.
- To start the current loop controller, set the variable "gMotorVars.Flag_Run_Identify" equal to 1.

SpinTAC Position Move will generate motion profiles every time it is triggered to run.

First, load in the desired Position Step.

- To have the motor move 1.5 revolutions, set "gMotorVars.PosStepInt_MRev" to "1" and "gMotorVars.PosStepFrac_MRev" to "0.5". SpinTAC Position move will combine these two values to get to the final position step
- Then, set "gMotorVars.RunPositionProfile" to true to trigger SpinTAC Position Move.

The motor will spin exactly one and a half revolutions, but not very quickly. To get the motor to rotate one revolution much faster you need to increase the velocity, acceleration, and jerk limits.

- "gMotorVars.MaxVel_krpm" configures the velocity used in the profile. Set this value to 4.0. This will allow the motor to spin at maximum speed.
- Set "gMotorVars.PosStepInt_MRev" to 1.0 then set "gMotorVars. RunPositionProfile" to true.

The motor still did not spin very quickly. This is because we did not modify the acceleration or jerk limits. Despite setting the velocity limit to maximum, the trajectory isn't accelerating fast enough to reach that velocity.

- "gMotorVars.MaxAccel_krpmps" configures the acceleration used in the profile. Set this value to 75.0. This is the maximum value for acceleration in this project.
- "gMotorVars.MaxDecel_krpmps" configures the deceleration used in the profile. Set this value to 75.0. This is the maximum value for deceleration in this project.
- It is important to note that the deceleration limit needs to be in the following range:
  - [0.1 * gMotorVars.MaxAccel_krpmps, 10 * gMotorVars.MaxAccel_krpmps]
- Set "gMotorVars. PosStepInt _MRev" to 1.0 then set "gMotorVars. RunPositionProfile" to true.

The motor rotated a little faster, but not a lot faster. This is because we have not yet modified the jerk limit.

- "gMotorVars.MaxJerk_krpmps2" configures the jerk used in the profile. Set this value to 400.0. This is the maximum value for jerk in this project.
- Set "gMotorVars.PosStepInt_MRev" to 1.0 then set "gMotorVars. RunPositionProfile" to true.

Notice that the motor made the same single revolution much faster now that the limits have been increased.

SpinTAC Move supports three different curve types: trapezoid, s-curve, and st-curve. The curve type can be selected by changing "gMotorVars.SpinTAC.PosMoveCurveType." The differences between the three curves are discussed in detail in the InstaSPIN-MOTION User's Guide.

SpinTAC Move will alert the user when it has completed a profile via the done bit. When the profile is completed, "gMotorVars.SpinTAC.PosMoveDone" will be set to 1. This could be used in a project to alert the system when the motor has completed a movement.

When done experimenting with the motor:

- Set the variable "Flag_Run_Identify" to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how easy it is to use SpinTAC Move to generate constraint-based, time-optimal motion profiles. This lab also shows the different curves that can be used with SpinTAC Move. The st-curve provides a continuous jerk profile that will enable very smooth motion for jerk sensitive applications.

Lab 13c - Motion Sequence Position Example

Abstract

InstaSPIN-MOTION includes SpinTAC Plan, a motion sequence planner that allows you to easily build complex motion sequences. You can use this functionality to quickly build your application's motion sequence and speed up development time. This lab provides a very simple example of a motion sequence.

Introduction

SpinTAC™ Plan implements motion sequence planning. It allows for you to quickly build a motion sequence to run your application. SpinTAC™ Plan features: conditional transitions, variables, state timers, and actions. This lab will use a simple example to show how to quickly implement your application's motion sequence.

Objectives Learned

- Use SpinTAC™ Position Plan to design a motion sequence.
- Use SpinTAC™ Position Plan to run a motion sequence.
- Understand the features of SpinTAC™ Position Plan

Background

Lab 13c adds new API function calls for SpinTAC™ Position Plan. Figure 76 shows how SpinTAC™ Position Plan connects with the rest of the SpinTAC™ components.

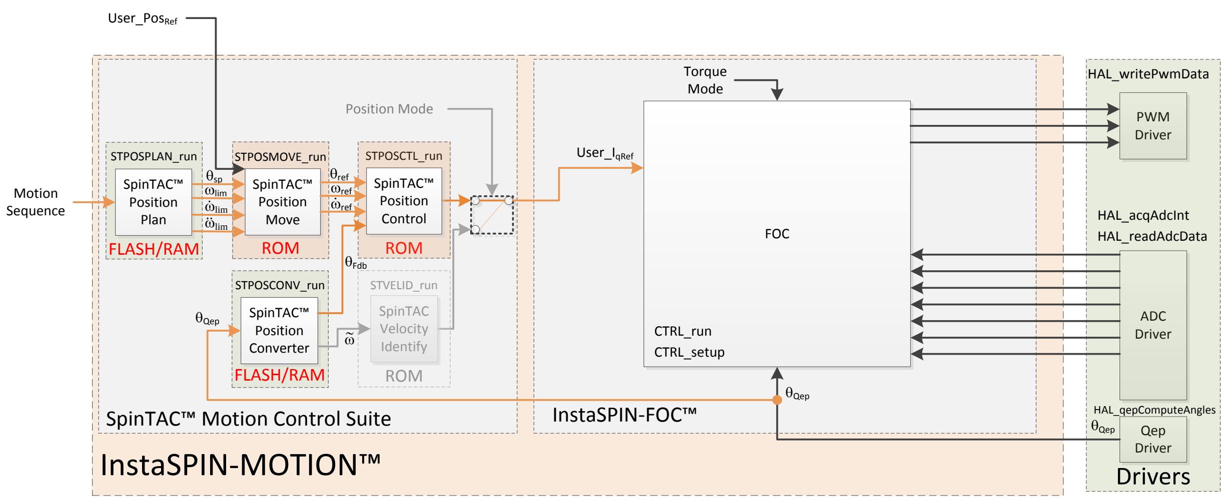


Figure 76: InstaSPIN-MOTION block diagram for lab 13c

This lab adds the SpinTAC Position Plan into the project. This block diagram shows how to integrate the SpinTAC Position Plan with the rest of the SpinTAC components. SpinTAC Position Plan accepts a motion sequence as an input and outputs the position step, velocity limit, acceleration limit, deceleration limit, and jerk limit. These get passed into SpinTAC Position Move which takes these limits and generates a profile to provide to SpinTAC Position Control.

The example motion sequence in this lab is an example of a simple motion sequence. It includes the basic features provided by SpinTAC Position Plan. Figure 77 shows the state transition map for the example motion sequence.

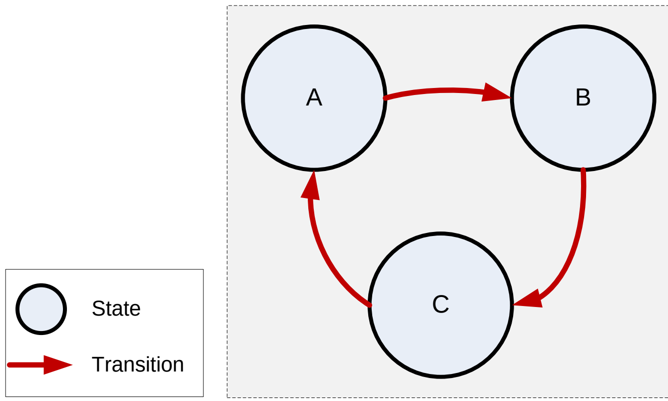


Figure 77: State Transition Map of Example Motion Sequence

This motion sequence transitions from state A to state B and onto state C as soon as the state timer has elapsed.

SpinTAC Position Plan only consumes the amount of memory that is required to configure your motion sequence. A simple motion sequence, like the one in this lab, will consume less memory than a more complicated motion sequence, like the one in the next lab. It is important that the allocated configuration array is correctly sized for your motion sequence.

Additional details around the operation and configuration of SpinTAC™ Position Plan are found in the InstaSPIN-MOTION User's Guide.

Project Files

There are no new project files.

Include the Header File

The critical header file for the SpinTAC components is spintac_position.h. This header file is common across all labs so there will be more includes in spintac_position.h than are needed for this lab.

Table 78: Important header files needed for SpinTAC components

spintac_position.h	Header file containing all SpinTAC header files used in main.c	
	SpinTAC	
	spintac_pos_plan.h	SpinTAC Position Plan structures and function declarations.

Declare the Global Structure

Lab 13c adds global variables to monitor the internal states of SpinTAC™ Position Plan and to control SpinTAC™ Position Plan. These variables provide an external interface to start, stop, and pause SpinTAC™ Position Plan. These variables are also used to store the configuration of SpinTAC Position Plan. This array needs to be declared in the user code so that it can be sized to fit the requirements of the motion sequence defined by the user.

Table 79: Global object and variable declarations for SpinTAC™ Position Plan

globals		
	SpinTAC Plan	
	ST_PlanButton_e	Used to handle controlling SpinTAC Position Plan. This defines the different states that can be set to control the operation.
	ST_POSPLAN_CFG_ARRAY_DWORDS	The MACRO define used to establish the size of the SpinTAC Position Plan configuration array. This value is calcualted based on the number of elements that will be used in the SpinTAC Position Plan configuration
	stPosPlanCfgArray	This array is used to store the SpinTAC Position Plan configuration.

Initialize the Configuration Variables

During the initialization and setup, the project will call the ST_setupPosPlan function to configure and load the motion sequence into SpinTAC Position Plan. This function is declared in the main source file for this project. A detailed explanation of the API calls in ST_setupPosPlan can be found in the InstaSPIN-MOTION User's Guide.

Table 80: InstaSPIN functions used in Initialization and Setup

Setup		
	SpinTAC	
	ST_setupPosPlan	This function calls into SpinTAC Position Plan to configure the motion sequence.

Configuring SpinTAC™ Position Plan

When the motion sequence in SpinTAC Position Plan is configured there are many different elements that build the motion sequence. The elements covered in this lab are States and Transitions. Each of these elements has a different configuration function. It is important that the configuration of SpinTAC Position Plan is done in this order. If the configuration is not done in this order it could cause a configuration error.

States

STPOSPLAN_addCfgState(Position Plan Handle, Position Step Integer [MRev], Position Step Fraction [MRev], Time in State [ISR ticks])
This function adds a state into the motion sequence. It is configured by setting the position step (in integer and fraction) that you want the motor to accomplish during this state and with the minimum time it should remain in this state.

Transition

STPOSPLAN_addCfgTran(Position Plan Handle, From State, To State, Condition Option, Condition Index 1, Condition Index 2, Velocity Limit [pu/s], Acceleration Limit [pu/s^2], Deceleration Limit [pu/s^2], Jerk Limit [pu/s^3])

This function establishes the transitions between two states. The From State and To State values describe which states this transition is valid for. The condition option specifies if a condition needs to be evaluated prior to the transition. The condition index 1 & 2 specify which conditions should be evaluated. If less than two conditions need to be evaluated, set the unused values to 0. Velocity limit sets the maximum velocity that should be used when making this transition. This value cannot exceed the maximum velocity that is configured for the motion sequence. Acceleration limit & Deceleration limit sets the acceleration & deceleration to use to transition between the From State speed and the To State speed. These values cannot exceed the acceleration & deceleration max that is configured for the motion sequence. The jerk limit sets the jerk to be used in the speed transition. This value should not exceed the jerk max that is configured for the motion sequence.

The functions used in ST_setupPosPlan are described in Table 81.

Table 81: InstaSPIN functions used in ST_setupPosPlan

ST_setupPosPlan		
	SpinTAC	
	STPOSPLAN_setCfgArray	This function provides the configuration array information into SpinTAC Position Plan
	STPOSPLAN_setCfg	This function provides the system maximum information to SpinTAC Position Plan
	STPOSPLAN_setCfgHaltState	This function provides the system halt information to SpinTAC Position Plan
	STPOSPLAN_addCfgState	This function adds a State into the SpinTAC Position Plan configuration
	STPOSPLAN_addCfgTran	This function adds a Transition into the SpinTAC Position Plan configuration
	STPOSPLAN_getErrorID	This function returns the error (ERR_ID) of SpinTAC Position Plan.

Main Run-Time loop (forever loop)

Nothing has changed in the forever loop from the previous lab.

Main ISR

The main ISR calls very critical, time dependent functions that run the SpinTAC components. The new functions that are required for this lab are listed in Table 82.

Table 82: InstaSPIN functions used in the main ISR

mainISR		
	SpinTAC	
	ST_runPosPlan	The ST_runPosPlan function calls the SpinTAC Position Plan object. This also handles enabling the SpinTAC Plan object.
	ST_runPosPlanIsr	The ST_runPosPlanIsr function calls the time-critical parts of the SpinTAC Position Plan object.

Call SpinTAC™ Position Plan

The ST_runPosPlan function has been added to the project to call the SpinTAC™ Plan component and to use it to generate motion sequences. Table 83 lists the InstaSPIN functions called in ST_runPosPlan.

Table 83: InstaSPIN functions used in ST_runPosPlan

ST_runPosPlan		
	SpinTAC Position Move	
	STPOSMOVE_getDone	This function returns if SpinTAC Position Move has completed running a profile
	SpinTAC Position Plan	
	STPOSPLAN_getErrorID	This function returns the error (ERR_ID) in SpinTAC Position Plan.
	STPOSPLAN_setEnable	This function sets the enable (ENB) bit in SpinTAC Position Plan.
	STPOSPLAN_setReset	This function sets the reset (RES) bit in SpinTAC Position Plan.
	STPOSPLAN_run	This function calls into SpinTAC Position Plan to run the motion sequence.
	STPOSPLAN_getStatus	This function returns the status (STATUS) of SpinTAC Position Plan.
	STPOSPLAN_getCurrentState	This function returns the current state (CurState) of SpinTAC Position Plan.
	STPOSPLAN_getPositionStep_mrev	This function returns the position step (PosStep_mrev) produced by SpinTAC Position Plan.
	STPOSPLAN_getVelocityLimit	This function returns the velocity limit (VelLim) produced by SpinTAC Position Plan.
	STPOSPLAN_getAccelerationLimit	This function returns the acceleration limit (DecLim) produced by SpinTAC Position Plan.
	STPOSPLAN_getDecelerationLimit	This function returns the deceleration limit (DecLim) produced by SpinTAC Position Plan.
	STPOSPLAN_getJerkLimit	This function returns the jerk limit (JrkLim) produced by SpinTAC Position Plan.

The ST_runPosPlanTick function has been added to the project to call the time-critical components of SpinTAC™ Position Plan. Table 84 lists the InstaSPIN functions called in ST_runPosPlanTick.

Table 84: InstaSPIN functions used in ST_runPosPlanTick

ST_runPosPlanIsr		
	SpinTAC	
	STPOSPLAN_runTick	This function calls into SpinTAC Position Plan to update the timer value in SpinTAC Position Plan.
	STPOSPLAN_setUnitProfDone	This function indicates to SpinTAC Position Plan that SpinTAC Position Move has completed running the requested profile.

Lab Procedure

In Code Composer, build proj_lab13c, connect to the target and load the .out file.

- Open the command file “sw\solutions\instaspin_motion\src\proj_lab13c.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

To run the motor a couple of steps are required:

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To start the current loop controller, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

To start the motion sequence, the SpinTAC™ Position Plan button needs to be set to start once the estimator state is set to online.

- Set “gMotorVars.SpinTAC.PosPlanRun” to ST_PLAN_START to begin the motion sequence.

The motor will run through a very simple motion sequence where the motor spins one revolution anti-clockwise and then spins 1 revolution clockwise.

- At the conclusion of the motion sequence “gMotorVars.SpinTAC.PosPlanDone” will be set to 1. This is done to indicate to the user program that the motion sequence has completed.

Now modify the SpinTAC™ Position Plan configuration to transition from State C to State B instead of State A.

- In the function ST_setupPosPlan, find the line highlighted in Figure 78. Change the value from STATE_A to STATE_B.

```
//Example: STPOSPLAN_addCfgTran(handle,    FromState, ToState, CondOption, CondIdx1, CondiIdx2
// NOTE: The deceleration limit must be set between the following bounds [acceleration limit,
STPOSPLAN_addCfgTran(stObj->posPlanHandle, STATE_A,   STATE_B, ST_COND_NC, 0,        0,
STPOSPLAN_addCfgTran(stObj->posPlanHandle, STATE_B,   STATE_C, ST_COND_NC, 0,        0,
STPOSPLAN_addCfgTran(stObj->posPlanHandle, STATE_C,   STATE_A, ST_COND_NC, 0,        0,
```

Figure 78: Code modification in proj_lab13c

- Recompile and download the .out file

Now the motor will not stop transitioning from anti-clockwise to clockwise until “gMotorVars.SpinTAC.PosPlanRun” is set to ST_PLAN_STOP.

Continue to explore the advanced features of SpinTAC™ Position Plan by making additional modifications to the motions sequence. Some examples are provided below.

- Add a State D to SpinTAC Plan
- Add a transition to and from State D
- Change the transitions to run the state machine from state C -> B -> A

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how easy it is to design motion sequences using SpinTAC™ Position Plan. This lab configures SpinTAC™ Position Plan to run a simple motion sequence. This lab also showcases how easy it is to modify the motion sequence and introduces the API calls that make up the SpinTAC™ Position Plan configuration.

Lab 13d - Motion Sequence Real World Example: Vending Machine

Abstract

SpinTAC™ Plan is a motion sequence planner. It allows you to easily build complex motion sequences. This will allow you to quickly implement your application's motion sequence and speed up development time. This lab provides a very complex example of a motion sequence. Additional information about trajectory planning, motion sequences, and SpinTAC™ Plan can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section Trajectory Planning).

Introduction

SpinTAC™ Plan implements motion sequence planning. It allows for you to quickly build a motion sequence to run your application. SpinTAC™ Plan features: conditional transitions, variables, state timers, and actions. This lab will use the example of a rotating vending machine to demonstrate these features.

Objectives Learned

- Use SpinTAC™ Position Plan to design a complicated motion sequence.
- Use SpinTAC™ Position Plan to run a complicated motion sequence.
- Understand the features of SpinTAC™ Position Plan

Background

This lab adds new API function calls for SpinTAC™ Position Plan. Figure 79 shows how SpinTAC™ Position Plan connects with the rest of the SpinTAC™ components.

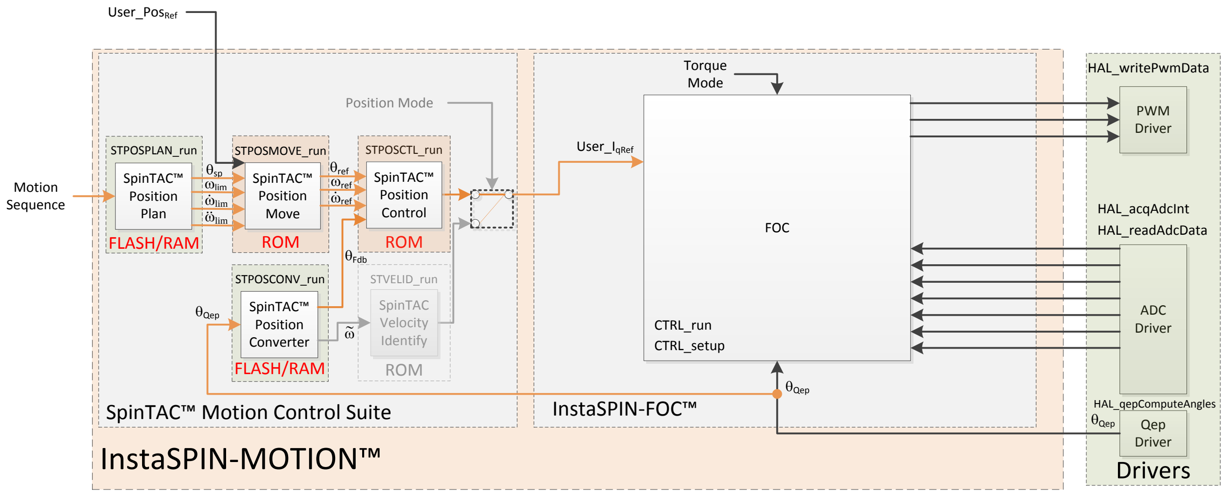


Figure 79: InstaSPIN-MOTION block diagram for lab 13d

This lab does not contain any changes from the block diagram perspective. The primary change is using a different motion sequence.

This lab contains an example of a rotating vending machine that can only dispense one item at a time. Figure 80 shows an example of this type of vending machine.

Figure 80: Example of a Rotating Vending Machine

The vending machine operates in a complex motion sequence. It features many interfaces to buttons as well as conditional state transitions. The entire motion sequence can be implemented in SpinTAC™ Position Plan. Figure 81 shows the state transition map for the vending machine.

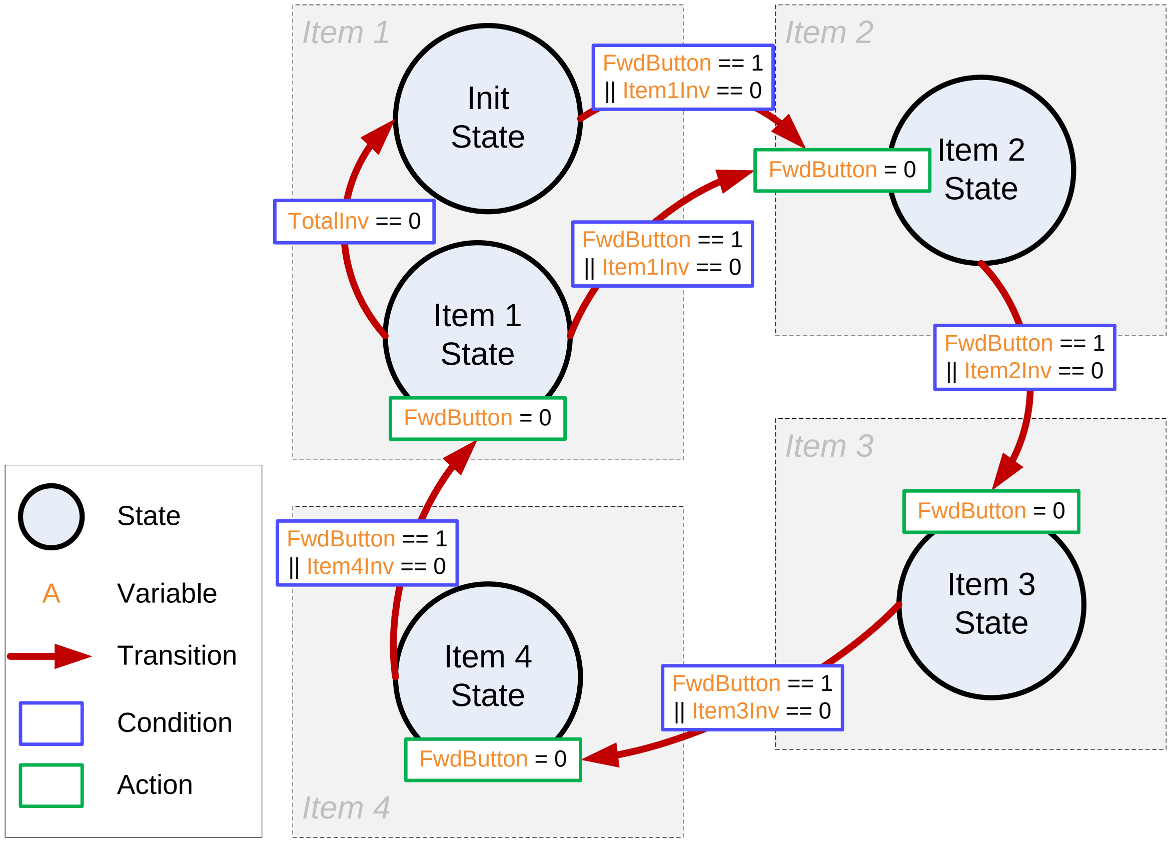


Figure 81: State Transition Map of Vending Machine Example

The vending machine dispenses 4 different types of items. The motion profile has four stages, one for each item. The vending machine will hold a position until the user indicates that the vending machine should advance to display the next item. A select button is used to dispense an item. This will reduce the inventory of that item. When an item's inventory reaches zero the vending machine will automatically bypass that item. When all of the items' inventories have been reduced to zero, the vending machine will return to the Init state, which completes the motion sequence.

Additional information about trajectory planning, motion sequences, and SpinTAC™ Position Plan can be found in the InstaSPIN-FOC and InstaSPIN-MOTION User Guide (see section Trajectory Planning).

Project Files

There are no new project files.

Include the Header File

There are no new includes.

Declare the Global Structure

This lab adds global variables to monitor the internal states of SpinTAC™ Position Plan and to control SpinTAC™ Position Plan. These variables provide an external interface to start, stop, and pause SpinTAC™ Position Plan. These new variables are covered in Table 85.

Table 85: Global object and variable declarations in SpinTAC Plan

globals		
	Vending Machine	
	gVendFwdButton	Advances the vending machine position
	gVendSelectButton	Selects an item from the vending machine, decrements the inventory
	gVendInventory	Array which holds the inventory of each item in vending machine
	gVendAvailableItem	Displays the item that is currentlly available to vend

Initialize the Configuration Variables

There are no new function calls used to setup SpinTAC Position Plan. However the contents of ST_setupVelPlan have been modified to run the example vending machine motion sequence.

Configuring SpinTAC™ Position Plan

When the motion sequence in SpinTAC Position Plan is configured there are many different elements that build the motion sequence. These elements are States, Variables, Conditions, Transitions, and Actions. Each of these elements has a different configuration function. It is important that the configuration of SpinTAC Position Plan is done in this order. If the configuration is not done in this order it could cause a configuration error.

States

STPOSPLAN_addCfgState(Position Plan Handle, Position Step Integer [MRev], Position Step Fraction [MRev], Time in State [ISR ticks])
This function adds a state into the motion sequence. It is configured by setting the position step (in integer and fraction) that you want the motor to accomplish during this state and with the minimum time it should remain in this state.

Variables

STPOSPLAN_addCfgVar(Position Plan Handle, Variable Type, Initial Value)
This function establishes a variable that will be used in the motion sequence. The variable type determines how SpinTAC™ Plan can use this variable. The initial value is the value that should be loaded into this variable initially. The variable can be up to a 32-bit value.

Conditions

STPOSPLAN_addCfgCond(Position Plan Handle, Variable Index, Comparison, Comparison Value 1, Comparison Value 2)
This function sets up a condition to be used in the motion sequence. This will be a fixed comparison of a variable against a value or value range. The variable index describes which variable should be compared. The comparison should be used to describe the type of comparison to be done. Comparison values 1 & 2 are used to establish the bounds of the comparison. If a comparison only requires one value it should be set in comparison value 1 and comparison value 2 should be set to 0.

Transition

STPOSPLAN_addCfgTran(Position Plan Handle, From State, To State, Condition Option, Condition Index 1, Condition Index 2, Velocity Limit [pu/s], Acceleration Limit [pu/s^2], Deceleration Limit [pu/s^2], Jerk Limit [pu/s^3])

This function establishes the transitions between two states. The From State and To State values describe which states this transition is valid for. The condition option specifies if a condition needs to be evaluated prior to the transition. The condition index 1 & 2 specify which conditions should be evaluated. If less than two conditions need to be evaluated, set the unused values to 0. Velocity limit sets the maximum velocity that should be used when making this transition. This value cannot exceed the maximum velocity that is configured for the motion sequence. Acceleration limit & Deceleration limit sets the acceleration & deceleration to use to transition between the From State speed and the To State speed. These values cannot exceed the acceleration & deceleration max that is configured for the motion sequence. The jerk limit sets the jerk to be used in the speed transition. This value should not exceed the jerk max that is configured for the motion sequence.

Actions

STPOSPLAN_addCfgAct(Position Plan Handle, State Index, Condition Option, Condition Index 1, Condition Index 2, Variable Index, Operation, Value, Action Trigger)

This function adds an action into the motion sequence. The state index describes which state the action should take place in. The condition option specifies if a condition needs to be evaluated prior to the action. The condition index 1 & 2 specify which conditions should be evaluated. If less than two conditions need to be evaluated, set the unused values to 0. The variable index indicates which variable the action should be done to. The operation determines what operation should be done to the variable, the only available options are to add a value or set a value. The value is what should be added or set to the variable. The action trigger indicates if the action should be performed when entering or exiting the state.

This function has been modified to configure SpinTAC Position Plan to run the motion sequence of a washing machine. There are new function calls in order to take advantage of the advanced features of SpinTAC Position Plan. The new functions are described in Table 86.

Table 86: InstaSPIN functions used in ST_setupPosPlan

ST_setupPosPlan		
	SpinTAC	
	STPOSPLAN_addCfgVar	This function adds a Variable into the SpinTAC Position Plan configuration
	STPOSPLAN_addCfgCond	This function adds a Condition into the SpinTAC Position Plan configuration
	STPOSPLAN_addCfgAct	This function adds a Action into the SpinTAC Position Plan configuration

Main Run-Time loop (forever loop)

Nothing has changed in the forever loop from the previous lab.

Main ISR

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

Call SpinTAC™ Position Plan

The ST_runPosPlan function has been updated to interface with the external components that make up the buttons of the simulated vending machine. Table 87 lists the functions used to interface with external components in the ST_runPosPlan function.

Table 87: InstaSPIN functions used in ST_runPosPlan

ST_runPosPlan		
	SpinTAC	
	STPOSPLAN_getVar	This function returns the value of a variable in SpinTAC Position Plan
	STPOSPLAN_setVar	This function sets the value of a variable in SpinTAC Position Plan
	STPOSPLAN_getCurrentState	This function returns the current state of the Motion Sequence being executed

Lab Procedure

In Code Composer, build proj_lab13d, connect to the target and load the .out file.

- Open the command file “sw\solutions\instaspin_motion\src\proj_lab13d.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

To run the motor a couple of steps are required:

- To start the project, set the variable “gMotorVars.Flag_enableSys” equal to 1.
- To start the current loop controller, set the variable “gMotorVars.Flag_Run_Identify” equal to 1.

To start the motion sequence, the SpinTAC™ Position Plan button needs to be set to start once the estimator state is set to online.

- Set “gMotorVars.SpinTAC.PosPlanRun” to ST_PLAN_START to begin the motion sequence.

The motor will run through a procedure that is designed to emulate a vending machine. It will hold the current position until you set “gVendFwdButton” to 1. This will alert the vending machine that the user wants to advance the machine one location. If you wish to remove an item from the vending machine set “gVendSelectButton” to 1. This will remove an item from the inventory. When the inventory for a specific item has been reduced to 0, the vending machine will bypass that location.

To stop the motion sequence, set “gMotorVars.SpinTAC.PosPlanRun” to ST_PLAN_STOP.

Now modify the SpinTAC™ Plan configuration to automatically advance items until the user clears that “gVendFwdButton.”

- In the function ST_setupPosPlan, find the lines highlighted in Figure 82. Comment out each line. This will not reset the value for “gVendFwdButton” when exiting a state.

```
//Example: STPOSPLAN_addCfgAct(handle,    StateIdx,   CondOption, CondIdx1, CondIdx2, VarIdx,   Operation, Value, ActionTriger);
STPOSPLAN_addCfgAct(stObj->posPlanHandle, VEND_ITEM0, ST_COND_NC, 0,        0,        VEND_Fwd, ST_ACT_EQ, 0,     ST_ACT_ENTR);   // In Item0, clear Fwd Button
STPOSPLAN_addCfgAct(stObj->posPlanHandle, VEND_ITEM1, ST_COND_NC, 0,        0,        VEND_Fwd, ST_ACT_EQ, 0,     ST_ACT_ENTR);   // In Item1, clear Fwd Button
STPOSPLAN_addCfgAct(stObj->posPlanHandle, VEND_ITEM2, ST_COND_NC, 0,        0,        VEND_Fwd, ST_ACT_EQ, 0,     ST_ACT_ENTR);   // In Item2, clear Fwd Button
STPOSPLAN_addCfgAct(stObj->posPlanHandle, VEND_ITEM3, ST_COND_NC, 0,        0,        VEND_Fwd, ST_ACT_EQ, 0,     ST_ACT_ENTR);   // In Item3, clear Fwd Button
```

Figure 82: Code modification to allow automatically advancing

- Recompile and download the .out file

When you run this modified motion sequence it will automatically advance the item until the user manually sets “gVendFwdButton” to 0.

Continue to explore the advanced features of SpinTAC™ Position Plan by making additional modifications to the motions sequence. Some examples are provided below.

- Add a fifth item to the vending machine

- Add a button that increments the inventory, for when the vending machine gets restocked

When done experimenting with the motor:

- Set the variable “Flag_Run_Identify” to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how easy it is to design complex motion sequences using SpinTAC™ Position Plan. This lab configures SpinTAC™ Position Plan to run a washing machine profile that features complex elements. This lab also showcases how easy it is to modify the motion sequence and introduces the API calls that make up the SpinTAC™ Position Plan configuration.

Lab 13e - Smooth Velocity Transitions in Position Control

Abstract

In addition to providing smooth position transitions, SpinTAC Position Move can also provide smooth speed transitions while still operating in a position control system. This lab demonstrates how to configure SpinTAC Position Move to generate speed transitions in position mode.

Introduction

InstaSPIN-MOTION's position profile generator can also generate velocity profiles. These velocity profiles are time-optimal curves within the user defined acceleration, deceleration, and jerk bounds. It supports basic ramp profile, as well as advanced s-curve and st-curve (Linestream Proprietary) curves. The proprietary st-curve features a continuous jerk to provide additional smoothing on the trajectory.

Prerequisites

This lab assumes that the system inertia has been identified and the SpinTAC Position Control has been tuned.

Objectives Learned

- Use SpinTAC Position Move to transition between speeds.
- Become familiar with the bounds that can be adjusted as part of SpinTAC Position Move
- Continue exploring how SpinTAC Position Control takes advantage of the advanced features of SpinTAC Position Move for speed transitions

Background

This lab adds new API functions calls to call SpinTAC Position Move. Figure 83 shows how SpinTAC Position Move connects with the rest of the SpinTAC components.

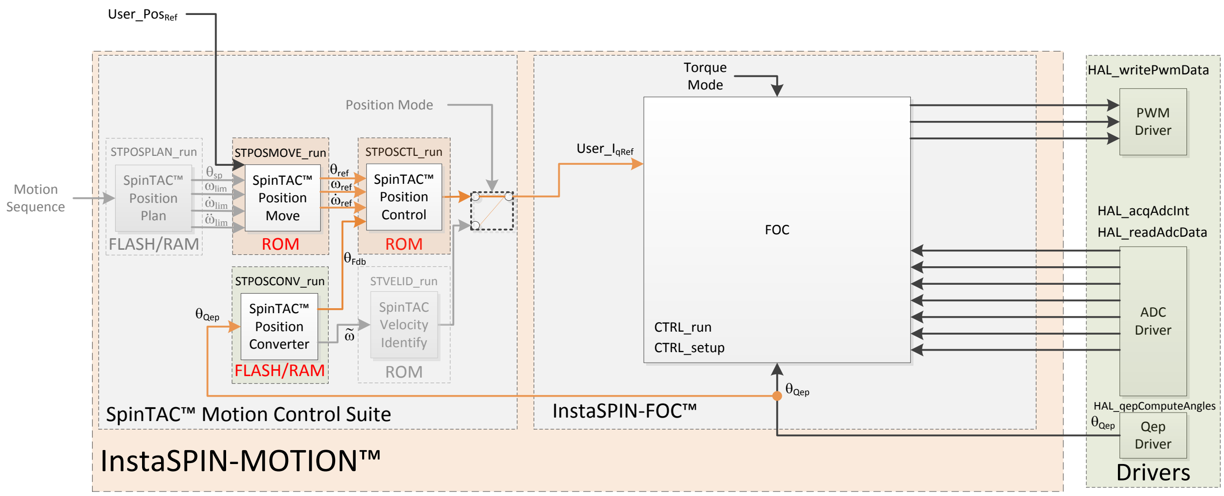


Figure 83: InstaSPIN-MOTION block diagram for lab 13e

This lab modifies the configuration of SpinTAC Position Move in order to supply velocity profiles instead of position profiles. To emulate velocity mode, SpinTAC Position Move will generate a position reference that increases at the same rate as the goal speed of a velocity profile.

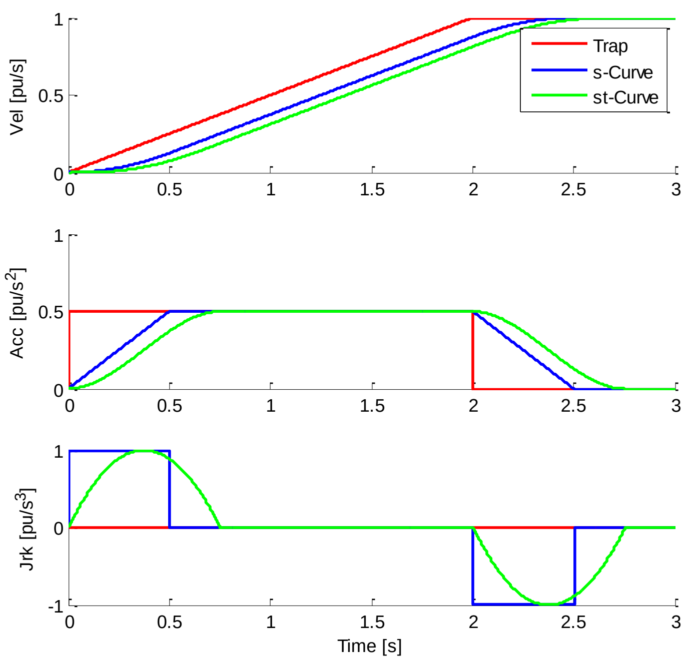


Figure 84: Velocity Trajectory Curves Available from SpinTAC™ Position Move

Figure 84 illustrates the differences between the three curve types available in SpinTAC™ Position Move. The st-curve represents the smoothest motion, which is critical for systems that are sensitive to large amounts of jerk. Jerk represents the rate of change of acceleration. A larger jerk will increase the acceleration at a faster rate. Steps, or sharp movement between two speeds, can cause systems to oscillate. The bigger the step in speed, the greater this tendency for the system to oscillate. Control over jerk can round the velocity corners, reducing oscillation. As a result, acceleration can be set higher. Controlling the jerk in your system will lead to less mechanical stress on your system components and can lead to better reliability and less failing parts.

Signals	Trapezoidal	S-Curve	st-Curve
Velocity	Continuous	Smooth	Smooth
Acceleration	Bounded	Continuous	Smooth
Jerk	N/A	Bounded	Continuous

Figure 85: Chart describing curve characteristics

Figure 85 shows the different characteristics of the three velocity curve types provided by SpinTAC Position Move. St-curve provides the smoothest motion by smoothing out the acceleration of the profile. For most applications the st-curve represents the best motion profile.

Project Files

There are no new project files in this project.

Include the Header File

The critical header file for the SpinTAC position components is spintac_position.h. This file contains the code required to configure SpinTAC Position Move. This header file is common across all position labs so there will be more includes than are needed for this lab. There are no new includes.

Declare the Global Structure

There are no new global object and variable declarations.

Initialize the Configuration Variables

During the initialization and setup of the project the SpinTAC Position Move needs to be configured. This is done the same as in Lab 13b.

Main Run-Time loop (forever loop)

One change has been made to the forever loop of this lab. SpinTAC Position Move should be held in reset until the Estimator state has advanced to OnLine. This is done so that it will not begin generating references until the motor system is ready.

Main ISR

There are no new functions calls in the Main ISR. The Main ISR is the same as Lab 13b.

Call SpinTAC Position Move

The new functions required to run SpinTAC Position Move are listed in Table 88. The difference between Lab 13b & Lab 13e is that we will be looking for a different trigger to enable SpinTAC Position Move, will be configuring a different type of profile, and will be setting a goal velocity and not a goal position.

Table 88: InstaSPIN functions used in ST_runPosMove

ST_runPosMove		
	SpinTAC	
	STPOSMOVE_getVelocityEnd	This function returns the goal velocity [pu/s] of SpinTAC Position Move
	STPOSOVE_setProfileType	This function sets the profile type in SpinTAC Position Move
	STPOSMOVE_setVelocityEnd	This function sets the goal velocity [pu/s] in SpinTAC Position Move

Lab Procedure

In Code Composer, build proj_lab13e. Start a Debug session and download the proj_lab13e.out file to the MCU.

- Open the command file "sw\solutions\instaspin_motion\src\proj_lab13e.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

To run the motor a couple of steps are required:

- To start the project, set the variable "gMotorVars.Flag_enableSys" equal to 1.
- To start the current loop controller, set the variable "gMotorVars.Flag_Run_Identify" equal to 1.

SpinTAC Position Move will generate motion profiles every time the velocity goal is updated.

- Set "gMotorVars.MaxVel_krpm" to "1.0" to make the motor rotate at 1000 rpm

The motor will begin spinning at 1000 rpm. To get the motor to accelerate much faster you need to increase the acceleration, and jerk limits.

- "gMotorVars.MaxAccel_krpmps" and "gMotorVars.MaxDecel_krpmps" configure the acceleration & deceleration used in the profile. Set these values to 75.0. This is the maximum value for acceleration & deceleration in this project.
- Set "gMotorVars.MaxVel_krpm" to "-1.0"

The motor accelerated a little faster, but not a lot faster. This is because we have not yet modified the jerk limit.

- "gMotorVars.MaxJerk_krpmps2" configures the jerk used in the profile. Set this value to 400.0. This is the maximum value for jerk in this project.
- Set "gMotorVars.MaxVel_krpm" to "1.0"

SpinTAC Move supports three different curve types: trapezoid, s-curve, and st-curve. This curve can be selected by changing "gMotorVars.SpinTAC.PosMoveCurveType." The differences between the three curves are discussed in detail in the InstaSPIN-MOTION User's Guide.

SpinTAC Move will alert the user when it has completed a profile via the done bit. When the profile is completed, "gMotorVars.SpinTAC.PosMoveDone" will be set to 1. This could be used in a project to alert the system when the motor has completed a movement.

When done experimenting with the motor:

- Set the variable "Flag_Run_Identify" to false to turn off the pwms to the motor.
- Turn off real-time control and stop the debugger.

Conclusion

This lab showed how easy it is to use SpinTAC Position Move to generate constraint-based, time-optimal velocity profiles. This operating mode can be easily mixed with the typical position profile mode of operation. This lab also shows the different curves that can be used with SpinTAC Position Move. The st-curve provides a continuous jerk profile that will enable very smooth motion for jerk sensitive applications.

Lab 13f – Dual Motor Sensored Position InstaSPIN-MOTION

Abstract

Please see code comments for implementation details.

Writeups will not be provided for InstaSPIN-MOTION proj_lab6e, proj_lab12c nor proj_lab13f. Please review the code to understand the technique and important variables employed in these labs.

A writeup for dual motor control using InstaSPIN-FOC is provided in the proj_lab10d writeup.

Proj_lab11d is a valid dual motor control project for InstaSPIN-FOC based on the simplified structure of proj_lab11. Labs 6e, 12c, and 13f are InstaSPIN-MOTION projects that build upon the lab 11d code base.

Please note that proj_lab11d does not include a trajectory module; as such, there is no acceleration variable, and any changes the speed reference are applied as a step input. Use caution. InstaSPIN-

Lab 20 – New ctrl Structure

Abstract

To provide easier access to the FOC elements of the control, a new ctrl structure has been created. The previous MotorWare ctrl included all of the modules used to implement FOC. The new ctrl only contains the PI controllers, i.e. the speed, Id, and Iq controllers. The rest of the FOC modules are located in the mainISR.

Introduction

In previous versions of MotorWare, the ctrl object contains the Park, Clarke, PI, and SVGEN modules. All of the FOC control algorithm is implemented in the ctrl object. Proj_lab20 introduces a new version of ctrl where only the PI controllers for speed, Id, and Iq are implemented. The rest of the FOC control objects are implemented in the main ISR.

Prerequisites

Assumes knowledge of up to proj_lab05b.

Objectives Learned

This project is only used to introduce the new control structure

Background

Old ctrl

The original version of the motor control system is shown in Figure 86. The advantage of this ctrl version is that every object is contained in one place and there is not much code in the main ISR. Disadvantages of this ctrl implementation is that if any signal needs to be added in between modules or a new module needs to be added the ctrl module has to be changed.

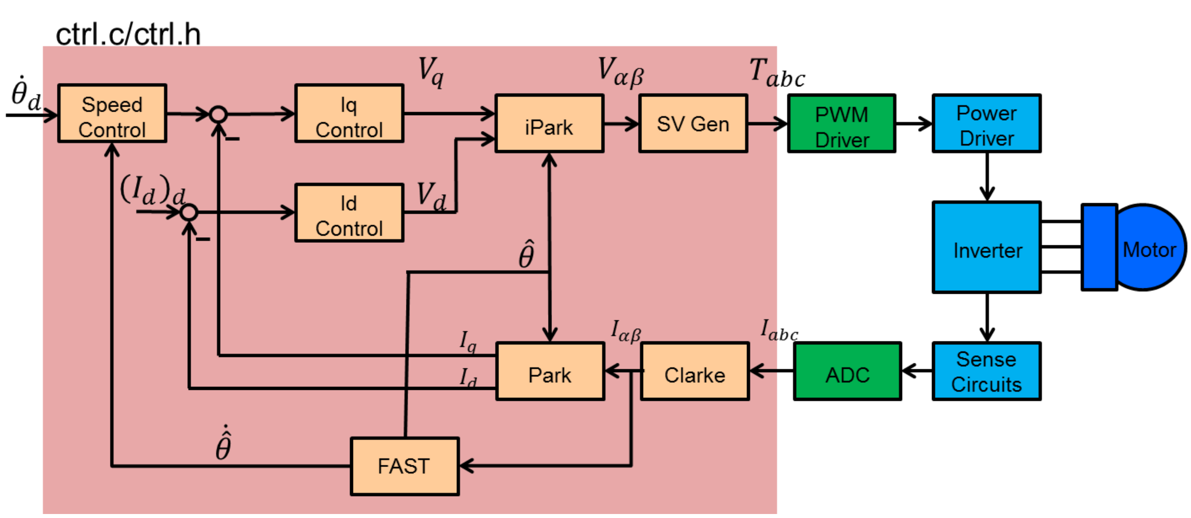


Figure 86: Old ctrl implementation that includes all of the FOC control modules.

New ctrl

The new ctrl version is shown in Figure 87. This ctrl version implements only the PI speed, Id, and Iq controllers. The advantage of this implementation is that it is easier to add new modules into the FOC control. The new ctrl will be used to implement the IPD, HFI, and AFSEL modules in project 21.

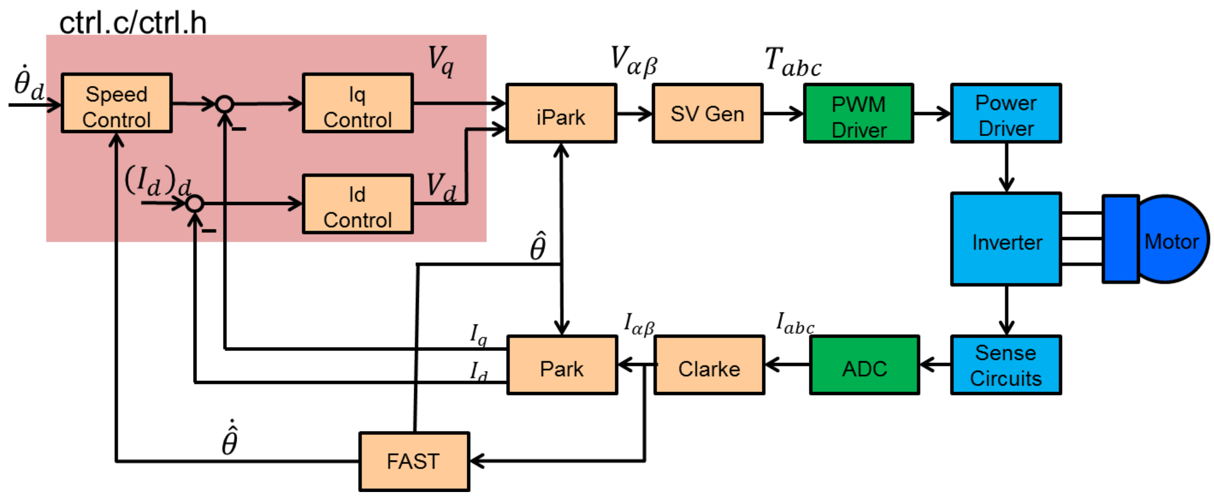


Figure 87: The new ctrl version that only contains the PI speed, Id, and Iq controllers. The rest of the FOC modules are implemented in the main ISR.

Updates with the new ctrl

Isolate the FAST estimator

The FAST estimator is now brought out into the main ISR as shown below. Now there is the ability to bring in other estimators to the FOC system like IPD and HFI.

```
{
  // run the estimator
  EST_run(estHandle, \
          &Iab_pu, \
          &Vab_pu, \
          gAdcData.dcBus, \
          speed_ref_pu);

  flag_enableSpeedCtrl = EST_doSpeedCtrl(estHandle);
  flag_enableCurrentCtrl = EST_doCurrentCtrl(estHandle);
}
```

Easy access to Idq and Vdq variables

The Idq and Vdq variables are now changed by an offset value. If field weakening is needed, `Idq_offset_pu.value[0]` can be set directly. When using IPD and HFI, `Vdq_offset_pu.value`'s are set directly.

```
  // set the offset based on the Id trajectory
Idq_offset_pu.value[0] = TRAJ_getIntValue(((CTRL_Obj *)ctrlHandle)->trajHandle_Id);
Idq_offset_pu.value[1] = _IQ(0.0);

Vdq_offset_pu.value[0] = 0;
Vdq_offset_pu.value[1] = 0;
```

CTRL_setup_user

A new function `CTRL_setup_user()` has been created to update the real time variables to the ctrl object.

```
CTRL_setup_user(ctrlHandle,
                angle_pu,
                speed_ref_pu,
                speed_pu,
                speed_outMax_pu,
                &Idq_offset_pu,
                &Vdq_offset_pu,
                flag_enableSpeedCtrl,
                flag_enableCurrentCtrl);
```

The parameters that are sent to CTRL_setup_user() are described below.

- Angle_pu – The d-axis angle (in per unit).
- Speed_ref_pu – The reference speed that is directly written to the speed PI controller (in per unit).
- Speed_pu – the speed that is estimated from the estimator (in per unit).
- Speed_outMax_pu – The output maximum of the speed PI controller (in amps per unit).
- *Idq_offset_pu – The Id and Iq array pointer of the measured Id and Iq values (in amps per unit).
- *Vdq_offset_pu – The Vd and Vq array pointer of the measured Vd and Vq values (in volts per unit).
- Flag_enableSpeedCtrl – Boolean flag to enable or disable the PI speed controller. If the speed controller is disabled, the Iq reference is set to zero and the Iq reference is set by the Iq_offset value above.
- Flag_enableCurrentCtrl – Boolean flag to enable or disable the PI Id and Iq controllers.

CTRL_runPiOnly

The function that is used to run the controller is `CTRL_runPiOnly(ctrlHandle)`.

Conclusion

A new ctrl object has been introduced. The new ctrl technique only contains the PI speed, Id, and Iq controllers. Now most of the objects used to implement FOC are brought out to the main ISR. Advantages of this ctrl implementation are easier access to variables inside of the FOC controller.

Lab 21 – Initial Position Detection and High Frequency Injection

Abstract

Signals are injected into the motor to find the d-axis initial position when the power is first applied to the motor control. After initial position detection, a frequency much higher than the motor's operating frequency range is injected to allow for zero speed control of the motor.

Introduction

In many applications, the rotor must only turn in the commanded direction immediately after power is applied to the controller. Initial position detection is performed by injecting signals into the motor, at power up, that use the interaction between the BH curve of the stator iron and the permanent magnet pole faces of the rotor to determine the d-axis of the motor. Once the d-axis is determined, high frequency signals are injected into the motor to stay locked onto an inductance saliency of the motor. The high frequency controller runs the motor at low speeds including zero speed. As the motor spins faster, there has to be a smart transition between the high frequency injection (HFI) technique and FAST. Two modules will be discussed, the IPD_HFI module which perform initial position detection and high frequency injection and the second module AFSEL which transitions control between HFI and FAST.

Prerequisites

This lab assumes that the motor's parameters are known and that the new ctrl structure talked about in lab 20 is understood.

Objectives Learned

- Learn where the IPD_HFI and AFSEL modules are added into lab 20 code structure.
- Tune IPD_HFI to perform initial position detection and high frequency injection.
- Tune AFSEL to transition between HFI and FAST smoothly.

Background

FAST will control a motor to low speeds but the motor must be spinning as shown in Figure 88. Initial position detection (IPD) at startup and (HFI) below "Minimum" speeds extend the speed operation range of FAST.

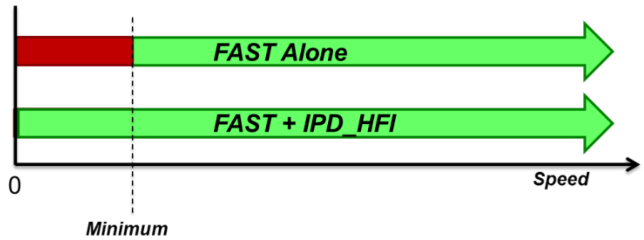


Figure 88: The low speed limit of FAST only and how IPD_HFI plus FAST will improve the total speed range.

The magnetic field strength will bias the stator's BH curve operating point as shown in Figure 89. Supporting and opposing magnetic fields are applied with the stator coil. When both fields add, the BH curve is pushed further into saturation. The BH curve operating point moves further into the linear region when the magnetic fields oppose. The difference in inductance between these two BH curve operating points allows the motor controller to determine where the rotor north pole is located.

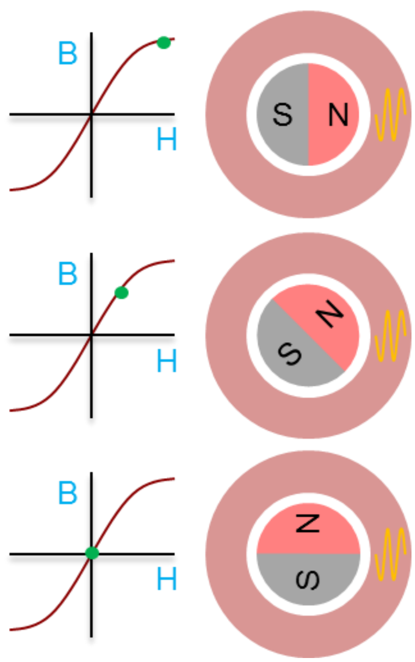


Figure 89: The permanent magnet of the rotor biases the BH curve of the stator iron which can be used to find the location of the north pole.

The IPD portion of the IPD_HFI module uses the BH curve of the iron that the stator coil is wrapped around to determine the north pole of the rotor and thus the d-axis.

Once the rotor's north pole is located, it must be tracked at all times during the operation of the motor. There are two basic kinds of permanent magnet synchronous motor designs, salient and non-salient.

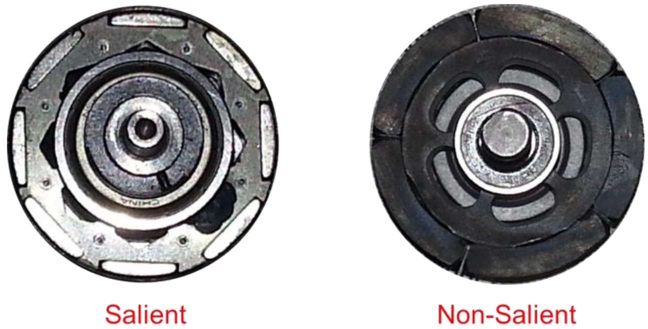


Figure 90: Salient verses non-salient rotor designs. The magnet material has much less relative permeability than iron.

Figure 90 shows two different rotor designs. The salient design has the magnets embedded in the iron. Because the magnetic material has a much less relative permeability than the surrounding iron, the reluctance difference for flux flowing through the magnet is greater than reluctance of the iron path. As the rotor's angle advances, the reluctance has a periodic variation. If the inductance is measured on a coil of the stator, it will look something like that shown in Figure 91.

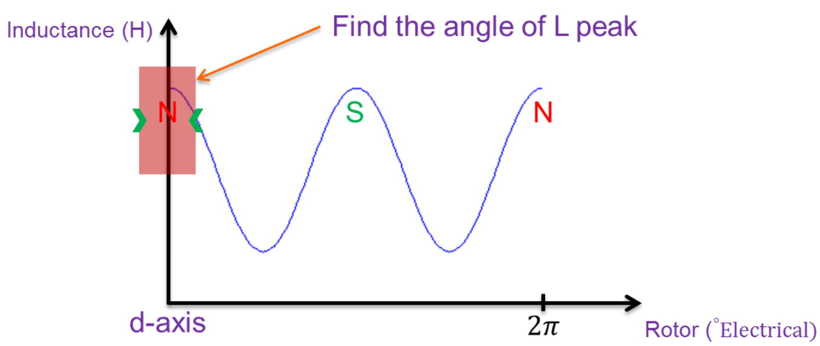


Figure 91: Inductance variation of a salient motor as the rotor angle advances.

This information can be used to find the location of the north pole of the rotor. The HFI part of IPD_HFI uses this information from a salient motor to stay locked onto the north pole of the rotor while it is spinning at low speeds. The only problem is that the south pole can as easily be locked onto as it also has an inductance peak. To make certain that its angle is locked onto the north pole, the HFI algorithm is initialized by the IPD during initial power up of the motor control.

The HFI algorithm works very well at low speeds but it has a maximum speed limit. Before this maximum speed limit is reached, control has to be handed over to a high speed observer. FAST is used as the high speed observer. The module that selects between low speed (HFI) and high speed (FAST) observers is automatic frequency select (AFSEL). Tuning parameters will be discussed in more detail later but at the least, AFSEL requires angle and frequency inputs from both the low and high speed estimators and the speed at which the control is passed from one estimator to the other. Figure 92 shows where the IPD_HFI and AFSEL modules fit into the FOC system.

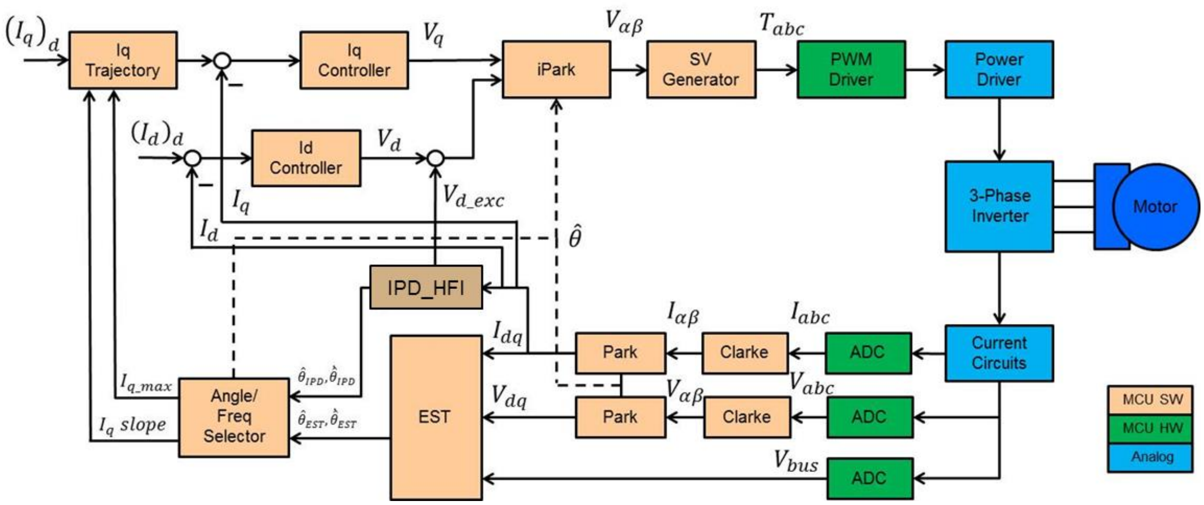


Figure 92: The block diagram of the whole IPD_HFI, FAST, and AFSEL control system.

Software Files

The module files are located in the modules directory under ipd_hfi and afsel, the project is in the solutions/instaspin_foc folder, see Figre 93 below.

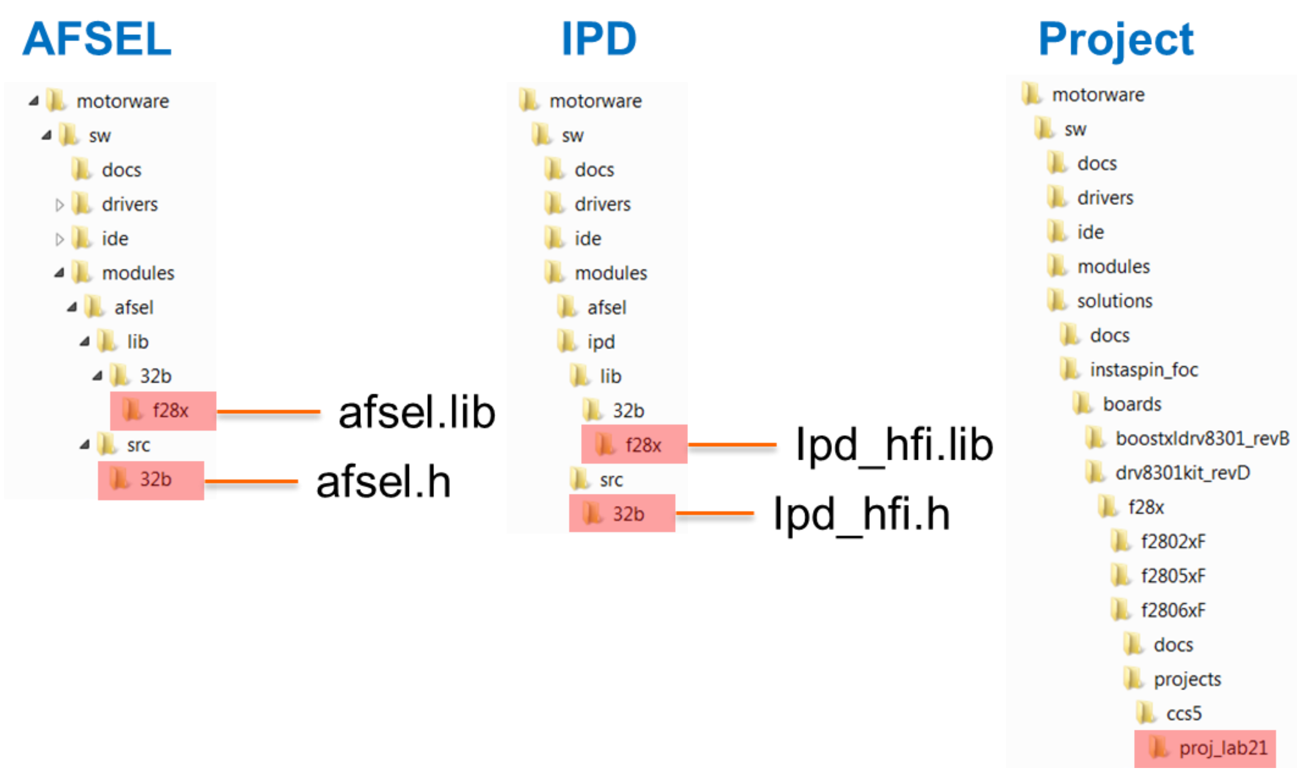


Figure 93: File locations for ipd_hfi, afsel and the project in MotorWare.

IPD_HFI Parameter Explanation

All of the tuning parameters for IPD, HFI and AFSEL for proj_lab21 are located in the user.h file. The parameters used for IPD and HFI are shown in Figure 95. An explanation of the parameters is shown graphically in Figure 94 and in the list below:

IPD and HFI parameters that need to be tuned

- excFreq – The HFI injection frequency that depends on the motor time constant
- Kspd – Gain that determines the time to converge to the d-axis angle
- excMag_coarse – Magnitude of the injected frequency for finding the rotor's north pole
- excMag_fine – Injected frequency magnitude that is used the whole time ipd_HFI is enabled
- waitTime_coarse – Time to find the north pole within 180degrees during ipd_hfi startup
- waitTime_fine – Time to find the north pole accurately during ipd_hfi startup

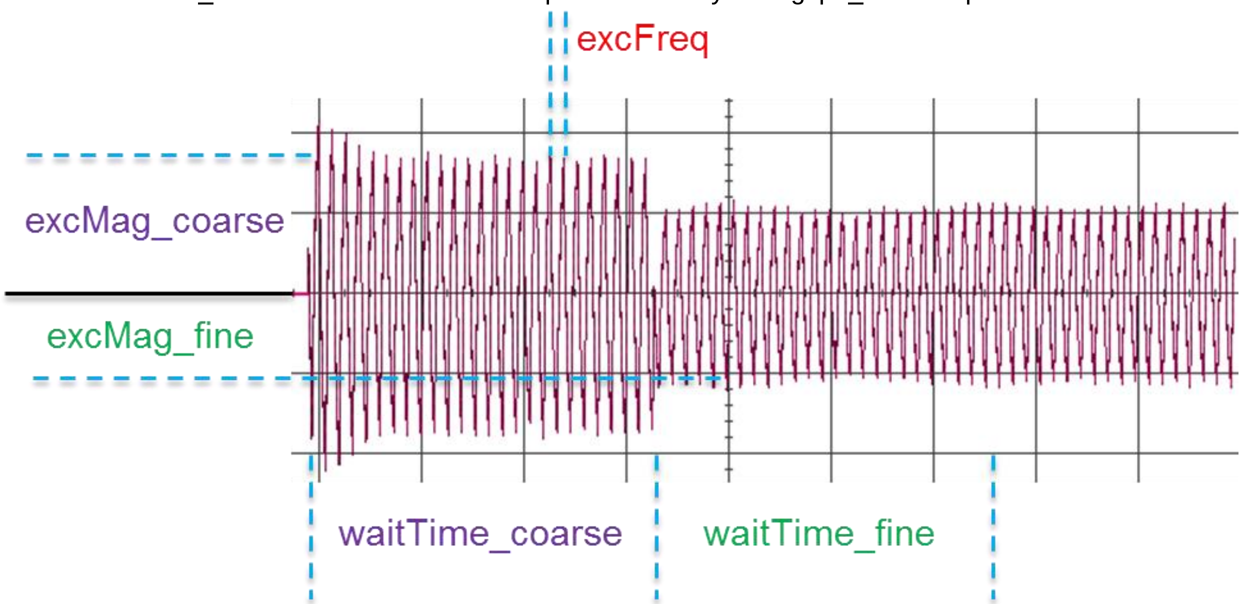


Figure 94: Showing the IPD and HFI tuning parameters graphically.

```
//! \brief     Sets the initial position detection, high frequency injection (IPD_HFI) parameters
//! \param[in] handle                   The initial position detection, high frequency injection (IPD_HFI) handle
//! \param[in] estFreq_Hz               The estimation frequency of the IPD algorithm, Hz
//! \param[in] excFreq_Hz               The excitation frequency of the IPD algorithm, Hz
//! \param[in] lpFilterCutOffFreq_Hz    The lowpass filter cutoff frequency, Hz
//! \param[in] hpFilterCutOffFreq_Hz    The highpass filter cutoff frequency, Hz
//! \param[in] iqFullScaleFreq_Hz       The IQ full scale frequency, Hz
//! \param[in] Kspd                     The speed gain value
//! \param[in] excMag_coarse_pu         The excitation magnitude during coarse position detection, Hz
//! \param[in] excMag_fine_pu           The excitation magnitude during fine position detection, Hz
//! \param[in] waitTime_coarse_sec      The wait time for coarse position detection, sec
//! \param[in] waitTime_fine_sec        The wait time for fine position detection, sec
extern void IPD_HFI_setParams(IPD_HFI_Handle handle,
                              float_t estFreq_Hz,
                              float_t excFreq_Hz,
                              float_t lpFilterCutOffFreq_Hz,
                              float_t hpFilterCutOffFreq_Hz,
                              float_t iqFullScaleFreq_Hz,
                              float_t Kspd,
                              float_t excMag_coarse_pu,
                              float_t excMag_fine_pu,
                              float_t waitTime_coarse_sec,
                              float_t waitTime_fine_sec);
```

Figure 95: IPD parameters used to tune the IPD and HFI algorithms to a motor.

IPD_HFI Tuning Example

At the time this project was written, salient motors are difficult to buy. A motor that will be used in this example was found to have saliency, a picture of its rotor is shown in Figure 90 on the left. It is an Anaheim Automation BLY341S-24V-3000. Tuning starts with excitation frequency.

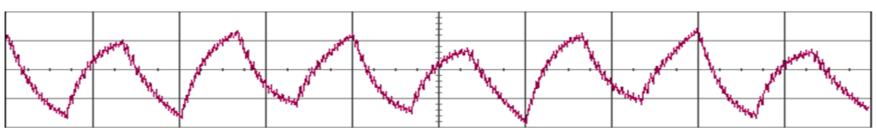
Excitation Frequency (excFreq_Hz)

The excitation frequency is dependent on the time constant of the motor's stator. The excitation frequency is found below for the Anaheim motor:

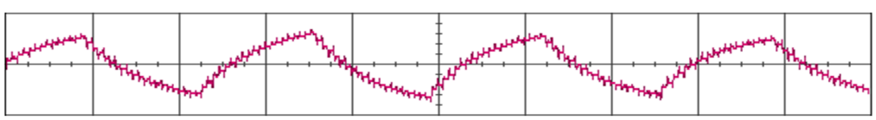
- ExcFreq_Hz < ${}^{R}/_{L} \cdot 10.0 = {}^{0.12}/_{230\mu} \cdot 10.0 = 830.0\ Hz$
- Choose the excitation frequency to be 830.0 Hz, but …
    - The actual excitation frequency is determined by: $\text{ExcFreq} = \frac{ISR}{2} \cdot \frac{1}{IntVal}$
    - The ISR = 15KHz, so the ExcFreq can be these values:
        - ExcFreq = 682, 750, 833, 937.5, etc.…
        - Choose 750 Hz for the excitation frequency
- excFreq_Hz = 750 Hz for the Anaheim motor

Course Excitation Frequency Magnitude (excMag_coarse)

- The excitation magnitude is in volts per unit
- It is best to use a current probe when adjusting this voltage level
- As described in Figure 89, the magnitude must be large enough to put the BH curve of the stator iron into saturation. Caution must be exercised so the iron is not saturated too much. The figures below illustrate when the current is too high and just right.
- excMag_coarse = 0.3 pu for the Anaheim motor



Too High: The current in the inductor cannot be extinguished from cycle to cyle and the waveform is not uniform.



Correct: The current is uniform and is allowed enough time to saturate the motor's stator iron. ExcMag_course = 0.3 PU for Anaheim Motor

Fine Excitation Frequency Magnitude (excMag_fine)

The fine excitation magnitude only needs to find the reluctance change of the motor, so it should not put the motor into saturation.

- The excitation is in Volts PU
- Usually smaller in magnitude than Excitation Magnitude Course
- Tuning is done on a trial and error basis with these tips:
  - Increase the excitation magnitude to get a better signal.
  - Do not increase too far:
    - The excitation magnitude is a voltage that is added to the output voltage of the inverter. The higher it is, the less torque capability and high speed transition that will be available for the motor.
- After all IPD tuning is finished:
  - Adjust the magnitude to get the most locked rotor torque.
- ExcMag_fine = 0.2 for Anaheim Motor

Wait Times (waitTime_coarse_sec, waitTime_fine_sec)

The wait times are very important to allow the IPD algorithm to start the motor efficiently

- WaitTime_course is the time that ExcMag_course is applied to find the hemisphere where the north pole of the rotor is located.
- WaitTime_fine is the time that ExcMag_fine is applied to find the exact angle of the rotor north pole.
- Before fine tuning these values:
  - Kspd must be determined
  - Set WaitTime_course to 0.8 seconds
  - Set WaitTime_fine to 0.8 seconds
- These large times will assure that the north pole is found while adjusting Kspd.

Kspd

Kspd - Is a gain that determines the speed at which the IPD module can lock onto an inductance peak of the motor. This inductance peak has been chosen by the course adjustment to be the north pole (d-axis) of the rotor.

- Set the Kspd value to a small start value
  - gMotorVars.Kspd = 6
- Set the variable “gThrottle” to a small value
  - Anaheim – gThrottle = 0.02
- If the motor is oscillating back and forth, that means Kspd is too low.
- Increase Kspd by increments of 5 until the motor starts smoothly.
- Cycle between gThrottle = 0.0 and gThrottle = 0.02 to make sure the motor starts smoothly.
- If the motor does not start smoothly, increase Kspd again.
- Kspd does have an uppder limit and when this limit is reached, the rotor will oscillate violently.
- Now that Kspd is adjusted, the Wait Times can be tuned

Wait Times (waitTime_coarse_sec, waitTime_fine_sec)

- WaitTime_course is adjusted first:
    - Reduce WaitTime_course from 0.8 seconds and start the IPD control.
    - As long as the motor continues to start in the proper direction, continue to decrease WaitTime_course until the rotor starts in the incorrect direction.
    - Now the lower limit of WaitTime_course is known.
    - Set WaitTime_course to a value that is larger than the lower limit.
- WaitTime_fine will determine the best starting torque per amp of the control.
    - Reduce WaitTime_fine under load until the motor does not start anymore.
        - The load that is used is the amount of torque produced by the maximum attainable Iq value of the IPD algorithm. Usually this value is less than the maximum current of the motor.
    - The lower limit of WaitTime_fine is now determined.
    - Set WaitTime_fine to a value that is greater than the upper limit.

Auto-Frequency Select (AFSEL)

AFSEL parameters are shown in Figure 96. When under HFI control, the maximum speed that can be attained is less than possible due to the voltage signal that is injected on top of the motor power signal and the voltage drop across the stator inductance. If HFI control is used above its maximum speed capability, the motor will go out of control. Usually the motor will all of a sudden change directions. When selecting freqHigh, it is important to find the maximum frequency that the motor can run at and set freqHigh at a safety margin below that frequency. A good ball park starting point for freqLow is to set it at half of freqHigh. FreqLow is determined by the minimum speed that the FAST estimator can control the motor under load. Always keep room between freqHigh and freqLow. HFI does not have as quick of a response time as FAST. The rate at which the Iq reference increases must be limited or the HFI will go out of control. IqSlopeLfEst is set to reduce the rate at which the Iq current reference changes. As Iq rises, the iron that causes the change in reluctance saturates. As a result, the motor loses its saliency. Some motor designs lose saliency under lower Iq references than other motors. IqMaxLfEst is the parameter that limits the maximum Iq reference when under HFI control. A list of the parameters for AFSEL is shown below:

AFSEL parameters that need to be tuned

- IqMaxLfEst – Maximum Iq current when under HFI control
- IqSlopeLfEst – Slope for Iq current when under HFI control
- IqMaxHfEst – Maximum Iq current when under FAST control
- IqSlopeHfEst – Slope for Iq current when under FAST control
- freqHigh – Speed at which control changes from HFI to FAST
- freqLow – Speed at which control changes from FAST to HFI

```
//! \brief     Sets the parameters
//! \param[in] handle                 The angle and frequency selector (AFSEL) handle
//! \param[in] IqMaxLfEst             The maximum Iq reference during low speed estimation
//! \param[in] IqMaxHfEst             The maximum Iq reference during high speed estimation
//! \param[in] IqSlopeLfEst           The Iq reference trajectory during low speed estimation
//! \param[in] IqSlopeHfEst           The Iq reference trajectory during high speed estimation
//! \param[in] freqLow_pu             The low speed transition frequency
//! \param[in] freqHigh_pu            The high speed transition frequency
//! \param[in] hfiHandle              The handle to the ipd_hfi object
//! \param[in] estHandle              The handle to the estimator object
extern void AFSEL_setParams(AFSEL_Handle handle,
                            const _iq IqMaxLfEst,
                            const _iq IqMaxHfEst,
                            const _iq IqSlopeLfEst,
                            const _iq IqSlopeHfEst,
                            const _iq freqLow_pu,
                            const _iq freqHigh_pu,
                            IPD_HFI_Handle hfiHandle,
                            EST_Handle estHandle);
```

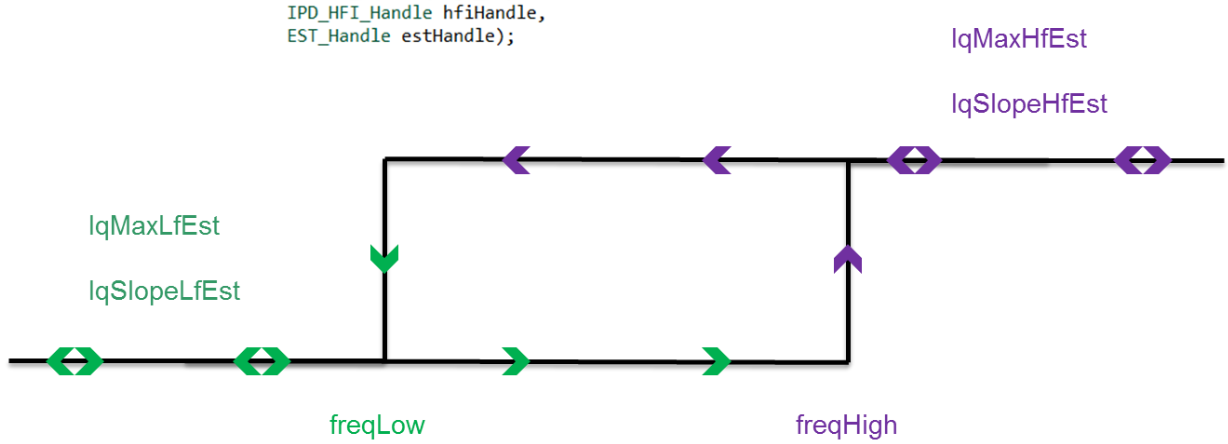


Figure 96: AFSEL parameters and a graphical explanation.

Maximum Iq (IqMaxLfEst and IqMaxHfEst)

The AFSEL parameters are adjusted after the IPD_HFI algorithm is tuned. Due to saturation in the salient motor, a maximum Iq value for the HFI algorithm is set while the HFI is in control. Once FAST is in control, the maximum Iq value can be increased to the rated value of the motor.

- IqMaxLfEst is adjusted after IPD_HFI is tuned. The maximum Iq for the low frequency estimator (HFI) is increased until the motor is out of control.
- IqMaxHfEst is usually set to the maximum current of the motor.

Iq Slope (IqSlopeLfEst and IqSlopeHfEst)

- IqSlopeLfEst is adjusted during acceleration when the low frequency estimator is active. If the maximum Iq reference is commanded and the motor becomes unstable, the slope value must be increased until the motor startup and acceleration is stable.
- IqSlopeHfEst is set to a much faster value than IqSlopeLfEst.

Conclusion

The FAST algorithm has a low speed limitation due to the bemf disappearing at low speeds. To allow for the full speed range of control a high frequency is injected into the motor and controlled with the HFI algorithm. The IPD algorithm uses the interaction of the rotor's magnet and the BH hysteresis curve of the stator's iron to initialize the HFI algorithm to the d-axis immediately after the power is turned on to the motor. The HFI algorithm has a speed limit that is well below the speed capability of the motor. The auto-frequency select (AFSEL) algorithm automatically selects whether the HFI or FAST estimators control the FOC system.