InstaSPIN-FOC™ and InstaSPIN-MOTION™

User's Guide

Literature Number: SPRUHJ1H
January 2013–Revised June 2019

Contents

Preface **20**

1 Introduction **23**

1.1 An Overview of InstaSPIN-FOC and FAST 24

1.1.1 FAST Estimator Features 25

1.1.2 InstaSPIN-FOC Solution Features 25

1.1.3 InstaSPIN-FOC Block Diagrams 26

1.1.4 Comparing FAST Estimator to Typical Solutions 28

1.1.5 FAST Provides Sensorless FOC Performance 29

1.2 An Overview of InstaSPIN-MOTION and SpinTAC 31

1.2.1 InstaSPIN-MOTION Key Capabilities and Benefits 33

1.2.2 InstaSPIN-MOTION Block Diagrams 37

1.2.3 Application Examples 45

2 Quick Start Kits - TI Provided Software and Hardware **50**

2.1 Evaluating InstaSPIN-FOC and InstaSPIN-MOTION 53

3 InstaSPIN and MotorWare **58**

3.1 MotorWare Directory Structure 60

3.1.1 MotorWare – drivers 61

3.1.2 MotorWare – ide 62

3.1.3 MotorWare – modules 63

3.1.4 MotorWare – solutions 63

3.2 MotorWare Object-Oriented Design 65

3.2.1 Objects 66

3.2.2 Methods 66

3.3 InstaSPIN-FOC API 68

3.3.1 Controller API Functions – ctrl.c, ctrl.h, CTRL_obj.h 70

3.3.2 Estimator API Functions – FAST Library – est.h, est_states.h 99

3.3.3 Hardware Abstraction Layer (HAL) API Functions – hal.c, hal.h, hal_obj.h 136

3.3.4 User Settings – user.c, user.h, userParams.h 152

3.3.5 Miscellaneous Functions 161

3.4 InstaSPIN-MOTION and the SpinTAC API 161

3.4.1 Header Files, Public Library, and ROM Library 165

3.4.2 Version Information 166

3.4.3 SpinTAC Structure Names 167

3.4.4 SpinTAC Variables 168

3.5 SpinTAC API 169

3.5.1 SpinTAC Velocity Control 169

3.5.2 SpinTAC Velocity Move 173

3.5.3 SpinTAC Velocity Plan 175

3.5.4 SpinTAC Velocity Identify 179

3.5.5 SpinTAC Position Convert 182

3.5.6 SpinTAC Position Control 184

3.5.7 SpinTAC Position Move 187

3.5.8 SpinTAC Position Plan 190

3.5.9 SpinTAC Functions 193

Copyright © 2013–2019, Texas Instruments Incorporated

4 User Parameters (user.h) 204
4.1 Currents and Voltages 205
4.1.1 USER_IQ_FULL_SCALE_FREQ_Hz 205
4.1.2 USER_IQ_FULL_SCALE_VOLTAGE_V 205
4.1.3 USER_ADC_FULL_SCALE_VOLTAGE_V 205
4.1.4 USER_VOLTAGE_SF 205
4.1.5 USER_IQ_FULL_SCALE_CURRENT_A 205
4.1.6 USER_ADC_FULL_SCALE_CURRENT_A 206
4.1.7 USER_CURRENT_SF 206
4.1.8 USER_NUM_CURRENT_SENSORS 206
4.1.9 USER_NUM_VOLTAGE_SENSORS 206
4.1.10 I_A_offset , I_B_offset , I_C_offset 206
4.1.11 V_A_offset , V_B_offset , V_C_offset 206
4.2 Clocks and Timers 207
4.2.1 USER_SYSTEM_FREQ_MHz 207
4.2.2 USER_PWM_FREQ_kHz 207
4.2.3 USER_MAX_VS_MAG_PU 207
4.2.4 USER_PWM_PERIOD_usec 207
4.2.5 USER_ISR_FREQ_Hz 207
4.2.6 USER_ISR_PERIOD_usec 208
4.3 Decimation 208
4.3.1 USER_NUM_PWM_TICKS_PER_ISR_TICK 208
4.3.2 USER_NUM_ISR_TICKS_PER_CTRL_TICK 208
4.3.3 USER_NUM_CTRL_TICKS_PER_CURRENT_TICK 208
4.3.4 USER_NUM_CTRL_TICKS_PER_EST_TICK 208
4.3.5 USER_NUM_CTRL_TICKS_PER_SPEED_TICK 208
4.3.6 USER_NUM_CTRL_TICKS_PER_TRAJ_TICK 208
4.3.7 USER_CTRL_FREQ_Hz 209
4.3.8 USER_EST_FREQ_Hz 209
4.3.9 USER_TRAJ_FREQ_Hz 209
4.3.10 USER_CTRL_PERIOD_usec 209
4.3.11 USER_CTRL_PERIOD_sec 209
4.4 Limits 209
4.4.1 USER_MAX_NEGATIVE_ID_REF_CURRENT_A 209
4.4.2 USER_ZEROSPEEDLIMIT 209
4.4.3 USER_FORCE_ANGLE_FREQ_Hz 210
4.4.4 USER_MAX_CURRENT_SLOPE_POWERWARP 210
4.4.5 USER_MAX_ACCEL_Hzps 210
4.4.6 USER_MAX_ACCEL_EST_Hzps 210
4.4.7 USER_MAX_CURRENT_SLOPE 210
4.4.8 USER_IDRATED_FRACTION_FOR_RATED_FLUX 210
4.4.9 USER_IDRATED_FRACTION_FOR_L_IDENT 210
4.4.10 USER_IDRATED_DELTA 210
4.4.11 USER_SPEEDMAX_FRACTION_FOR_L_IDENT 211
4.4.12 USER_FLUX_FRACTION 211
4.4.13 USER_POWERWARP_GAIN 211
4.4.14 USER_R_OVER_L_EST_FREQ_Hz 211
4.5 Poles 211
4.5.1 USER_VOLTAGE_FILTER_POLE_Hz 211
4.5.2 USER_VOLTAGE_FILTER_POLE_rps 211
4.5.3 USER_OFFSET_POLE_rps 211
4.5.4 USER_FLUX_POLE_rps 211
4.5.5 USER_DIRECTION_POLE_rps 211

Copyright © 2013–2019, Texas Instruments Incorporated

4.5.6 USER_SPEED_POLE_rps 212
4.5.7 USER_DCBUS_POLE_rps 212
4.5.8 USER_EST_KAPPAQ 212
4.6 User Motor and ID Settings 212
4.6.1 USER_MOTOR_TYPE 212
4.6.2 USER_MOTOR_NUM_POLE_PAIRS 212
4.6.3 USER_MOTOR_Rr 212
4.6.4 USER_MOTOR_Rs 212
4.6.5 USER_MOTOR_Ls_d 212
4.6.6 USER_MOTOR_Ls_q 212
4.6.7 USER_MOTOR_RATED_FLUX 212
4.6.8 USER_MOTOR_MAGNETIZING_CURRENT 213
4.6.9 USER_MOTOR_RES_EST_CURRENT 213
4.6.10 USER_MOTOR_IND_EST_CURRENT 213
4.6.11 USER_MOTOR_MAX_CURRENT 213
4.6.12 USER_MOTOR_FLUX_EST_FREQ_Hz 213
4.6.13 USER_MOTOR_ENCODER_LINES (InstaSPIN-MOTION Only) 213
4.6.14 USER_MOTOR_MAX_SPEED_KRPM (InstaSPIN-MOTION Only) 213
4.6.15 USER_SYSTEM_INERTIA (InstaSPIN-MOTION Only) 213
4.6.16 USER_SYSTEM_FRICTION (InstaSPIN-MOTION Only) 213
4.6.17 USER_SYSTEM_BANDWIDTH_SCALE (InstaSPIN-MOTION Only) 213
4.7 SpinTAC Parameters (spintac_velocity.h and spintac_position.h) 214
4.7.1 Macro Definitions 214
4.7.2 Type Definitions 215
4.7.3 Functions 216
4.8 Setting ACIM Motor Parameters in user.h 218
4.8.1 Getting Parameters From an ACIM Datasheet 219

5 Managing Motor Signals 222

5.1 Software Prerequisites 223
5.1.1 IQ Full-Scale Frequency 223
5.1.2 IQ Full-Scale Voltage 223
5.1.3 IQ Full-Scale Current 225
5.1.4 Max Current 225
5.1.5 Decimation Rates 226
5.1.6 System Frequency 227
5.1.7 PWM Frequency 227
5.1.8 Max Voltage Vector 227
5.2 Hardware Prerequisites 228
5.2.1 Current Feedback Gain 229
5.2.2 Current Feedback Polarity 230
5.2.3 Voltage Feedback 232
5.2.4 Voltage Filter Pole 234
5.2.5 Number of Shunt Resistors 235
5.2.6 Dead-Time Configuration 236
5.2.7 Analog Inputs Configuration 237
5.2.8 PWM Outputs Configuration 238

6 Motor Identification and State Diagrams 239

6.1 InstaSPIN Motor Identification 240
6.2 Motor Identification Process Overview 242
6.2.1 Controller (CTRL) State Machine 242
6.2.2 Estimator (EST) State Machine 245
6.2.3 Controller (CTRL) and Estimator (EST) State Machine Dependencies 247
6.3 Differences between PMSM and ACIM Identification Process 248

Copyright © 2013–2019, Texas Instruments Incorporated

6.4 Prerequisites ... 249
6.4.1 Mechanical Prerequisites ... 249
6.4.2 Hardware Prerequisites ... 249
6.4.3 Software Prerequisites ... 249
6.4.4 Software Configuration for PMSM Motor Identification ... 249
6.4.5 Software Configuration for ACIM Motor Identification ... 251
6.5 Full Identification of PMSM Motors ... 252
6.5.1 CTRL_State_Idle and EST_State_Idle ... 254
6.5.2 CTRL_State_OffLine and EST_State_Idle (Hardware Offsets Calibrated) ... 254
6.5.3 CTRL_State_OnLine and EST_State_RoverL ... 255
6.5.4 CTRL_State_OnLine and EST_State_Rs ... 259
6.5.5 CTRL_State_OnLine and EST_State_RampUp ... 262
6.5.6 CTRL_State_OnLine and EST_State_RatedFlux ... 264
6.5.7 CTRL_State_OnLine and EST_State_Ls ... 266
6.5.8 CTRL_State_OnLine and EST_State_RampDown ... 268
6.5.9 CTRL_State_OnLine and EST_State_MotorIdentified ... 269
6.5.10 CTRL_State_Idle and EST_State_Idle ... 270
6.6 Full Identification of ACIM Motors ... 270
6.6.1 CTRL_State_Idle and EST_State_Idle ... 272
6.6.2 CTRL_State_OffLine and EST_State_Idle ... 272
6.6.3 CTRL_State_OnLine and EST_State_RoverL ... 272
6.6.4 CTRL_State_OnLine and EST_State_Rs ... 272
6.6.5 CTRL_State_OnLine and EST_State_RampUp ... 272
6.6.6 CTRL_State_OnLine and EST_State_IdRated ... 273
6.6.7 CTRL_State_OnLine and EST_State_RatedFlux ... 276
6.6.8 CTRL_State_OnLine and EST_State_RampDown ... 278
6.6.9 CTRL_State_Idle and EST_State_LockRotor ... 279
6.6.10 CTRL_State_OnLine and EST_State_Ls ... 280
6.6.11 CTRL_State_OnLine and EST_State_Rr ... 281
6.6.12 CTRL_State_OnLine and EST_State_RampDown ... 282
6.6.13 CTRL_State_OnLine and EST_State_MotorIdentified ... 284
6.6.14 CTRL_State_Idle and EST_State_Idle ... 285
6.7 Recalibration of PMSM and ACIM Motor Identification ... 285
6.7.1 Recalibration of PMSM and ACIM Motors After Full Identification ... 285
6.7.2 Recalibration of PMSM and ACIM Motors after Using Parameters from user.h ... 293
6.8 Setting PMSM Motor Parameters in user.h ... 293
6.8.1 Getting Parameters from a PMSM Datasheet ... 294
6.9 Troubleshooting Motor Identification ... 297
6.9.1 General Checklist ... 297
6.9.2 Troubleshooting PMSM Motor Identification ... 298
6.9.3 Troubleshooting ACIM Motor Identification ... 303
7 Inertia Identification ... 304
7.1 InstaSPIN-MOTION Inertia Identification ... 306
7.2 Inertia Identification Process Overview ... 307
7.3 Software Configuration for SpinTAC Velocity Identify ... 310
7.3.1 Include the Header File ... 310
7.3.2 Declare the Global Variables ... 310
7.3.3 Initialize the Configuration Variables ... 310
7.3.4 Call SpinTAC Velocity Identify ... 311
7.4 Troubleshooting Inertia Identification ... 312
7.4.1 ERR_ID ... 312
7.4.2 2003 Error ... 312
7.4.3 2004 Error ... 313

Copyright © 2013–2019, Texas Instruments Incorporated

7.4.4 2006 Error 313
7.5 Difficult Applications for Inertia Identification 313
7.5.1 Automotive Pumps (Large-Cogging Force / Large Friction) 313
7.5.2 Direct Drive Washing Machines (Low-Rated Speed and Large Back EMF) 314
7.5.3 Compressors (Large Start-Up Current) 316
8 MCU Considerations 318
8.1 InstaSPIN-Enabled Devices 319
8.1.1 softwareUpdate1p6() - Function is Required in User Code 319
8.2 ROM and User Memory Overview 320
8.2.1 InstaSPIN-FOC Full Implementation in ROM 320
8.2.2 InstaSPIN-FOC Minimum Implementation in ROM 322
8.2.3 InstaSPIN-MOTION in ROM 323
8.3 Details on CPU Load and Memory Footprint Measurements 324
8.3.1 CPU Utilization Measurement Details 324
8.3.2 Memory Allocation Measurement Details 325
8.3.3 IQ Math Built in ROM 326
8.3.4 Stack Utilization Measurement Details 326
8.3.5 InstaSPIN Main Interrupt 327
8.3.6 Clock Rate 327
8.4 Memory Footprint 328
8.4.1 Device Memory Map 328
8.4.2 InstaSPIN Memory Footprint 330
8.4.3 Memory Wait-States 331
8.4.4 Flash Configuration Required Even for RAM-Only Execution 332
8.4.5 Debug (IDE) of EXE-Only Memory 332
8.5 CPU Load 333
8.5.1 F2806xF Devices 333
8.5.2 F2806xM Devices 341
8.5.3 F805xF Devices 348
8.5.4 F2805xM Devices 349
8.5.5 F2802xF Devices 351
8.6 Digital and Analog Pins 353
8.6.1 Pin Utilization 353
8.6.2 F2805x Analog Front-End (AFE) 354
9 Real-Time Structure 358
9.1 InstaSPIN Software Execution Clock Tree 359
9.2 Decimating in Software for Real-Time Scheduling 362
9.2.1 USER_NUM_ISR_TICKS_PER_CTRL_TICK 362
9.2.2 USER_NUM_CTRL_TICKS_PER_CURRENT_TICK 368
9.2.3 USER_NUM_CTRL_TICKS_PER_EST_TICK 369
9.2.4 Practical Example 370
9.2.5 USER_NUM_CTRL_TICKS_PER_SPEED_TICK 374
9.2.6 USER_NUM_CTRL_TICKS_PER_TRAJ_TICK 375
9.3 Decimating in Hardware 377
10 Managing Startup Time 382
10.1 Startup with Offsets and Rs Recalibration 383
10.2 Startup with Only Offsets Recalibration 384
10.3 Startup with Rs Recalibration 385
10.4 Startup with No Recalibration 387
10.5 Bypassing Inertia Estimation 388
11 Tuning Regulators 390
11.1 PI Controllers Introduction 391

Copyright © 2013–2019, Texas Instruments Incorporated

11.2 PI Design for Current Controllers 393
11.3 PI Design for Speed Controllers 397
11.4 Calculating PI Gains Based On Stability and Bandwidth 399
11.5 Calculating Speed and Current PI Gains Based on Damping Factor 401
11.6 Considerations When Adding Poles to the Speed Loop 406
11.7 Speed PI Controller Considerations: Current Limits, Clamping and Inertia 408
11.8 Considerations When Designing PI Controllers for FOC Systems 411
11.8.1 FOC Differences Between Motor Types 412
11.8.2 Coupling Between Q-Axis and D-Axis 412
11.9 Sampling and Digital Systems Considerations 415
11.9.1 Sampling Period Considerations in the Integral Gain 419
11.9.2 Number Format Considerations 419
11.9.3 PI Coefficients Scaling Considerations 420
12 InstaSPIN-MOTION Controllers 421
12.1 Stability 422
12.1.1 Quantifying Stability 422
12.1.2 Performance 424
12.1.3 Trade-Off Between Stability and Performance 426
12.1.4 Tuning the SpinTAC Controller 426
12.2 Software Configuration for the SpinTAC Velocity Control 428
12.2.1 Include the Header File 428
12.2.2 Declare the Global Structure 429
12.2.3 Initialize the Configuration Variables 429
12.2.4 Call SpinTAC Velocity Control 430
12.2.5 Troubleshooting SpinTAC Velocity Control 430
12.3 Optimal Performance in Speed Control 431
12.3.1 Introduction 431
12.3.2 Comparing Speed Controllers 431
12.3.3 Disturbance Rejection 431
12.3.4 Profile Tracking 433
12.3.5 InstaSPIN-MOTION Velocity Control Advantage 435
12.4 Software Configuration for SpinTAC Position Control 440
12.4.1 Include the Header File 440
12.4.2 Declare the Global Structure 441
12.4.3 Initialize the Configuration Variables 441
12.4.4 Call SpinTAC Position Control 442
12.4.5 Troubleshooting SpinTAC Position Control 442
12.5 Optimal Performance in Position Control 443
12.5.1 Introduction 443
12.5.2 Comparing Position Controllers 443
12.5.3 Disturbance Rejection 443
12.5.4 Profile Tracking 445
12.5.5 InstaSPIN-MOTION Position Control Advantage 447
13 Trajectory Planning 453
13.1 InstaSPIN-MOTION Profile Generation 454
13.1.1 Jerk Impact on System Performance 454
13.2 Software Configuration for SpinTAC Velocity Move 455
13.2.1 Include the Header File 455
13.2.2 Declare the Global Structure 456
13.2.3 Initialize the Configuration Variables 456
13.2.4 Call SpinTAC Velocity Move 456
13.2.5 Troubleshooting SpinTAC Velocity Move 457
13.3 Software Configuration for SpinTAC Position Move 458

Copyright © 2013–2019, Texas Instruments Incorporated

13.3.1 Include the Header File 458
13.3.2 Declare the Global Structure 458
13.3.3 Initialize the Configuration Variables 458
13.3.4 Call SpinTAC Position Move 459
13.3.5 Troubleshooting SpinTAC Position Move 460
13.4 InstaSPIN-MOTION Sequence Planning 461
13.4.1 SpinTAC Velocity Plan Elements 461
13.4.2 SpinTAC Velocity Plan Element Limits 462
13.4.3 SpinTAC Velocity Plan Example: Washing Machine Agitation 463
13.4.4 SpinTAC Velocity Plan Example: Garage Door 463
13.4.5 SpinTAC Velocity Plan Example: Washing Machine 464
13.4.6 SpinTAC Position Plan Example: Vending Machine 466
13.5 Software Configuration for SpinTAC Velocity Plan 467
13.5.1 Include the Header File 468
13.5.2 Define the Size of the Configuration Array 468
13.5.3 Declare the Global Structure 468
13.5.4 Initialize the Configuration Variables 468
13.5.5 Call SpinTAC Velocity Plan 470
13.5.6 Call SpinTAC Velocity Plan Tick 471
13.5.7 Update SpinTAC Velocity Plan with SpinTAC Velocity Move Status 471
13.6 Troubleshooting SpinTAC Velocity Plan 471
13.6.1 ERR_ID 471
13.6.2 Configuration Errors 472
13.7 Software Configuration for SpinTAC Position Plan 474
13.7.1 Include the Header File 474
13.7.2 Define the Size of the Configuration Array 474
13.7.3 Declare the Global Structure 474
13.7.4 Initialize the Configuration Variables 475
13.7.5 Call SpinTAC Position Plan 476
13.7.6 Call SpinTAC Position Plan Tick 478
13.7.7 Update SpinTAC Position Plan with SpinTAC Position Move Status 478
13.8 Troubleshooting SpinTAC Position Plan 478
13.8.1 ERR_ID 478
13.8.2 Configuration Errors 479
13.9 Conclusion 481
14 Managing Full Load at Startup, Low-Speed and Speed Reversal 482
14.1 Low-Speed Operation with Full Load 484
14.1.1 Low Speed with Full Load Considerations 484
14.1.2 Low Speed With Full Load Transient Examples 486
14.2 Speed Reversal with Full Load 495
14.2.1 Low Speed with Full Load Speed Reversal Considerations 495
14.2.2 Low Speed with Full Load Speed Reversal Examples 495
14.3 Motor Startup with Full Load 502
14.3.1 Motor Startup with Full Load Considerations 502
14.3.2 Motor Startup with Full Load Examples 503
14.4 Rapid Acceleration from Standstill With Full Load 511
14.4.1 Fastest Motor Startup with Full Load without Motor Alignment Considerations 511
14.4.2 Fastest Motor Startup with Full Load with Motor Alignment Considerations 516
14.5 Overloading and Motor Overheating 521
14.5.1 Overloading and Motor Overheating Considerations 521
14.5.2 Overloading and Motor Overheating Example 522
14.6 InstaSPIN-MOTION and Low-Speed Considerations 526
15 Rs Online Recalibration 527

Copyright © 2013–2019, Texas Instruments Incorporated

15.1 Overview 529
15.2 Resistance vs. Temperature 529
15.3 Accurate Rs Knowledge Needed at Low Speeds Including Startup 529
15.4 Introduction to Rs Online Recalibration 530
15.5 Rs Online vs. Rs Offline 532
15.6 Enabling Rs Online Recalibration 534
15.7 Disabling Rs Online Recalibration 536
15.8 Modifying Rs Online Parameters 536
15.8.1 Adjusting Injected Current Magnitude 537
15.8.2 Adjusting Slow Rotating Angle 540
15.8.3 Adjusting Delta Increments and Decrements of the Rs Online Value 542
15.8.4 Adjusting Filter Parameters 543
15.9 Monitoring Rs Online Resistance Value 545
15.9.1 Rs Online Floating Point Value 545
15.9.2 Rs Online Fixed Point Value 545
15.10 Using the Rs Online Feature as a Temperature Sensor 546
15.11 Rs Online Related State Diagrams (CTRL and EST) 546
16 PowerWarp™ 547
16.1 Overview 548
16.2 Enabling PowerWarp 549
16.3 PowerWarp Current Slopes 550
16.4 Practical Example 551
16.5 Case Study 553
17 Shunt Current Measurements 556
17.1 Introduction 557
17.2 Signals 557
17.3 1-Shunt 557
17.4 2-Shunt 561
17.5 3-Shunt 562
17.6 Development Kits 563
17.6.1 DRV8312 564
17.6.2 DRV8301 564
17.7 Conclusion 564
18 Sensored Systems 565
18.1 Hardware Configuration for Quadrature Encoder 566
18.1.1 Pin Usage 566
18.2 Software Configuration for Quadrature Encoder 566
18.2.1 Configure Motor for EQEP Operation 566
18.2.2 Initialize EQEP Handle 566
18.2.3 Set Digital IO to connect to QEP Peripheral 567
18.2.4 Enable Clock to eQEP 567
18.2.5 Initialize ENC Module 567
18.2.6 Set Up ENC Module 567
18.2.7 Call eQEP Function 568
18.2.8 Provide eQEP Angle to FOC 568
18.3 InstaSPIN-MOTION Position Convert 568
18.3.1 Software Configuration for SpinTAC Position Convert 568
18.3.2 Troubleshooting SpinTAC Position Convert 569
A Definition of Terms and Acronyms 571
Revision History 573

Copyright © 2013–2019, Texas Instruments Incorporated

List of Figures

1-1. FAST - Estimating Flux, Angle, Speed, Torque - Automatic Motor Identification 24
1-2. i. Block Diagram of Entire InstaSPIN-FOC Package in ROM (except F2802xF devices) 26
1-3. Block Diagram of InstaSPIN-FOC in User Memory, with Exception of FAST in ROM 27
1-4. Sensored FOC System 30
1-5. InstaSPIN-MOTION = C2000 Piccolo Microcontroller + FAST Software Sensor (optional) + Auto-Tuned Inner Torque Controller + SpinTAC Motion Control Suite 32
1-6. SpinTAC Motion Control Suite Components 33
1-7. Simple Tuning Interface 35
1-8. Curves Available in SpinTAC Move 35
1-9. State Transition Map for a Washing Machine 36
1-10. State Transition Map for a Garage Door System 37
1-11. InstaSPIN-MOTION in User Memory, with Exception of FAST and SpinTAC in ROM 39
1-12. InstaSPIN-MOTION in ROM 40
1-13. InstaSPIN-MOTION Speed Control with a Mechanical Sensor 42
1-14. InstaSPIN-MOTION Position Control with Mechanical Sensor and Redundant FAST Software Sensor 44
1-15. Washing Machine Profile 46
1-16. InstaSPIN-MOTION Minimizes Error 47
1-17. First Spin Cycle - 500 rpm 48
1-18. Second Spin Cycle - 2000 rpm 49
2-1. InstaSPIN-MOTION GUI Using *Motor Identification* Tab 53
2-2. InstaSPIN-MOTION GUI Using *Speed or Torque* Tab 54
2-3. InstaSPIN-MOTION GUI Using *SpinTAC 1:Startup* Tab 55
2-4. InstaSPIN-MOTION GUI *SpinTAC 2:Tuning* Tab 56
2-5. InstaSPIN-MOTION GUI Using *SpinTAC 3:Motion* Tab 57
3-1. Block Diagram of InstaSPIN-FOC in User Memory, with Exception of FAST in ROM 69
3-2. InstaSPIN-MOTION Velocity Control 161
3-3. InstaSPIN-MOTION Position Control 163
3-4. SpinTAC Module Directory Structure 165
3-5. SpinTAC Velocity Control Interfaces 170
3-6. SpinTAC Velocity Control State Transition Map 172
3-7. SpinTAC Velocity Move Interfaces 173
3-8. SpinTAC Velocity Move State Transition Map 174
3-9. SpinTAC Velocity Plan Interfaces 176
3-10. SpinTAC Velocity Plan State Transition Map 177
3-11. SpinTAC Velocity Identify Interfaces 179
3-12. SpinTAC Velocity Identify State Transition Map 181
3-13. SpinTAC Position Convert Interfaces 182
3-14. SpinTAC Position Convert State Transition Map 183
3-15. SpinTAC Position Control Interfaces 184
3-16. SpinTAC Position Control State Transition Map 186
3-17. SpinTAC Position Move Interfaces 187
3-18. SpinTAC Position Move State Transition Map 189
3-19. SpinTAC Position Plan Interfaces 190
4-1. Example ACIM Motor Datasheet 219
5-1. USER_MOTOR_MAX_CURRENT in InstaSPIN 226
5-2. 1.65-V Reference from 3.3-V Input Circuit Example 229
5-3. Typical Differential Amplifier Circuit 229

Copyright © 2013–2019, Texas Instruments Incorporated

5-4. Calculated Resistance Values Circuit 230
5-5. Positive Feedback 231
5-6. Negative Feedback 232
5-7. Voltage Feedback Circuit 233
5-8. Voltage Feedback Circuit 234
5-9. BOOSTXL-DRV8305 EVM HALF-BRIDGES & BACK-EMF SENSE Circuit 234
5-10. Shunt Resistors 235
5-11. Dead-Time Configuration 236
5-12. Analog Connections 237
5-13. PWM Pin Configuration 238
6-1. InstaSPIN Motor Identification Components 240
6-2. Full Implementation of InstaSPIN-FOC (F2805xF, F2805xM, F2806xF, and F2806xM Devices Only) 241
6-3. Minimum Implementation of InstaSPIN-FOC (F2802xF, F2805xF, F2805xM, F2806xF, and F2806xM Devices) 242
6-4. Controller (CTRL) State Diagram 243
6-5. Estimator (EST) State Diagram 245
6-6. Controller and Estimator State Diagrams - Dependency Shown 247
6-7. PMSM and ACIM States in EST State Diagram 248
6-8. Full PMSM Identification - CTRL and EST Sequence of States 253
6-9. CCStudio Watch Window after Offset Calibration 254
6-10. 50% PWM Duty for Offset Calculation 255
6-11. RoverL EST State 255
6-12. Injected Current for Measuring RoverL EST State 257
6-13. Internal Code that Sets Kp and Ki Initial Gains for Current Controllers 258
6-14. Rs EST State 259
6-15. Phase Current During Rs Identification EST State 261
6-16. Ramp-Up EST State 262
6-17. Ramp-Up Timing 263
6-18. Phase Current During RampUp EST State 263
6-19. RampUp Timing with change in Acceleration and Final Speed 264
6-20. PMSM Rated Flux EST State 264
6-21. Phase Current During Rated Flux EST State 265
6-22. Stator Inductance EST State 266
6-23. Injected Current for Ls Identification 267
6-24. Current Ramp for Ls Identification 268
6-25. Complete PMSM Motor ID Process in EST State Diagram 269
6-26. Phase Current Measurement of Entire PMSM Motor ID Process 270
6-27. Full ACIM Identification - CTRL and EST Sequence of States 271
6-28. Ramp-Up EST State 272
6-29. Oscilloscope Plot of ACIM RampUp Acceleration 273
6-30. ACIM Id Rated EST State 273
6-31. Oscilloscope Plot of Phase Current During Id Rated EST State 274
6-32. Phase Current Oscillation During Id Rated Measurement 275
6-33. Reduced Phase Current Oscillation During Id Rated Measurement 276
6-34. ACIM Rated Flux EST State 277
6-35. Phase Current During Id Rated EST State 277
6-36. ACIM Ramp Down EST State 278
6-37. Ramp Down of ACIM Phase Current Prior to LockRotor State 279
6-38. ACIM Lock Rotor EST State 279

Copyright © 2013–2019, Texas Instruments Incorporated

6-39. ACIM Stator Inductance EST State 280
6-40. ACIM Current for the Stator Inductance EST State 281
6-41. Rotor Resistance EST State 281
6-42. Injected Current for Rr Identification 282
6-43. ACIM Ramp Down EST State after Completing Rr 283
6-44. ACIM Current During Rr and RampDown 283
6-45. Complete ACIM Motor ID Process in EST State Diagram 284
6-46. Phase Current of Entire ACIM Motor ID Process 285
6-47. PMSM and ACIM Recalibration - CTRL and EST Sequence of States 286
6-48. Motor Recalibration EST States 287
6-49. Phase Current during Offset Recalibration 288
6-50. Phase Current Showing Rs Recalibration Timing 290
6-51. Transitioning to Online from EST Rs 291
6-52. Phase Current Transitioning from EST Rs to Online 291
6-53. Transitioning to Online from EST Idle 292
6-54. Phase Current Transitioning from EST Idle to Online 292
6-55. Timing of Complete Recalibration 293
6-56. Example PMSM Motor Datasheet 295
6-57. Determining Motor Flux from Phase Voltage of Motor in Generator Mode 297
6-58. RampUp Timing with change in Acceleration 300
6-59. PMSM RampUp Acceleration 300
7-1. Example of Identifying Inertia in a Simple Motion System 304
7-2. Histogram of 100 Inertia Identification Trials 306
7-3. SpinTAC Speed Controller Inertia Tolerance 307
7-4. Flowchart for SpinTAC Velocity Identify Process 308
7-5. SpinTAC Velocity Identify Torque Reference 309
7-6. SpinTAC Velocity Identify Speed Feedback 309
7-7. Speed Feedback for Inertia Identification for an Automotive Pump 314
7-8. Speed Feedback for Inertia Identification for a Direct Drive Washing Machine 315
7-9. DC Bus Voltage for Inertia Identification for a Direct Drive Washing Machine 316
7-10. Speed Feedback for Inertia Identification for a Compressor 317
8-1. InstaSPIN-FOC Full Implementation in ROM 321
8-2. InstaSPIN-FOC Minimum Implementation in ROM 322
8-3. InstaSPIN-MOTION in ROM 323
8-4. InstaSPIN Software Execution Clock Tree 325
8-5. Function Calls from the Main ISR 327
8-6. F2806xF and F2806xM Allocated Memory for InstaSPIN-FOC and SpinTAC Library 328
8-7. F2805xF and F2805xM Allocated Memory for InstaSPIN-FOC and SpinTAC Library 329
8-8. F2802xF Allocated Memory for InstaSPIN-FOC Library 330
8-9. SpinTAC Velocity Plan Example 346
8-10. Current Signal Routing Directly to PGAs With Single-Ended Connections 355
8-11. Feedback of Phase Currents Using External Differential Amplifiers 356
8-12. Using the AFE's Built-In Voltage Reference For Measuring a Bipolar Signal 357
9-1. Clock Timing - CPU to ISR Generation 360
9-2. Software Execution Clock Tree 361
9-3. Real-Time Scheduling Tick Rates 362
9-4. Tick Counter Flowchart 363
9-5. InstaSPIN Timing 363
9-6. InstaSPIN Timing Software Execution Clock Tree 363

Copyright © 2013–2019, Texas Instruments Incorporated

9-7. InstaSPIN Timing Completes Execution with No ISR Overrun 364
9-8. InstaSPIN Timing Software Execution Clock Tree - No ISR Overrun 364
9-9. InstaSPIN Timing with a Higher PWM Frequency 365
9-10. Interrupt Overrun without Decimation Timing 365
9-11. Interrupt Overrun without Decimation Software Execution Clock Tree 366
9-12. Interrupt Overrun with Decimation Timing 366
9-13. Interrupt Overrun with Decimation Software Execution Clock Tree 366
9-14. ISR Frequency Waveforms 367
9-15. Software Execution Clock Tree for ISR Waveforms 368
9-16. Tick Rate Timing 368
9-17. Tick Rate Software Execution Clock Tree 369
9-18. FAST Estimator Tick Rate Timing 369
9-19. FAST Estimator Tick Rate Software Execution Clock Tree 370
9-20. Tick Rates Timing 370
9-21. Tick Rates Software Execution Clock Tree 371
9-22. CTRL vs. EST Timing - Tick Rate = 2 371
9-23. CTRL vs. EST Software Execution Clock Tree - Tick Rate = 2 372
9-24. CTRL vs. EST Timing - Tick Rate = 3 372
9-25. CTRL vs. EST Software Execution Clock Tree - Tick Rate = 3 373
9-26. ISR vs. CTRL Timing - Tick Rate = 2 373
9-27. ISR vs. CTRL Software Execution Clock Tree - Tick Rate = 2 374
9-28. Speed Controller Timing - Tick Rate = 10 374
9-29. Speed Controller Software Execution Clock Tree - Tick Rate = 10 375
9-30. CTRL vs TRAJ Tick Rate Timing 375
9-31. CTRL vs TRAJ Tick Rate Software Execution Clock Tree 376
9-32. All Tick Rates and Dependencies Timing 377
9-33. All Tick Rates and Dependencies Software Execution Clock Tree 377
9-34. Hardware Decimation Software Execution Clock Tree 378
9-35. SOC Event Timing 378
9-36. SOC Event Software Execution Clock Tree 379
9-37. PWM Converions on Every Second PWM Cycle Timing 379
9-38. PWM Converions on Every Second PWM Cycle Software Execution Clock Tree 380
9-39. PWM Converions on Every Third PWM Cycle Timing 380
9-40. PWM Converions on Every Third PWM Cycle Software Execution Clock Tree 381
10-1. Startup with Offsets and Rs Recalibration 383
10-2. Current and Output Voltage for Each State 383
10-3. Startup with Only Offsets Recalibration 384
10-4. Offset State Current and Output Voltage 385
10-5. Startup with Rs Recalibration 386
10-6. Rs Recalibration Current and Output Voltage 386
10-7. Startup with No Recalibration 387
10-8. Rs Recalibration Bypass Current and Output Voltage 388
11-1. USS New Mexico Around the Time it was Retrofitted with PID Control 391
11-2. Parallel Path Topology 392
11-3. Series Topology 393
11-4. Frequency Response 394
11-5. PI Controller in a Current Controller 394
11-6. Cascaded Speed Control Loop 397
11-7. Bode Plot 400

Copyright © 2013–2019, Texas Instruments Incorporated

11-8. Speed Controller Open Loop Magnitude and Phase Response as a Function of δ 402
11-9. Speed Controller Closed Loop Bandwidth as a Function of δ 403
11-10. Step Response of Speed Controller as a Function of δ 404
11-11. Simulated Step Response of Speed Controller Design from the Above Example 406
11-12. Speed Controller with Filtered Speed Feedback 406
11-13. Step Response of a System with Variable Damping and Pole Placement 408
11-14. PI Controller with Static Integrator Clamping 409
11-15. PI Controller with Dynamic Integrator Clamping 409
11-16. Example Comparison of Integrator Clamping Techniques 410
11-17. Average Motor Torque Readings 411
11-18. Typical FOC Speed Control of a PMSM 411
11-19. Decoupled PI Controllers for a PMSM 413
11-20. Simulated Effectiveness of Current Regulator Decoupling 414
11-21. Compensation Block Used for Axis Decoupling with ACIMs 415
11-22. Digital Field-Oriented Control System for a PMSM 416
11-23. Magnitude and Phase Plots for a Sample-and-Hold 417
11-24. Effect of Viscous Damping (kv) on the Load Bode Plot 418
11-25. Typical Implementation of a Digital Integrator 419
12-1. Typical Step Responses of Stable and Unstable Systems 422
12-2. Typical Open Loop Bode of SpinTAC Velocity Control 423
12-3. Typical Open Loop Bode of SpinTAC Position Control 424
12-4. Typical Performance Bode of SpinTAC Velocity Control 425
12-5. Typical Performance Bode of SpinTAC Position Control 425
12-6. Comparison of Bandwidths for SpinTAC Velocity Control 427
12-7. Comparison of Bandwidths for SpinTAC Position Control 428
12-8. Velocity Tuning Comparison for Applied Torque Disturbance 432
12-9. Velocity Tuning Comparison for Removed Torque Disturbance 433
12-10. Velocity Tuning Comparison for Profile Tracking 434
12-11. PI and SpinTAC Comparison for Applied Torque Disturbance Velocity Control 436
12-12. PI and SpinTAC Comparison for Removed Torque Disturbance Velocity Control 437
12-13. PI and SpinTAC Comparison for Feedforward Impact on Velocity Profile Tracking 438
12-14. PI and SpinTAC Comparison for Integrator Windup During Disturbance Rejection Velocity Control 439
12-15. PI and SpinTAC Comparison for Step Response Velocity Control 440
12-16. Position Tuning Comparison for Applied Torque Disturbance 444
12-17. Position Tuning Comparison for Removed Torque Disturbance 445
12-18. Position Tuning Comparison for Profile Tracking 446
12-19. PI and SpinTAC Position Control Comparison for Applied Torque Disturbance 448
12-20. PI and SpinTAC Position Control Comparison for Removed Torque Disturbance 449
12-21. PI and SpinTAC Comparison for Feedforward Impact on Position Profile Tracking 450
12-22. PI and SpinTAC Comparison for Low Speed Position Profile Tracking 451
12-23. PI and SpinTAC Position Control Comparison for Step Response 452
13-1. Comparison of Curves Provided by SpinTAC Position Move 454
13-2. Impact of Jerk on Iq Reference 455
13-3. State Transition Map of Example Washing Machine Agitation 463
13-4. State Transition Map of Example Garage Door 464
13-5. State Transition Map of Example Washing Machine 465
13-6. Velocity Profile During Example Washing Machine 466
13-7. State Transition Map of Example Vending Machine 467
14-1. Photograph of Test Fixture 483

Copyright © 2013–2019, Texas Instruments Incorporated

14-2. Voltage Feedback Circuit of High-Voltage Kit (TMDSHVMTRPFCKIT) 485
14-3. 4-Hz, No-Load to Full-Load Transient Plot 487
14-4. Flux Plot 488
14-5. Angle Plot 488
14-6. Zoom-in on Angle Plot - Motor Loaded 489
14-7. Zoom-in on Angle Plot - Load Removed 489
14-8. Speed Plot 490
14-9. Torque Plot 490
14-10. Iq Current Plot 491
14-11. 2-Hz, No-Load to Full-Load Transient Plot 491
14-12. Flux Plot 492
14-13. Angle Plot 492
14-14. Zoom-in on Angle Plot - Increased Motor Load 493
14-15. Zoom-in on Angle Plot - Decreased Motor Load 493
14-16. Speed Plot 494
14-17. Torque Plot 494
14-18. Iq Current Plot 495
14-19. -4 to +4 Hz with Full Load Plot 496
14-20. Flux Plot 496
14-21. Angle Plot 497
14-22. Zoom-in on Angle Plot 497
14-23. Speed Plot 498
14-24. Torque Plot 498
14-25. Iq Current Plot 499
14-26. -2 to +2 Hz with Full Load Plot 499
14-27. Flux Plot 500
14-28. Angle Plot 500
14-29. Zoom-in on Angle Plot 501
14-30. Speed Plot 501
14-31. Torque Plot 502
14-32. Iq Current Plot 502
14-33. Enable Forced Angle 503
14-34. Standstill to 4 Hz with Full Load Plot 504
14-35. Speed Controller Cycle 504
14-36. Flux Plot 505
14-37. Angle Plot 505
14-38. Zoom-in on Angle Plot 506
14-39. Speed Plot 506
14-40. Torque Plot 507
14-41. Iq Current Plot 507
14-42. Standstill to 2 Hz with Full Load Plot 508
14-43. Flux Plot 508
14-44. Angle Plot 509
14-45. Zoom-in on Angle Plot 509
14-46. Speed Plot 510
14-47. Torque Plot 510
14-48. Iq Current Plot 511
14-49. Fast Acceleration Without Alignment Plot 512
14-50. Flux Plot 513

Copyright © 2013–2019, Texas Instruments Incorporated

14-51. Angle Plot 513
14-52. Zoom-in on Angle Plot 514
14-53. Speed Plot 514
14-54. Torque Plot 515
14-55. Iq Current Plot 515
14-56. Fastest Motor Startup with Full Load with Motor Alignment Plot 517
14-57. Zoom-in on the Current Plot 518
14-58. Flux Plot 519
14-59. Angle Plot 519
14-60. Zoom-in on Angle Plot 520
14-61. Speed Plot 520
14-62. Torque Plot 521
14-63. Iq Current Plot 521
14-64. Overloading and Motor Overheating Plot 522
14-65. Zoom-in on Overloading and Motor Overheating Plot 522
14-66. Stator Resistance Plot 523
14-67. Flux Plot 523
14-68. Angle Plot 524
14-69. Zoom-in on Angle Plot 524
14-70. Speed Plot 525
14-71. Torque Plot 525
14-72. Iq Current Plot 526
15-1. FAST Estimator - Rs Online Highlighted 527
15-2. Rs Online Recalibration 530
15-3. Phase Currents at Light Loads - Rs Online Disabled 531
15-4. Phase Currents at Light Loads - Rs Online Enabled 531
15-5. Phase Currents with Mechanical Load - Rs Online Disabled 532
15-6. Phase Currents with Mechanical Load - Rs Online Enabled 532
15-7. Rs Online and Rs Offline Flowchart 533
15-8. Result of Adding 0.25 A for Rs Online 537
15-9. Result of Increasing Load for Rs Online 538
15-10. Maximum Current With Rs Online Enabled 539
15-11. 2.2-A Motor With an Rs Online Current of 5% 539
15-12. Current Shape Changes When Frequency Equals Slow Rotating Angle Frequency 541
15-13. Result of RsOnLine_Angle_Delta_pu 542
15-14. Resistance Response to Initial Value Difference 543
15-15. Delta Values Changed to Double Default Value 543
15-16. Rs Online Varies Depending on Cut-Off Frequency 544
16-1. FAST Estimator with PowerWarp 548
16-2. InstaSPIN Controller Flowchart - PowerWarp Executed in Closed Loop 549
16-3. FAST Estimator State Machine Flowchart - PowerWarp Executed in Closed Loop 550
16-4. PowerWarp Improves Motor Efficiency 552
16-5. Current Reduced When PowerWarp Enabled 552
16-6. Current Slopes When PowerWarp Disabled 553
16-7. PowerWarp Algorithm Enabled vs. TRIAC Control of Induction Motor 554
16-8. InstaSPIN-FOC with PowerWarp Enabled vs. InstaSPIN-FOC with PowerWarp Disabled 555
17-1. Typical SVM Waveform Sampled by Counter 557
17-2. Single-Shunt Current Measurement Circuit With Inverter 558
17-3. Single-Shunt Current Measurement When Sampling Times are Long Enough 559

Copyright © 2013–2019, Texas Instruments Incorporated

17-4. SVM and Regions Where Current Measurement is Not Allowed 559
17-5. Example of When Current Sampling Window Disappears 560
17-6. Phase Shifting the PWMs to Allow for a Large Enough Current Measurement Window 560
17-7. Two-Shunt Measurement Circuit With Inverter 561
17-8. Sampling Current When Using Two-Shunt Measurement Technique 562
17-9. Three-Shunt Measurement Circuit With Inverter 563
17-10. Using Three-Shunt Current Sampling Technique 563

Copyright © 2013–2019, Texas Instruments Incorporated

List of Tables

0-1. C2000 InstaSPIN Device Guide 20
1-1. FAST Estimator vs. Typical Solutions 28
1-2. InstaSPIN-MOTION Application Examples 32
3-1. User Code Header Files 166
3-2. SpinTAC Version Structure 166
3-3. SpinTAC Structure Names 167
3-4. SpinTAC Variables 168
3-5. SpinTAC Velocity Control Interface Parameters 170
3-6. SpinTAC Velocity Control State Transition 172
3-7. SpinTAC Velocity Move Interfaces 173
3-8. SpinTAC Velocity Move State Transition 175
3-9. SpinTAC Velocity Plan Interfaces 176
3-10. SpinTAC Velocity Plan State Transition 177
3-11. SpinTAC Velocity Plan Additional Functions 178
3-12. SpinTAC Velocity Identify Interfaces and Parameters 180
3-13. SpinTAC Velocity Identify State Transition 181
3-14. SpinTAC Position Convert Interfaces and Parameters 182
3-15. SpinTAC Position Convert State Transition 184
3-16. SpinTAC Position Control Interface Parameters 185
3-17. SpinTAC Position Control State Transition 186
3-18. SpinTAC Position Move Interfaces 187
3-19. Position Move State Transition 189
3-20. SpinTAC Position Plan Interfaces 191
3-21. SpinTAC Position Plan State Transition 192
3-22. SpinTAC Position Plan Additional Functions 192
4-1. ACIM Motor Parameters in user.h 218
5-1. hal.c Configuring the PLL 227
5-2. Maximum SVM Input Ranges 228
6-1. Controller (CTRL) States 243
6-2. State Transitions for Controller (CTRL) State Diagram 243
6-3. Estimator (EST) States 245
6-4. State Transitions for Estimator (EST) State Diagram 246
6-5. Listing of PMSM and ACIM EST States 248
6-6. PMSM Motor Parameters in user.h 294
7-1. SpinTAC Velocity Identify Error Code 312
8-1. InstaSPIN-Enabled Devices 319
8-2. Allocated Memory for InstaSPIN-FOC Library 328
8-3. ROM Table Addresses 329
8-4. Total Memory Usage of InstaSPIN-FOC for F2806xF and F2805xF Devices 330
8-5. Total Memory Usage of InstaSPIN-FOC for F2802xF Devices 330
8-6. Code Size and RAM Usage for SpinTAC Components 331
8-7. Stack Utilization of SpinTAC Components + InstaSPIN-FOC 331
8-8. CPU Execution Time Wait States (F2806xF and F2806xM Devices) 331
8-9. CPU Execution Time Wait States (F2805xF and F2805xM Devices) 331
8-10. CPU Execution Time Wait States (F2802xF Devices) 332
8-11. Full Implementation Memory Usage Executing in RAM 333
8-12. Full Implementation Executing in RAM 334

Copyright © 2013–2019, Texas Instruments Incorporated

8-13. Minimum Implementation Memory Usage Executing in RAM 334
8-14. Minimum Implementation Executing in RAM 334
8-15. Full Implementation Memory Usage Executing in FLASH 335
8-16. Full Implementation Executing in FLASH 335
8-17. Minimum Implementation Memory Usage Executing in FLASH 336
8-18. Minimum Implementation Executing in FLASH 336
8-19. Full Implementation Executing from ROM and FLASH 337
8-20. Minimum Implementation Executing from ROM and FLASH 338
8-21. Full Implementation Executing from ROM and FLASH 338
8-22. Minimum Implementation Executing from ROM and FLASH 339
8-23. CPU Cycle Utilization for SpinTAC with Library Executing in RAM on F2806xM Devices 341
8-24. CPU Cycle Utilization for SpinTAC Library Executing in Flash on F2806xM Devices 342
8-25. Full Implementation Executing from ROM and FLASH 348
8-26. Full Implementation Executing from ROM and FLASH 348
8-27. CPU Cycle Utilization for SpinTAC Library Executing in Flash on F2805xM Device 349
8-28. Minimum Implementation Memory Usage Executing in FLASH 351
8-29. Minimum Implementation Executing in FLASH 351
8-30. Minimum Implementation Executing from ROM, RAM, and FLASH 352
8-31. Minimum Implementation Executing from ROM, RAM and FLASH 353
8-32. Pin Utilization Per Motor 353
12-1. Time Domain Common Criteria 426
12-2. SpinTAC Velocity Control ERR_ID Code 430
12-3. SpinTAC Position Control ERR_ID Code 442
12-4. InstaSPIN-MOTION Position Control Advantage 447
13-1. SpinTAC Velocity Move ERR_ID Code 457
13-2. SpinTAC Position Move ERR_ID Code 460
13-3. Memory Requirements for SpinTAC Velocity Plan Elements 462
13-4. SpinTAC Velocity Plan ERR_ID 472
13-5. SpinTAC Velocity Plan ERR_code 472
13-6. SpinTAC Position Plan ERR_ID 478
13-7. SpinTAC Position Plan ERR_code 479
15-1. Temperature Sensor Implementation Values 546
17-1. The Eight SVM Switching States 558
17-2. Current and Voltage Ratings for TI Development Kits 563
17-3. Recommended Op-Amp Slew Rates for Corresponding Number of Sense Resistors 564
18-1. Pins Required to Connect Quadrature Encoder to eQEP Module 566
18-2. SpinTAC Position Convert ERR_ID Code 570

Copyright © 2013–2019, Texas Instruments Incorporated

About This Manual

Table 0-1. C2000 InstaSPIN Device Guide

Device Families	InstaSPIN Library Version	Solution Support	Software Package	User's Guide
TMS320F2806xF	V1p6	InstaSPIN-FOC	MotorWare	This Guide
TMS320F2086xM	V1p6	InstaSPIN-MOTION	MotorWare	This Guide
TMS320F2802xF	V1p7	InstaSPIN-FOC	MotorWare	This Guide
TMS320F2805xF	V1p7	InstaSPIN-FOC	MotorWare	This Guide
TMS320F2805xM	V1p7	InstaSPIN-MOTION	MotorWare	This Guide
TMS320F28004xC	V2p0	InstaSPIN-FOC	C2000Ware MotorControl SDK	Found within C2000Ware MotorControl SDK

Welcome and thank you for selecting Texas Instrument's InstaSPIN™ solutions. This document will guide you through the technical details of InstaSPIN software enabling you to integrate this solution into your application. The structure of this document can be summarized as:

- Introduction to
  - InstaSPIN-FOC™ and FAST™
  - InstaSPIN-MOTION™ and SpinTAC™
- Running a motor immediately with TI hardware and software
- Understand software details, from reviewing API function calls to state diagrams and tuning the speed and position control loops
- Understanding hardware aspects that directly impact InstaSPIN's performance.

All of the above are provided to help you develop a successful product using InstaSPIN-FOC or InstaSPIN-MOTION software. Example projects (labs) are a key part of this success and are designed to relate specifically with the topics in this document. They are intended for you to not only experiment with InstaSPIN but to also use as reference for your design. The most up-to-date InstaSPIN-FOC and InstaSPIN-MOTION solutions and design resources, along with practical videos, can be found here: http://www.ti.com/instaspin.

Definition of terms that are used throughout this document can be found in Appendix A at the end of this document. The most common terms used are the following:

- FOC:
  - Field-Oriented Control
- InstaSPIN-FOC:
  - Complete sensorless FOC solution provided by TI on-chip in ROM on select devices (FAST observer, FOC, speed and current loops), efficiently controlling your motor without the use of any mechanical rotor sensors.
- FAST
  - Unified observer structure which exploits the similarities between all motors that use magnetic flux for energy transduction, automatically identifying required motor parameters and providing motor feedback signals: Flux, flux Angle, motor shaft Speed, and Torque.
- SpinTAC Motion Control Suite:

Copyright © 2013–2019, Texas Instruments Incorporated

  - Includes an advanced speed and position controller, a motion engine, and a motion sequence planner. The SpinTAC disturbance-rejecting speed controller proactively estimates and compensates for system disturbances in real-time, improving overall product performance. The SpinTAC motion engine calculates the ideal reference signal (with feed forward) based on user-defined parameters. SpinTAC supports the standard industry curves, and LineStream's proprietary “smooth trajectory” curve. The SpinTAC motion sequence planner operates user-defined state transition maps, making it easy to design complex motion sequences.
- InstaSPIN-MOTION:
  - A comprehensive sensorless or sensored FOC solution for motor-, motion-, speed-, and position-control. This solution delivers robust system performance at the highest efficiency for motor applications that operate in various motion state transitions. InstaSPIN-MOTION includes the FAST unified software observer, combined with SpinTAC Motion Control Suite from LineStream Technologies.

Copyright © 2013–2019, Texas Instruments Incorporated

- MotorWare™ software:
  - TI supplied scalable software architecture for motor control, of which InstaSPIN-FOC is a part.

The InstaSPIN-FOC and InstaSPIN-MOTION software is available on the TMS320F2806xF, TMS320F2086xM, TMS320F2802xF, TMS320F2805xF, and TMS320F2805xM device families with plans to release on more devices in the future.

For more details, see the device-specific data sheets and the device-specific technical reference manuals (TRMs). The InstaSPIN TRMs have the latest performance data resulting from tests conducted by TI motor labs. This document, the *InstaSPIN User's Guide*, differs in that it is a functional "How-To" guide for using InstaSPIN-FOC or InstaSPIN-MOTION in your application.

Whether you are using TI supplied inverters and motors or using your own, this document helps you learn about this new and empowering solution from TI.

Trademarks

InstaSPIN, InstaSPIN-FOC, FAST, InstaSPIN-MOTION, MotorWare, PowerWarp, C2000, Piccolo, NexFET, Code Composer Studio, controlSUITE are trademarks of Texas Instruments.
SpinTAC is a trademark of LineStream Technologies.
All other trademarks are the property of their respective owners.

Copyright © 2013–2019, Texas Instruments Incorporated

Introduction

This chapter provides an overview of the InstaSPIN-FOC and FAST estimator and InstaSPIN-MOTION and SpinTAC motion control suite solutions.

Topic **Page**

1.1 An Overview of InstaSPIN-FOC and FAST 24
1.2 An Overview of InstaSPIN-MOTION and SpinTAC..... 31

Copyright © 2013–2019, Texas Instruments Incorporated

1.1 An Overview of InstaSPIN-FOC and FAST

TMS320F2806xF (69F, 68F, and 62F — 80- or 100-pin packages), TMS320F2802xF (26F and 27F — 48-pin package), and TMS320F2805xF (54F and 52F, — 80-pin packages) devices are the first from Texas Instruments that include the FAST (Figure 1-1) estimator and additional motor control functions needed for cascaded speed and torque loops for efficient three-phase field-oriented motor control (FOC).

Together — with F2806xF, F2805xF, and F2802xF peripheral drivers in user code — they enable a sensorless (also known as self-sensing) InstaSPIN-FOC solution which can identify, tune the torque controller and efficiently control your motor in minutes, without the use of any mechanical rotor sensors. This entire package is called InstaSPIN-FOC, which is made available in ROM. For the F2806xF devices, the ROM contains the FAST estimator and the FOC blocks; for the F2802xF devices, only the FAST estimator is in ROM; and for the F2805xF devices, the ROM contains the FAST estimator and the FOC blocks. In the case of the F2806xF devices, the user also has the option of executing all FOC functions in user memory (FLASH or RAM), which makes calls to the proprietary FAST estimator firmware in ROM. In the case of the F2805xF devices, the user also has the option of executing all FOC functions in user memory (FLASH or RAM), which makes calls to the proprietary FAST estimator firmware in ROM. In the case of the F2802xF devices, all the FOC blocks are loaded and executed from user's memory (FLASH or RAM) while the estimator is run from ROM.InstaSPIN-FOC was designed for flexibility, to accommodate a range of system software architectures. The range of this flexibility is shown in Figure 1-2 and Figure 1-3.

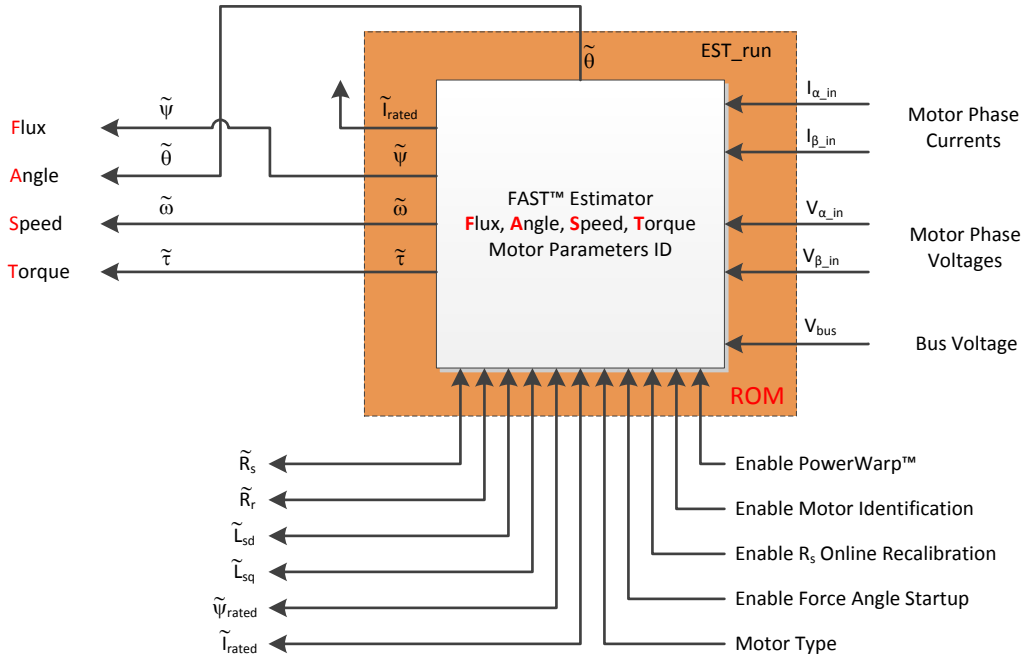


Figure 1-1. FAST - Estimating Flux, Angle, Speed, Torque - Automatic Motor Identification

Copyright © 2013–2019, Texas Instruments Incorporated

1.1.1 FAST Estimator Features

- Unified observer structure which exploits the similarities between all motors that use magnetic flux for energy transduction
  - Both synchronous (BLDC, SPM, IPM), and asynchronous (ACIM) control are possible
  - Salient compensation for Interior Permanent Magnet motors: observer tracks rotor flux and angle correctly when Ls-d and Ls-q are provided
- Unique, high quality motor feedback signals for use in control systems
  - High-quality **F**lux signal for stable flux monitoring and field weakening
  - Superior rotor flux **A**ngle estimation accuracy over wider speed range compared to traditional observer techniques (independent of all rotor parameters for ACIM)
  - Real-time low-noise motor shaft **S**peed signal
  - Accurate high bandwidth **T**orque signal for load monitoring and imbalance detection
- Angle estimator converges within first cycle of the applied waveform, regardless of speed
- Stable operation in all power quadrants, including generator quadrants
- Accurate angle estimation at steady state speeds below 1 Hz (typ.) with full torque
- Angle integrity maintained even during slow speed reversals through zero speed
- Angle integrity maintained during stall conditions, enabling smooth stall recovery
- Motor Identification process measures required electrical motor parameters of unloaded motor in under 2 minutes (typ.)
- “On-the-fly” stator resistance recalibration (online Rs) tracks stator resistance changes in real time, resulting in robust operation over temperature. This feature can also be used as a temperature sensor of the motor's windings (basepoint calibration required)
- Superior transient response of rotor flux angle tracking compared to traditional observers
- PowerWarp™ adaptively reduces current consumption to minimize the combined (rotor and stator) copper losses without compromising ACIM output power levels

1.1.2 InstaSPIN-FOC Solution Features

- Includes FAST estimator to measure rotor flux (magnitude and angle) in a sensorless FOC system
- Automatic torque (current) loop tuning, with option for user adjustments
- Automatic configuration of speed loop gains (Kp and Ki) provides stable operation for most applications, user adjustments required for optimum transient response
- Automatic or manual field weakening and field boosting
- Bus Voltage compensation
- Automatic offset calibration insures quality samples of feedback signals

Copyright © 2013–2019, Texas Instruments Incorporated

1.1.3 InstaSPIN-FOC Block Diagrams

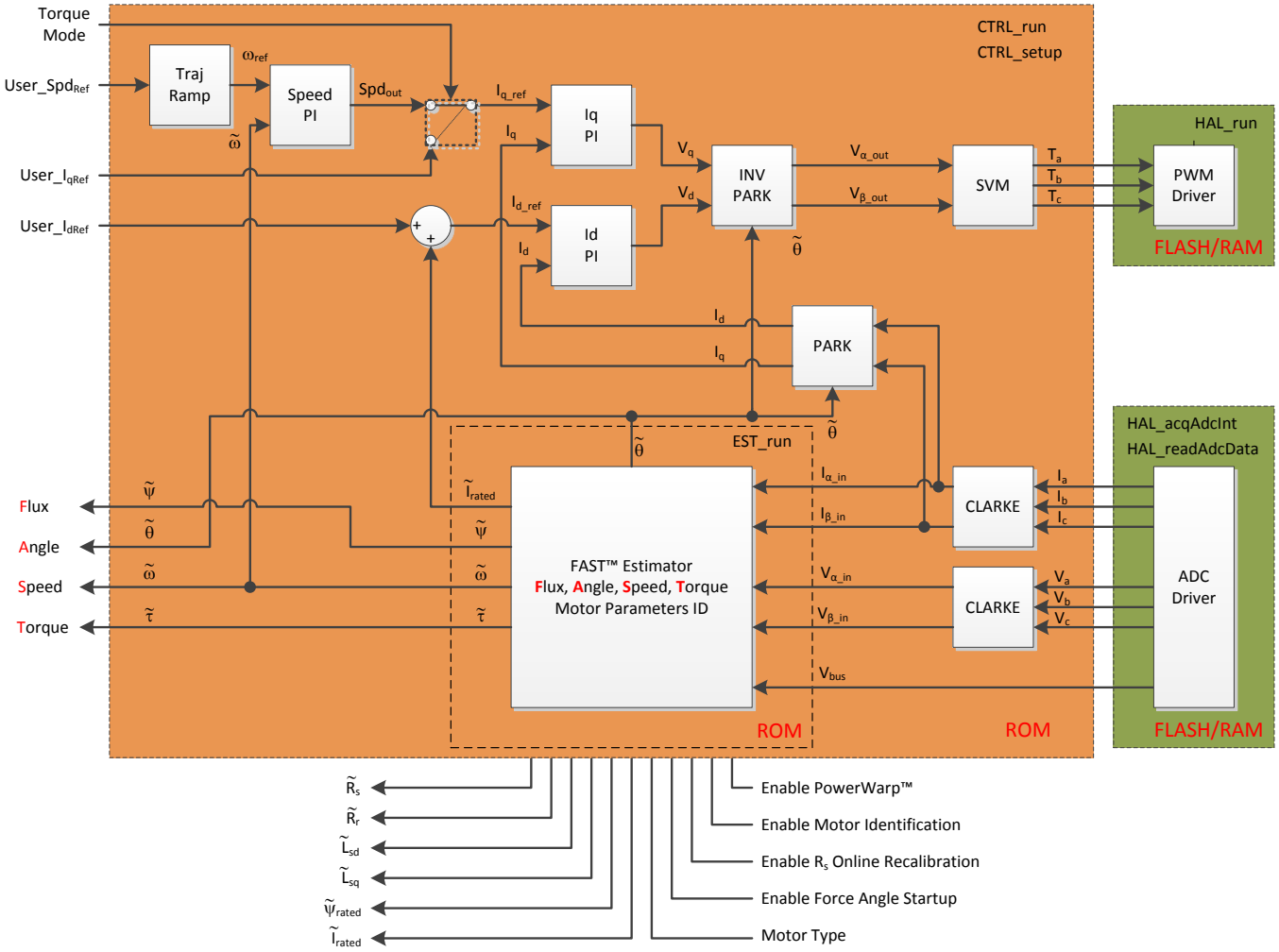


Figure 1-2. i. Block Diagram of Entire InstaSPIN-FOC Package in ROM (except F2802xF devices)

Copyright © 2013–2019, Texas Instruments Incorporated

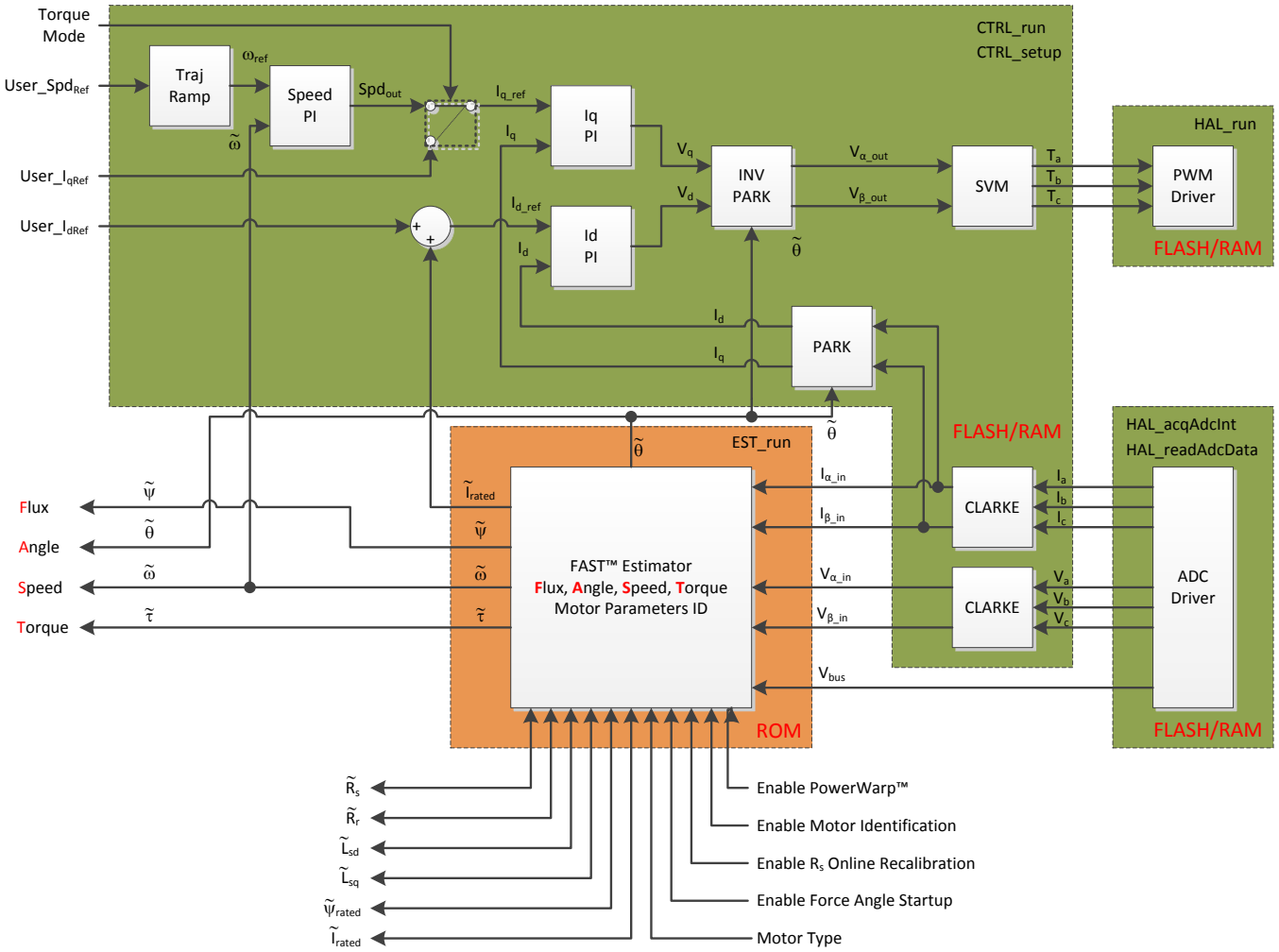


Figure 1-3. Block Diagram of InstaSPIN-FOC in User Memory, with Exception of FAST in ROM

Copyright © 2013–2019, Texas Instruments Incorporated

1.1.4 Comparing FAST Estimator to Typical Solutions

Table 1-1. FAST Estimator vs. Typical Solutions

Topic	Typical Software Sensors and FOC Solutions	TI's FAST Estimator and InstaSPIN-FOC Solution
Electrical Motor Parameters	Motor-model based observers heavily dependent on motor parameters	Relies on fewer motor parameters Off-line parameter identification of motor – no Datasheet required!On-line parameter monitoring and re-estimation of stator resistance
Estimator Tuning	Complex observer tuning done multiple times for speed/loads for each motor	No estimator tuning required. Once motor parameters are identified, it works the same way every time across speed/torque dynamics
Estimator Accuracy	Angle-tracking performance is typically only good at over 5-10Hz with challenges at higher speeds and compensation for field weakening; Dynamic performance influenced by hand tuning of observer; Motor stalls typically crash observer	FAST provides reliable angle tracking which converges within one electrical cycle of the applied waveform, and can track at less than 1 Hz frequency (dependent on quality and resolution of analog sensing)Angle tracking exhibits excellent transient response (even with sudden load transients which can stall the motor, thus enabling a controlled restart with full torque)
Start-up	Difficult or impossible to start from zero speedObserver feedback at zero speed is not stable, resulting in poor rotor angle accuracy and speed feedback	InstaSPIN-FOC includes: • Zero Speed start with forced-angle • 100% torque at start-up • FAST rotor flux angle tracking converges within one electrical cycle FAST is completely stable through zero speed, providing accurate speed and angle estimation.
Current Loop	Tuning FOC current control is challenging – especially for novices	Automatically sets the initial tuning of current controllers based on the parameters identified. User may update gains or use own controllers if desiredThe identification process to fully tune the observer and torque controller takes less than 2 minutes
Feedback Signals	System offsets and drifts are not managed	FAST includes automatic hardware/software calibration and offset compensationFAST requires 2-phase currents (3 for 100% and over-modulation), 3-phase voltages to support full dynamic performance, DCbus voltage for ripple compensation in current controllers FAST includes an on-line stator resistance tracking algorithm
Motor Types	Multiple techniques for multiple motors: standard back-EMF, Sliding Mode, Saliency tracking, induction flux estimators, or “mixed mode” observers	FAST works with all 3-phase motor types, synchronous and asynchronous, regardless of load dynamics. Supports salient IPM motors with different Ls-d and Ls-qIncludes PowerWarp for induction motors = energy savings
Field-Weakening	Field-weakening region challenging for observers – as Back-EMF signals grow too large, tracking and stability effected	FAST estimator allows easy field weakening or field boosting applications due to the stability of the flux estimation in a wide range
Motor Temperature	Angle tracking degrades with stator temperature changes	Angle estimation accuracy is improved from online stator resistance recalibration
Speed Estimation	Poor speed estimation causes efficiency losses in the FOC system and less stable dynamic operation	High quality low noise Speed estimator, includes slip calculation for induction motors
Torque Estimation	Torque and vibration sensors typically required	High bandwidth motor Torque estimator

Copyright © 2013–2019, Texas Instruments Incorporated

1.1.5 FAST Provides Sensorless FOC Performance

1.1.5.1 FAST Estimator Replaces Mechanical Sensor

Field Oriented Control (FOC) of an electric motor results in superior torque control, lower torque ripple, and in many cases, improved efficiency compared to traditional AC control techniques. For best dynamic response, rotor flux referenced control algorithms are preferred to stator flux referenced techniques. To function correctly, these systems need to know the spacial angle of the rotor flux with respect to a fixed point on the stator frame (typically the magnetic axis of the phase A stator coil). This has traditionally been accomplished by a mechanical sensor (e.g., encoder or resolver) mounted to the shaft of the motor. These sensors provide excellent angle feedback, but inflict a heavy toll on the system design. There are six major system impacts resulting from sensored angle feedback, as discussed below and illustrated in Section 3.5.4:

1. The sensor itself is very expensive (often over $2500 for a good resolver and several dollars for high volume integrated encoders)
2. The installation of the sensor requires skilled assembly, which increases labor costs
3. The sensor often requires separate power supplies, which increases system costs and reduces reliability
4. The sensor is the most delicate component of the system, which impacts system reliability, especially in harsh real-world applications
5. The sensor feedback signals are brought back to the controller board via connectors, which also increases system costs and can significantly reduce reliability, depending on the type of connector
6. The cabling required to bring the sensor signals back to the controller creates multiple challenges for the system designer:
   - Additional costs for the cable, especially if there is a substantial distance between the motor and controller
   - Susceptibility to sources of noise, which requires adding expense to the cable with special shielding or twisted pairs
   - The sensor and associated cabling must be earth grounded for safety reasons. This often adds additional cost to isolate these signals, especially if the processor which processes the sensor signals is not earth grounded

In some applications where the motor is enclosed (e.g., compressors), a sensored solution is impractical due to the cost of getting the feedback wires through the casing. For these reasons, designers of FOC systems are highly motivated to eliminate the sensor altogether, and obtain the rotor flux angle information by processing signals which are already available on the controller circuit board. For synchronous machines, most techniques involve executing software models of the motor being controlled to estimate the back-EMF waveforms (rotor flux), and then processing these sensed waveforms to extract an estimation of the rotor shaft angle, and a derivation of its speed. For asynchronous machines the process is a bit more complicated, as this software model (observer) must also account for the slip which exists between the rotor and rotor flux.

However, in both cases, performance suffers at lower speeds due to the amplitude of the back-EMF waveforms being directly proportional to the speed of the motor (assuming no flux weakening). As the back-EMF amplitude sinks into the noise floor, or if the ADC resolution cannot faithfully reproduce the small back-EMF signal, the angle estimation falls apart, and the motor drive performance suffers.

To solve the low-speed challenge, techniques have been created that rely on high frequency injection to measure the magnetic irregularities as a function of angle (i.e., magnetic saliency) to allow accurate angle reconstruction down to zero speed. However, this introduces another set of control problems. First, the saliency signal is non-existent for asynchronous motors and very small for most synchronous machines (especially those with surface mount rotor magnets). For the motors that do exhibit a strong saliency signal (e.g., IPM motors), the signal often shifts with respect to the rotor angle as a function of loading, which must be compensated. Finally, this angle measurement technique only works at lower speeds where the fundamental motor frequency does not interfere with the interrogation frequency. The control system has to create a mixed-control strategy, using high-frequency injection tracking at low speed, then move into Back-EMF based observers at nominal and high speeds.

Copyright © 2013–2019, Texas Instruments Incorporated

With any technique, the process of producing a stable software sensor is also extremely challenging, as this motor model (observer) is essentially its own control system that needs to be tuned per motor across the range of use. This tuning must be done with a stable forward control loop. Needed is a stable torque (and usually speed) loop to tune the observer, but how do you pre-tune your forward control without a functioning observer? One option is to use a mechanical sensor for feedback to create stable current and speed loops, and then tune your software sensor in parallel to the mechanical sensor. However, the use of a mechanical sensor is often not practical. This problem has delayed market use of software sensors for sensorless FOC control.

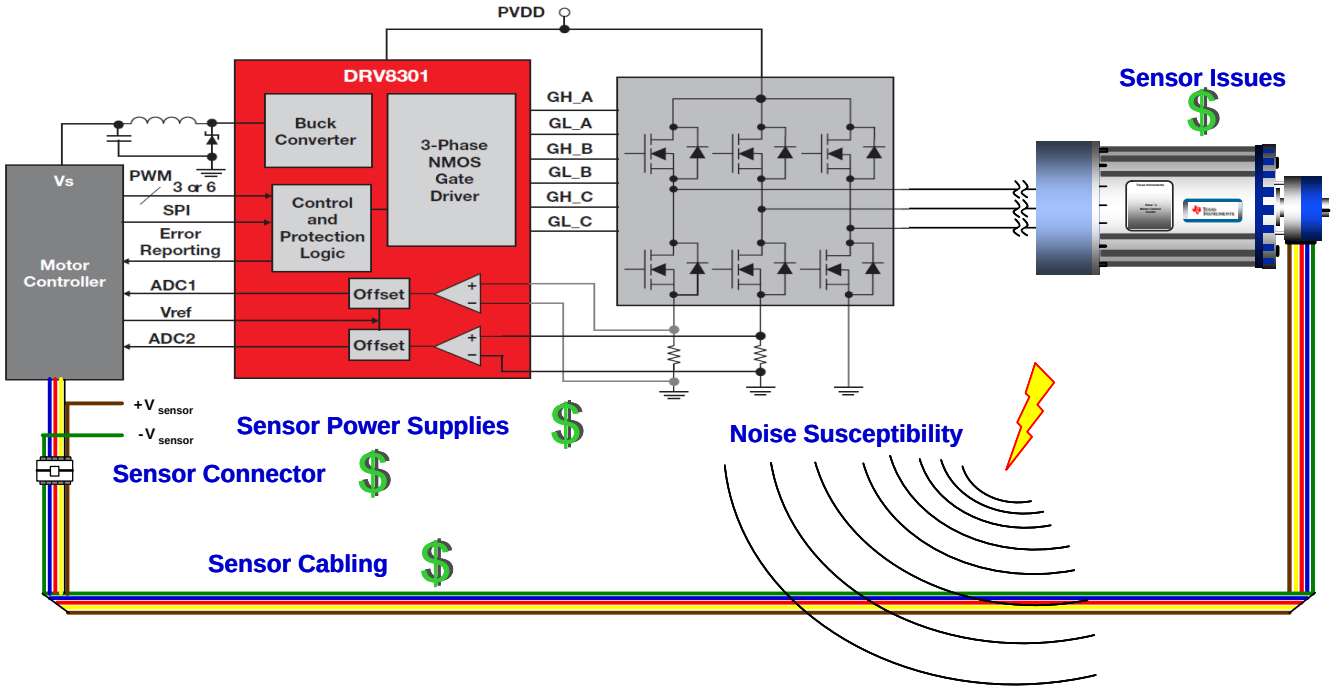


Figure 1-4. Sensored FOC System

In summary, these existing solutions all suffer from various maladies including:

- Poor low speed performance (back-EMF and SMO)
- Poor high speed performance (saliency observers)
- Poor dynamic response
- Calculation intensive (multi-modal observers)
- Parameter sensitivity
- Requirement for observer tuning

The most recent innovation in the evolution of sensorless control is InstaSPIN-FOC. Available as a C-callable library embedded in on-chip ROM on several TI processors, InstaSPIN-FOC was created to solve all of these challenges, and more. It reduces system cost and development time, while improving performance of three-phase variable speed motor systems. This is achieved primarily through the replacement of mechanical sensors with the proprietary FAST estimator. FAST is an estimator that:

- Works efficiently with all three phase motors, taking into account the differences between synchronous/asynchronous, salient/non-salient, and permanent/non-permanent/induced magnets
- Dramatically improves performance and stability across the entire operating frequency and load range for a variety of applications
- Removes the manual tuning challenge of traditional FOC systems:
  - observers and estimators, completely removes required tuning
  - current loop regulators, dramatically reduces required tuning

Copyright © 2013–2019, Texas Instruments Incorporated

- Eliminates or reduces motor parameter variation effects
- Automatically designs a stable and functional control system for most motors in under two minutes.

1.1.5.2 Rotor Angle Accuracy Critical for Performance

Why has the need for a precise estimation of the rotor flux angle driven many to use mechanical sensors?

For efficient control of three-phase motors, the objective is to create a rotating flux vector on the *stator* aligned to an *ideal orientation* with respect to the rotor in such a way that the *rotor* field follows the stator field while creating necessary torque and using the minimum amount of current.

- Stator: stationary portion of the motor connected to the microprocessor-controlled inverter
- Ideal Orientation: 90 degrees for non-salient synchronous; slightly more for salient machines, and slightly less in asynchronous machines since part of the current vector is also used to produce rotor flux
- Rotor: rotating portion of the motor, produces torque on the shaft to do work

  To achieve this, you need to extract the following information from the motor:
- Current being consumed by each phase
- Precise relative angle of the rotor flux magnetic field (usually within ±3 electrical degrees), so you can orient your stator field correctly
- For speed loops, you also need to know rotor speed.

1.2 An Overview of InstaSPIN-MOTION and SpinTAC

InstaSPIN-MOTION [TMS320F2806xM (69M and 68M — 80- or 100-pin packages) and TMS320F2805xM (54M and 52M, — 80-pin packages)] is the first offering from Texas Instruments to combine TI's 32-bit C2000™ Piccolo™ microcontrollers with comprehensive motor-, motion-, speed-, and position-control software. InstaSPIN-MOTION delivers robust velocity and position control at the highest efficiency for motor applications that operate in various motion state transitions. InstaSPIN-MOTION is your own motion control expert, on a single chip.

InstaSPIN-MOTION is a sensorless or sensored FOC solution that can identify, tune, and control your motor in minutes. InstaSPIN-MOTION features the FAST premium software sensor and the SpinTAC Motion Control Suite (Figure 1-5). The core algorithms are embedded in the read-only-memory (ROM) on TI's 32-bit C2000 Piccolo microcontrollers (MCUs).

Copyright © 2013–2019, Texas Instruments Incorporated

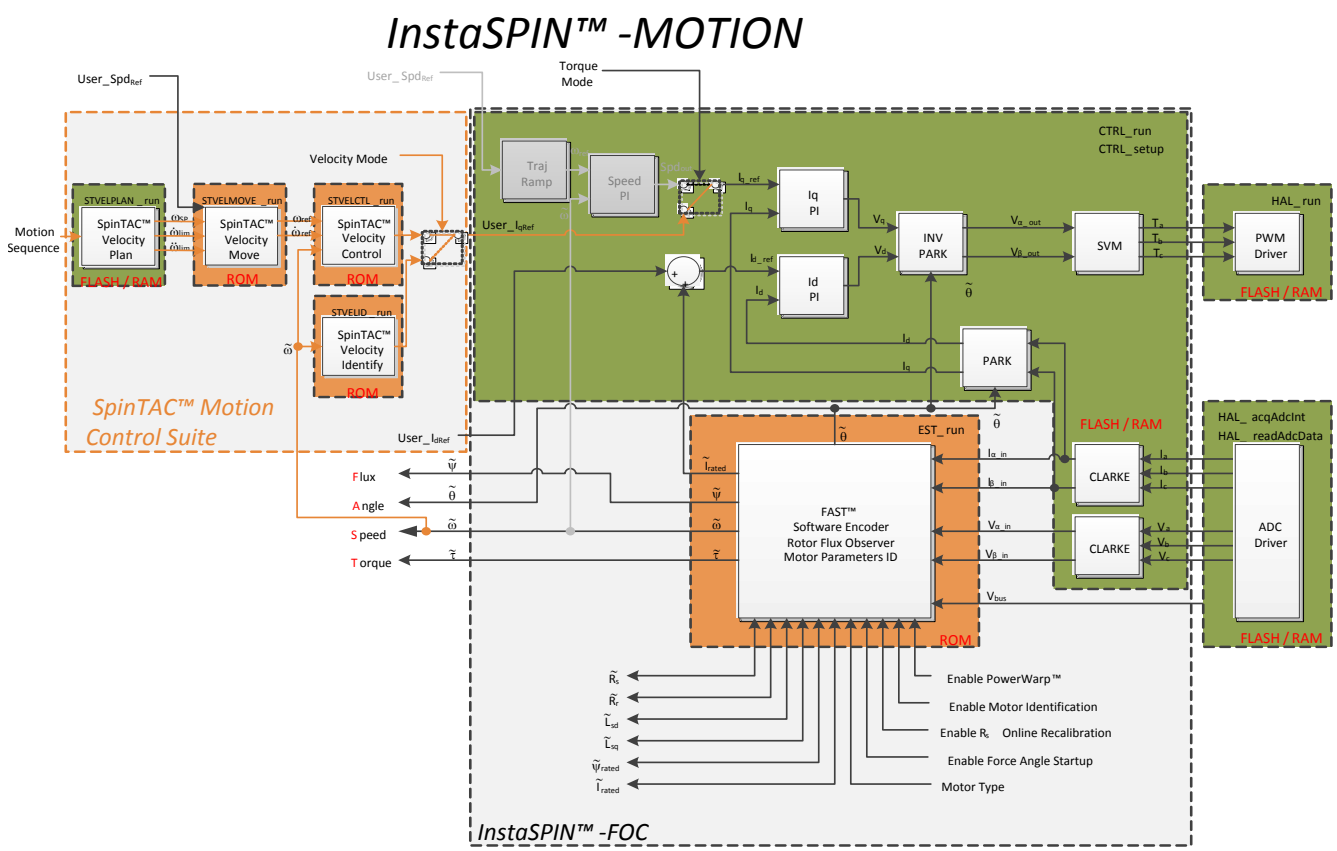


Figure 1-5. InstaSPIN-MOTION = C2000 Piccolo Microcontroller + FAST Software Sensor (optional) + Auto-Tuned Inner Torque Controller + SpinTAC Motion Control Suite

InstaSPIN-MOTION is ideal for applications that require accurate speed and position control, minimal disturbance, or undergo multiple state transitions or experience dynamic speed or load changes.

Table 1-2 provides examples of applications that will most benefit from InstaSPIN-MOTION.

Table 1-2. InstaSPIN-MOTION Application Examples

Application Characteristics	Examples
Accurate speed control	• Industrial fans • Conveyor systems • Elevators/escalators • Automotive body parts (electric windows, sunroofs, etc.) • Optical disc drives/hard drives • Medical mixing
Accurate position control	• Surveillance systems • Packaging systems • Medical robots • Gimbal systems • Textile/Sewing machines
Minimal disturbance	• Dental tools • Power tools • Security gates and doors

Copyright © 2013–2019, Texas Instruments Incorporated

Table 1-2. InstaSPIN-MOTION Application Examples (continued)

Application Characteristics	Examples
Undergoes multiple state transitions/dynamic changes	• HVAC pumps, fans and blowers • Generators • Air conditioning compressors • Washing machines • Exercise equipment • Medical pumps

1.2.1 InstaSPIN-MOTION Key Capabilities and Benefits

InstaSPIN-MOTION replaces inefficient, older design techniques with a solution that maximizes system performance and minimizes design effort. By embedding the motor expertise on the chip, InstaSPIN-MOTION enables users to focus on optimizing their application rather than struggling with motion control.

InstaSPIN-MOTION provides the following core capabilities:

- 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, angle, and speed) as well as shaft torque in a sensorless FOC system.
- Motor parameter identification, used to tune the FAST software observer and initialize the innermost current (torque) PI controllers for Iq and Id control of the FOC system.
- SpinTAC, a comprehensive motion control suite (see Figure 1-6) from LineStream Technologies, simplifies tuning and ensures optimal performance across dynamic speed and position ranges.

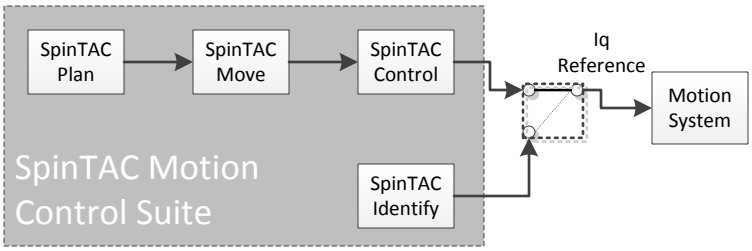


Figure 1-6. SpinTAC Motion Control Suite Components

1.2.1.1 The FAST Unified Software Observer

The FAST unified observer structure exploits the similarities between all motors that use magnetic flux for energy transduction:

- Supports both synchronous (BLDC, SPM, IPM), and asynchronous (ACIM) control.
- Provides salient compensation for interior permanent magnet motors: observer tracks rotor flux and angle correctly when Ls-d and Ls-q are provided.

FAST offers unique, high-quality motor feedback signals for control systems:

- High-quality Flux signal for stable flux monitoring and field weakening.
- Superior rotor flux Angle estimation accuracy over wider speed range compared to traditional observer techniques independent of all rotor parameters for ACIM.
- Real-time low-noise motor shaft Speed signal.
- Accurate high-bandwidth Torque signal for load monitoring and imbalance detection.

FAST replaces mechanical encoders and resolvers and accelerates control system design:

- Angle estimator converges within first cycle of the applied waveform, regardless of speed.
- Stable operation in all power quadrants, including generator quadrants.
- Accurate angle estimation at steady state speeds below 1 Hz (typ) with full torque.

Copyright © 2013–2019, Texas Instruments Incorporated

- Angle integrity maintained even during slow speed reversals through zero speed.
- Angle integrity maintained during stall conditions, enabling smooth stall recovery.
- Motor identification measures required electrical motor parameters of unloaded motor in under 2 minutes (typ).
- *On-the-fly* stator resistance recalibration (online Rs) tracks stator resistance changes in real time, resulting in robust operation over temperature. This feature can also be used as a temperature sensor of the motor's windings (basepoint calibration required).
- Superior transient response of rotor flux angle tracking compared to traditional observers.
- PowerWarp adaptively reduces current consumption to minimize the combined (rotor and stator) copper losses to the lowest, without compromising ACIM output power levels.

1.2.1.2 The SpinTAC Motion Control Suite

SpinTAC minimizes the time you spend defining how you want your motor to spin and ensures that your motor runs at its optimal level for ideal performance. Key benefits include:

- **Simplified Tuning** - Tune your system for the entire position and speed operating range with a single, easy-to-evaluate parameter.
- **Intuitive Trajectory Planning** - Easily design and execute complex motion sequences.
- **Mechanically Sound Movement** - Optimize your transitions between speeds based on your system's mechanical limitations.
- **Ideal Control** - Benefit from the most accurate speed and position control on the market, based on LineStream's patented Active Disturbance Rejection Control.

There are four components that comprise the SpinTAC Motion Control Suite: Identify, Control, Move, and Plan. Each of these components exist for both the Velocity and Position solution.

1.2.1.2.1 IDENTIFY

SpinTAC Identify estimates inertia (the resistance of an object to rotational acceleration around an axis). The greater the system inertia, the greater the torque needed to accelerate or decelerate the motor. The SpinTAC controller uses the system's inertia value to provide the most accurate system control. SpinTAC Identify automatically measures system inertia by spinning the motor in the application and measuring the feedback.

1.2.1.2.2 CONTROL

SpinTAC Control is an advanced speed and position controller featuring Active Disturbance Rejection Control (ADRC), which proactively estimates and compensates for system disturbance, in real time. SpinTAC automatically compensates for undesired system behavior caused by:

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

SpinTAC 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 system performance and minimizes mechanical system duress.

With single coefficient tuning, the SpinTAC controller allows users to quickly test and tune their velocity and position control from soft to stiff response. This single gain (bandwidth) typically works across the entire variable speed, position and load range of an application, reducing complexity and system tuning time typical in multi-variable PI-based systems. A single parameter controls both position and speed. These systems often require a dozen or more tuned coefficient sets to handle all possible dynamic conditions.

Copyright © 2013–2019, Texas Instruments Incorporated

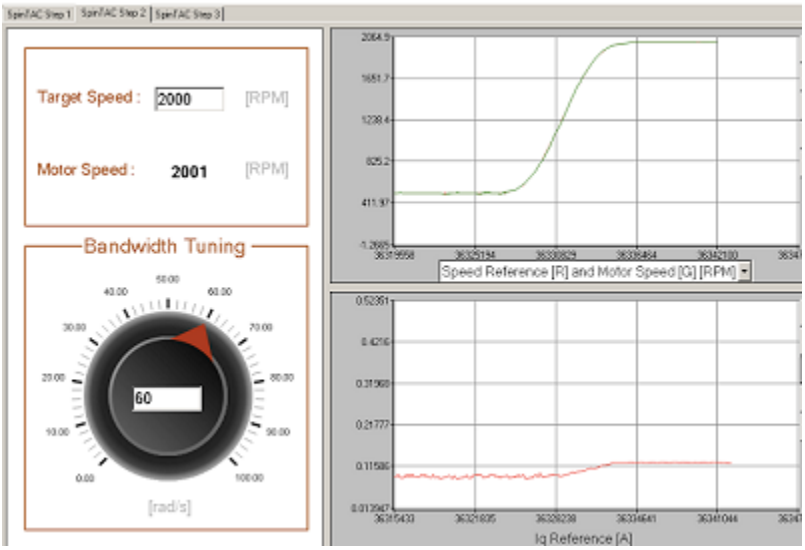


Figure 1-7. Simple Tuning Interface

The InstaSPIN-MOTION (F2805xM and F2806xM) GUI (see Figure 1-7), in conjunction with the InstaSPIN-MOTION Quick Start Guide, allow users to quickly evaluate InstaSPIN-MOTION (speed control) using TI's evaluation kits, the TI provided motors, or their own motor. The GUI is designed to quickly guide you through the InstaSPIN-MOTION evaluation process. You can obtain the GUI, free of charge, from ti.com/tool/motorkitscncd69miso. Once you determine that InstaSPIN-MOTION is right for your application, use the MotorWare-based projects, in conjunction with this document to design your project and conduct performance testing.

1.2.1.2.3 MOVE

SpinTAC Move provides an easy way to smoothly transition from one speed or position to another by computing the fastest path between Point A and Point B. SpinTAC Move generates a profile based on starting velocity or position, desired velocity or position, and configured system limitations for acceleration and 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 points, can cause systems to oscillate. The bigger the step, the greater this tendency. 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.

As opposed to pre-defined lookup tables, SpinTAC Move runs on the processor, consuming less memory than traditional solutions. Besides the industry standard trapezoidal curve and s-Curve, SpinTAC also provides a proprietary st-Curve, which is even smoother than s-Curve and allows users to limit the jerk of the motion.

Signals	Trapezoidal	s-Curve	st-Curve
Position	Smooth	Smooth	Smooth
Velocity	Continuous	Smooth	Smooth
Acceleration	Bounded	Continuous	Smooth
Jerk	Infinite	Bounded	Continuous

Figure 1-8. Curves Available in SpinTAC Move

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 1-8 describes the curves that are available for use in SpinTAC Move. The LineStream proprietary 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.

1.2.1.2.4 PLAN

SpinTAC Plan provides easy design and execution of complex motion sequences. The trajectory planning feature allows users to quickly build various states of motion (point A to point B) and tie them together with state based logic. SpinTAC Plan can be used to implement a motion sequence for nearly any application. Figure 1-9 displays the motion sequence for a washing machine and Figure 1-10 displays the motion sequence for a garage door. Both of these were easily designed using SpinTAC Plan. Once designed, the trajectories are directly embedded into the C code on the microcontroller.

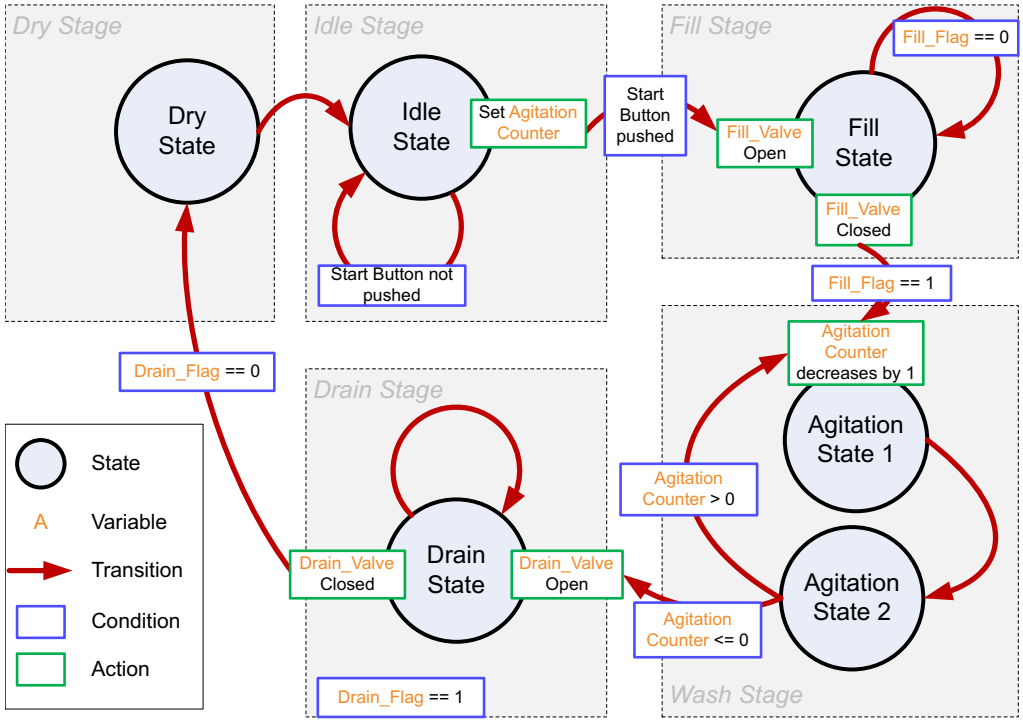


Figure 1-9. State Transition Map for a Washing Machine

Copyright © 2013–2019, Texas Instruments Incorporated

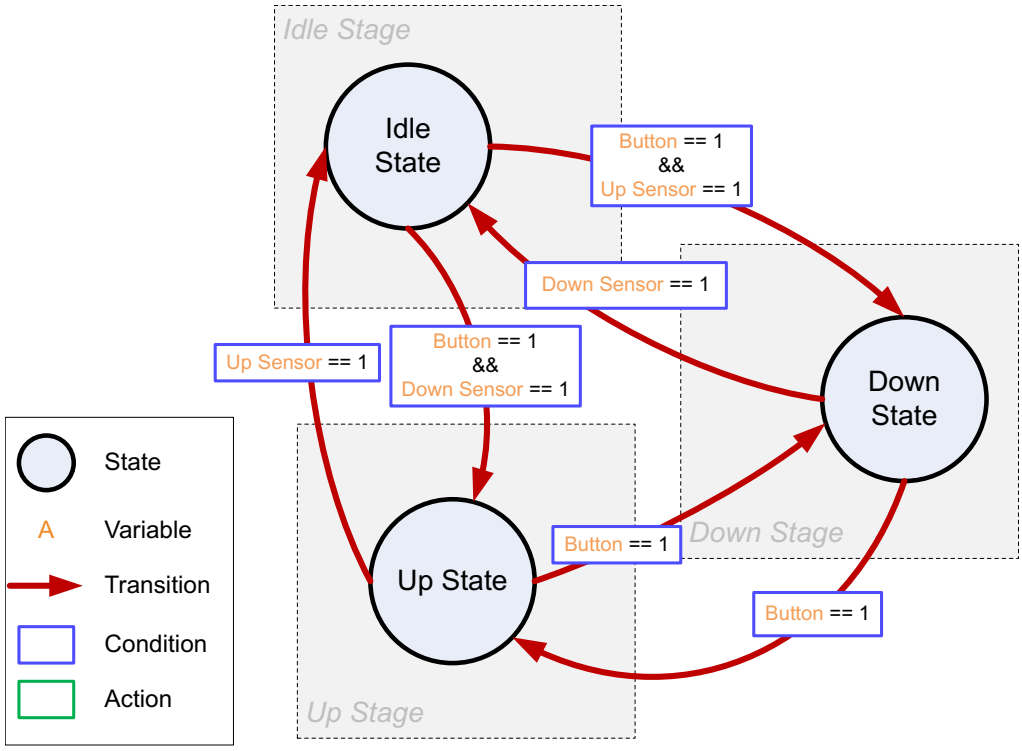


Figure 1-10. State Transition Map for a Garage Door System

1.2.1.3 Additional InstaSPIN-MOTION Features

- Automatic torque (current) loop tuning with option for user adjustments
- Automatic or manual field weakening and field boosting
- Bus voltage compensation
- Automatic offset calibration ensures quality samples of feedback signals

1.2.2 InstaSPIN-MOTION Block Diagrams

InstaSPIN-MOTION is designed in a modular structure. Customers can determine which functions will be included in their system. The FAST Observer resides in ROM. The core control algorithms of the SpinTAC library reside in ROM, and these functions are accessed by application program interface (API) from the user code.

InstaSPIN-MOTION supports a wide array of system designs. InstaSPIN-MOTION uses the FAST software encoder for sensorless FOC systems [for additional information, see the *TMS320F2802xF InstaSPIN-FOC Technical Reference Manual* (SPRUHP4), the *TMS320F2805xF InstaSPIN-FOC Technical Reference Manual* (SPRUHW0), and the *TMS320F2806xF InstaSPIN-FOC Technical Reference Manual* (SPRUHI9)]. InstaSPIN-MOTION also supports solutions that leverage mechanical sensors (e.g., encoders, resolvers). These scenarios are described below.

Copyright © 2013–2019, Texas Instruments Incorporated

Note that the variables used in Section 8.6, Figure 1-12, Figure 1-13, and Figure 1-14 are defined as follows:

- θ_{Qep}: position angle signal from encoder
- θ_{M}: formatted sawtooth position signal to be used in SpinTAC Position Controller
- θ_{SP}: Sawtooth position reference signal generated by SpinTAC Position Move
- ω_{lim}: Speed Limit (used in position profile generation)
- $\dot{\omega}_{lim}$: Acceleration Limit
- $\dot{\omega}_{lim}$: Jerk Limit
- ω_{Ref}: Speed Reference
- $\dot{\omega}_{Ref}$: Acceleration Reference
- $\tilde{\tau}_{r}$: Motor time constant

Scenario 1: InstaSPIN-MOTION Speed Control with FAST Software Encoder

In this scenario (see Section 8.6 and Figure 1-12), SpinTAC Velocity Control receives the speed estimate from the FAST estimator and generates the torque reference signal. This works with InstaSPIN-MOTION in user memory (see Section 8.6) or in ROM (see Figure 1-12). The SpinTAC Motion Control Suite provides the motion sequence state machine, generates the reference trajectory and controls the system speed.

Copyright © 2013–2019, Texas Instruments Incorporated

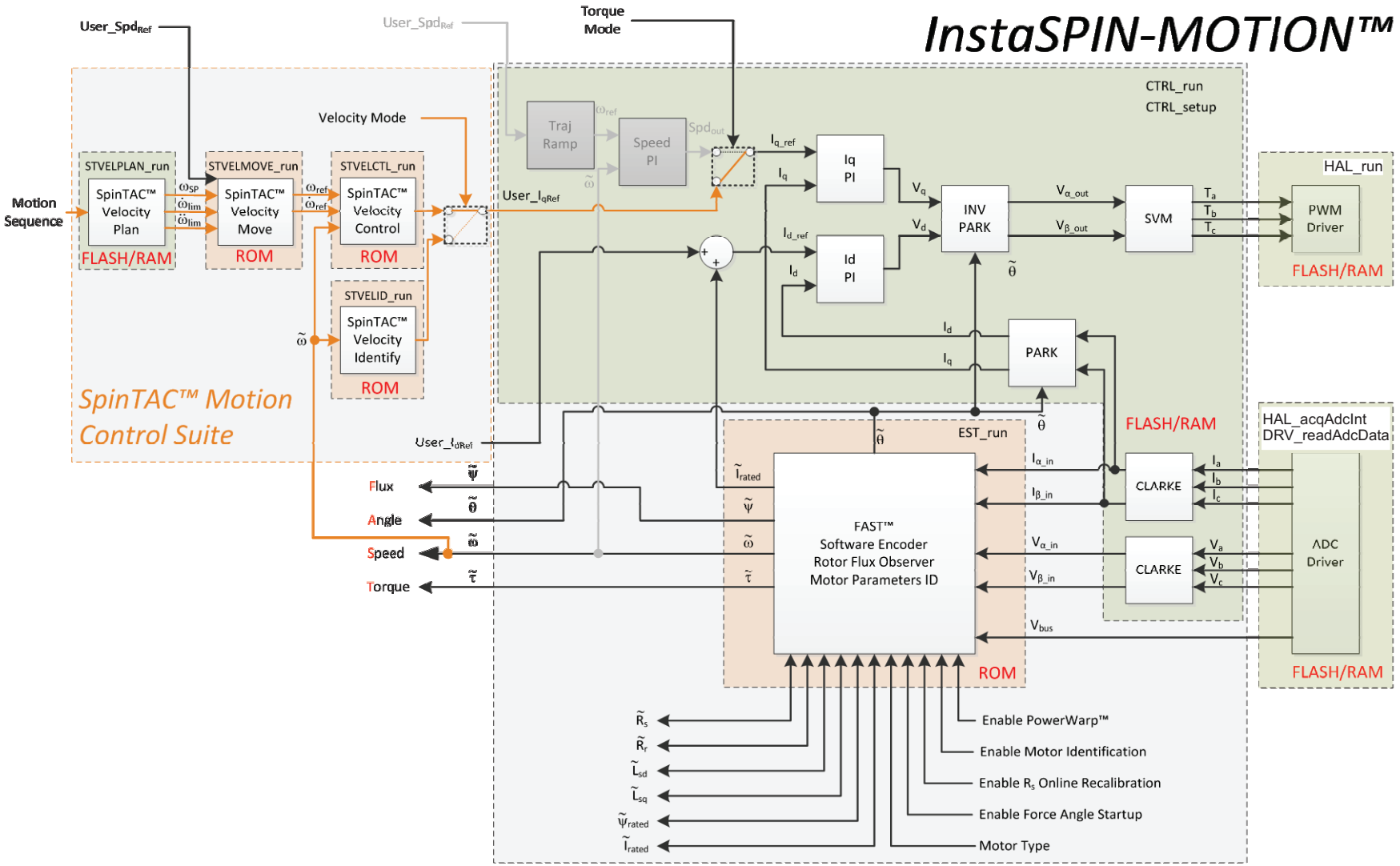


Figure 1-11. InstaSPIN-MOTION in User Memory, with Exception of FAST and SpinTAC in ROM

Copyright © 2013–2019, Texas Instruments Incorporated

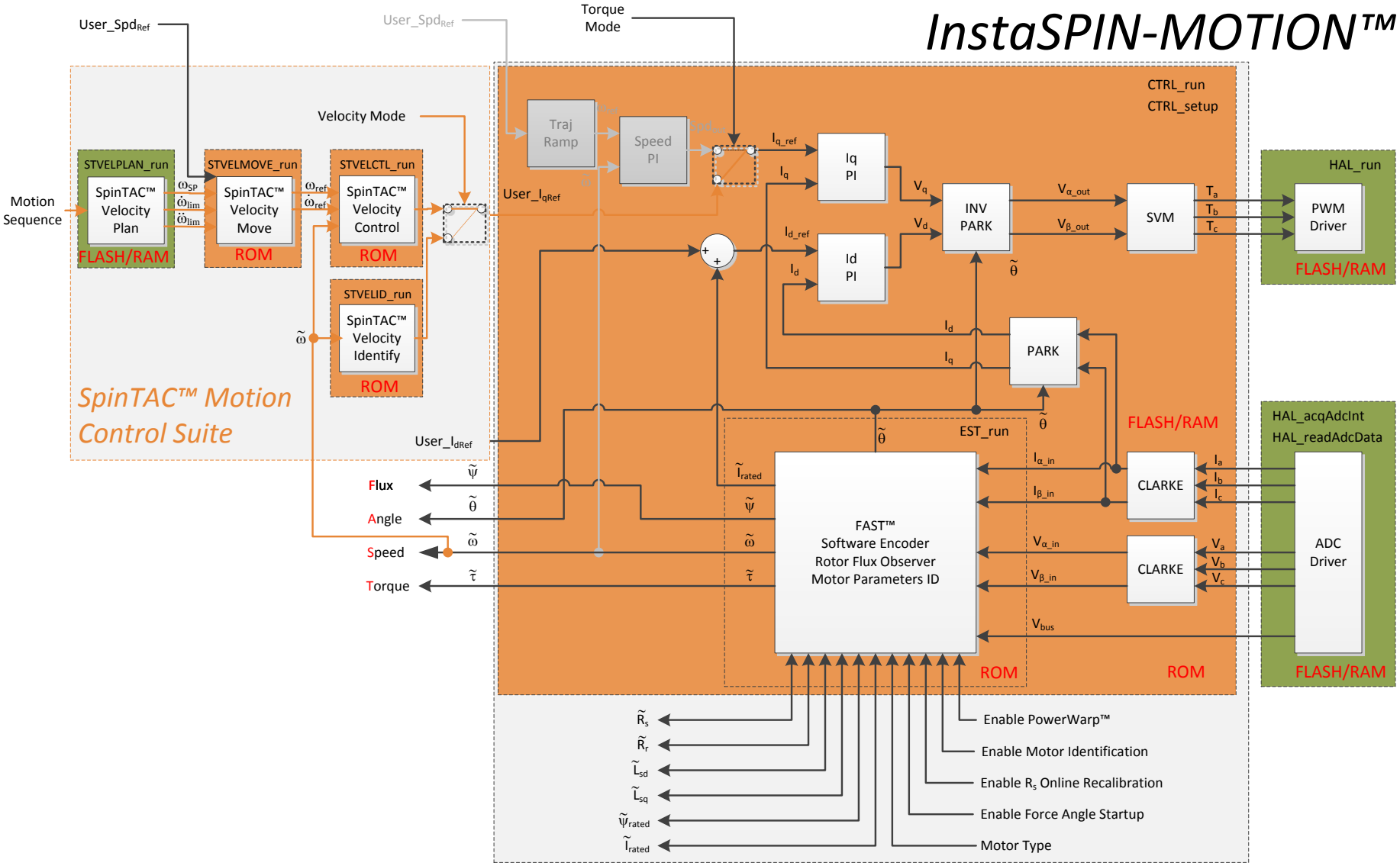


Figure 1-12. InstaSPIN-MOTION in ROM

Copyright © 2013–2019, Texas Instruments Incorporated

Scenario 2: InstaSPIN-MOTION Speed Control with a Mechanical Sensor

While sensorless solutions are appealing and cost effective for many applications, there are some applications that require the rigor and accuracy of a mechanical sensor. For these applications (see Figure 1-13), the quadrature encoder provides position information, which is then converted to speed feedback via the SpinTAC Position Converter. SpinTAC Velocity Control receives the speed feedback and generates the torque reference signal via IqRef. The SpinTAC Motion Control Suite provides the motion sequence state machine, generates the reference trajectory and controls the system speed.

Copyright © 2013–2019, Texas Instruments Incorporated

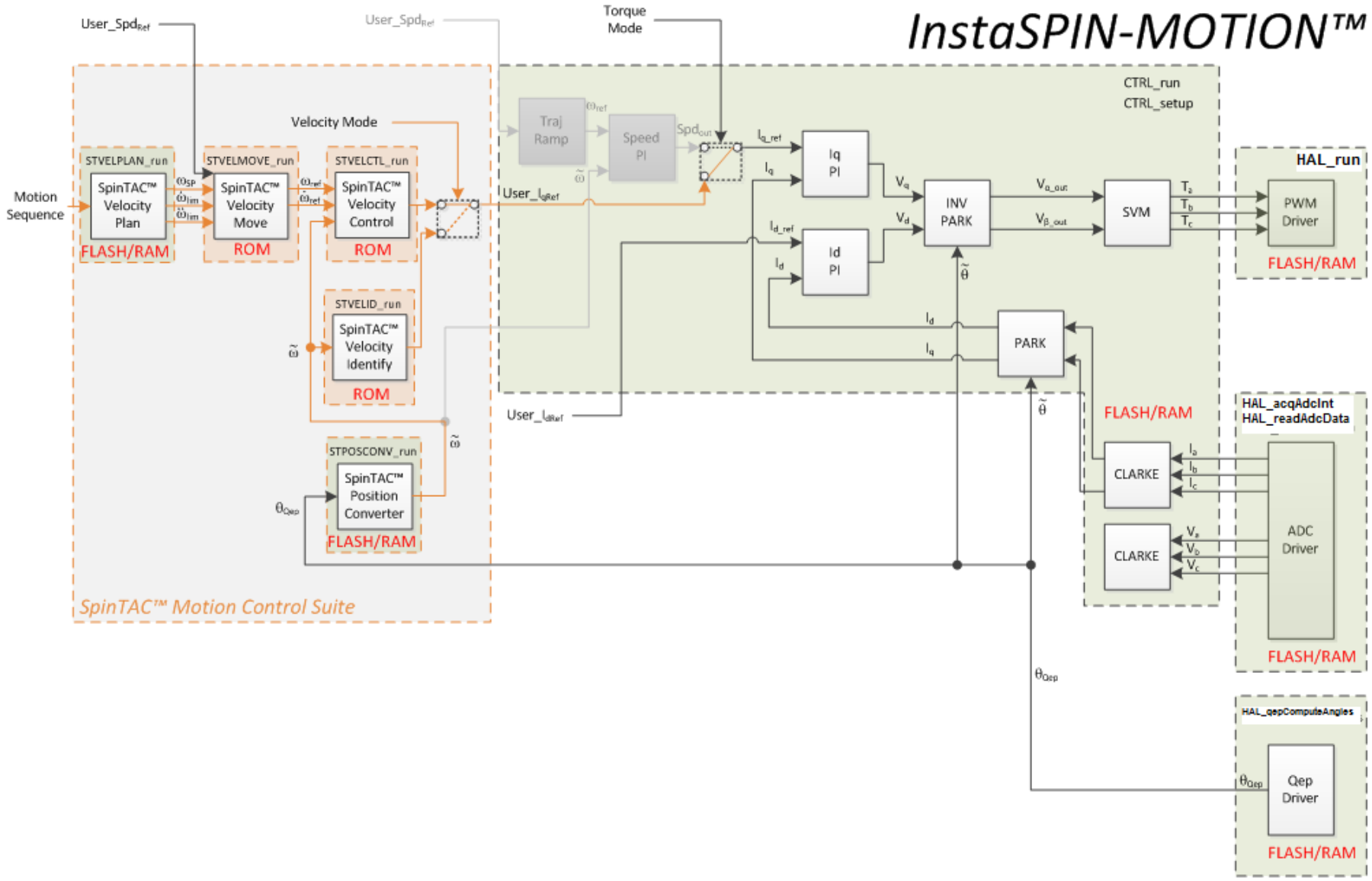


Figure 1-13. InstaSPIN-MOTION Speed Control with a Mechanical Sensor

Copyright © 2013–2019, Texas Instruments Incorporated

Scenario 3: InstaSPIN-MOTION Position Control with Mechanical Sensor and Redundant FAST Software Sensor

There are many applications where precise position control is required. For these applications, it is difficult to balance the many tuning parameters that are required. InstaSPIN-MOTION features accurate position, speed, and torque control, with combined position and speed single-variable tuning. This simplifies the tuning challenge and allows you to focus on your application and not on tuning your motor. Position applications require a mechanical sensor in order to precisely identify the motor angle at zero and very low speeds. The FAST Software Encoder may provide redundancy in position control applications; this can be used as a safety feature in case the mechanical encoder fails (see Figure 1-14).

Copyright © 2013–2019, Texas Instruments Incorporated

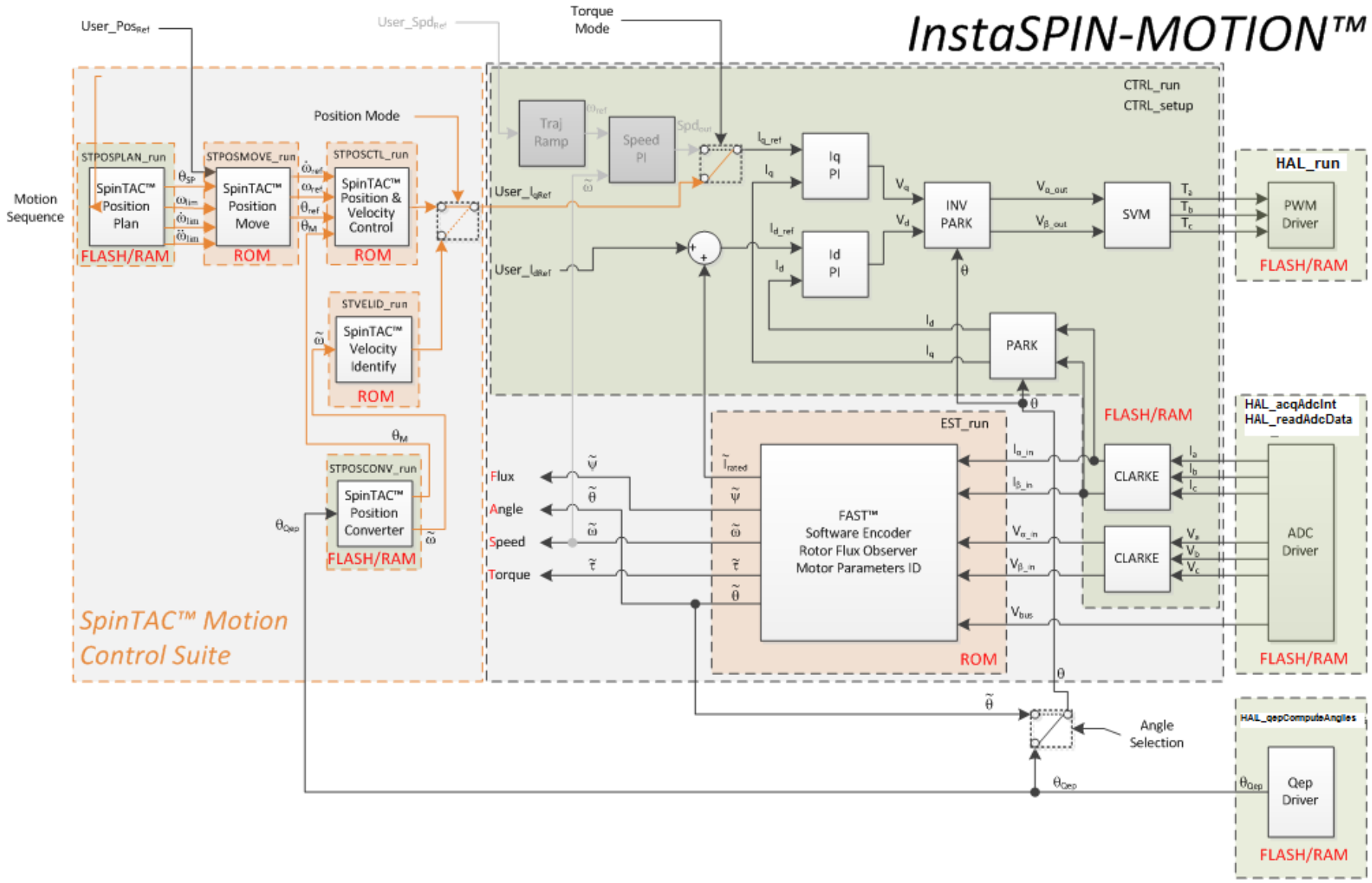


Figure 1-14. InstaSPIN-MOTION Position Control with Mechanical Sensor and Redundant FAST Software Sensor

Copyright © 2013–2019, Texas Instruments Incorporated

1.2.3 Application Examples

InstaSPIN-MOTION is ideal for applications that require accurate speed and position control, minimal disturbance, and for applications that undergo multiple state transitions or experience dynamic changes. A few examples are provided below.

1.2.3.1 Treadmill Conveyor: Smooth Motion Across Varying Speeds and Loads

Consistent speed control is critical for treadmill conveyor belts. When a person runs on the treadmill, their stride disturbs the motion of the belt. The runner's stride will be choppy if the motor driving the belt cannot quickly provide enough torque to overcome the disturbance. This problem is exacerbated when the user changes speeds as part of their exercise regime. If the belt does not smoothly accelerate or decelerate it seems like the treadmill is not operating correctly. In addition, at low speeds when a user steps on the belt, their weight can cause the belt to stop.

InstaSPIN-MOTION was applied to a commercial treadmill using a 4-HP, 220-V AC induction motor to drive the conveyor belt. The treadmill was tested across a variable speed range: 42 rpm at the low end, to 3300 rpm at top speed.

The customer found that InstaSPIN-MOTION's advanced controller automatically compensated for disturbances, keeping the speed consistent while running, and across changing speeds. The controller prevented the belt from stopping at low speeds when a load was applied. In addition, a single gain was used to control the entire operating range.

1.2.3.2 Video Camera: Smooth Motion and Position Accuracy at Low Speeds

High-end security and conference room cameras operate at very low speeds (e.g., 0.1 rpm) and require accurate and smooth position control to pan, tilt, and zoom. The motors that drive these cameras are difficult to tune for low speed, and they usually require a minimum of four tuning sets. In addition, there can be choppy movement at startup, which results in a shaky or unfocused picture.

InstaSPIN-MOTION was applied to a high-precision security camera driven by a 2-pole BLDC motor with a magnetic encoder. InstaSPIN-MOTION was able to control both velocity and position using a single tuning parameter that was effective across the entire operating range. SpinTAC Move was used to control the motor jerk, resulting in smooth startup.

1.2.3.3 Washing Machine: Smooth Motion and Position Accuracy at Low Speeds

Cycle transitions, changing loads, and environmental disturbances cause significant wear and tear on motors. Automatic, real-time reduction of disturbances can extend the life and performance of motors.

Consider washing machines, for example. Figure 6-59 displays the motion profile for three stages of a standard washing machine. The first stage represents the agitation cycle, rotating between 250 rpm and -250 rpm repeatedly. The second and third stages represent two different spin cycles. The second stage spins at 500 rpm and the third stage spins at 2000 rpm. This profile was easily created using SpinTAC Plan.

Copyright © 2013–2019, Texas Instruments Incorporated

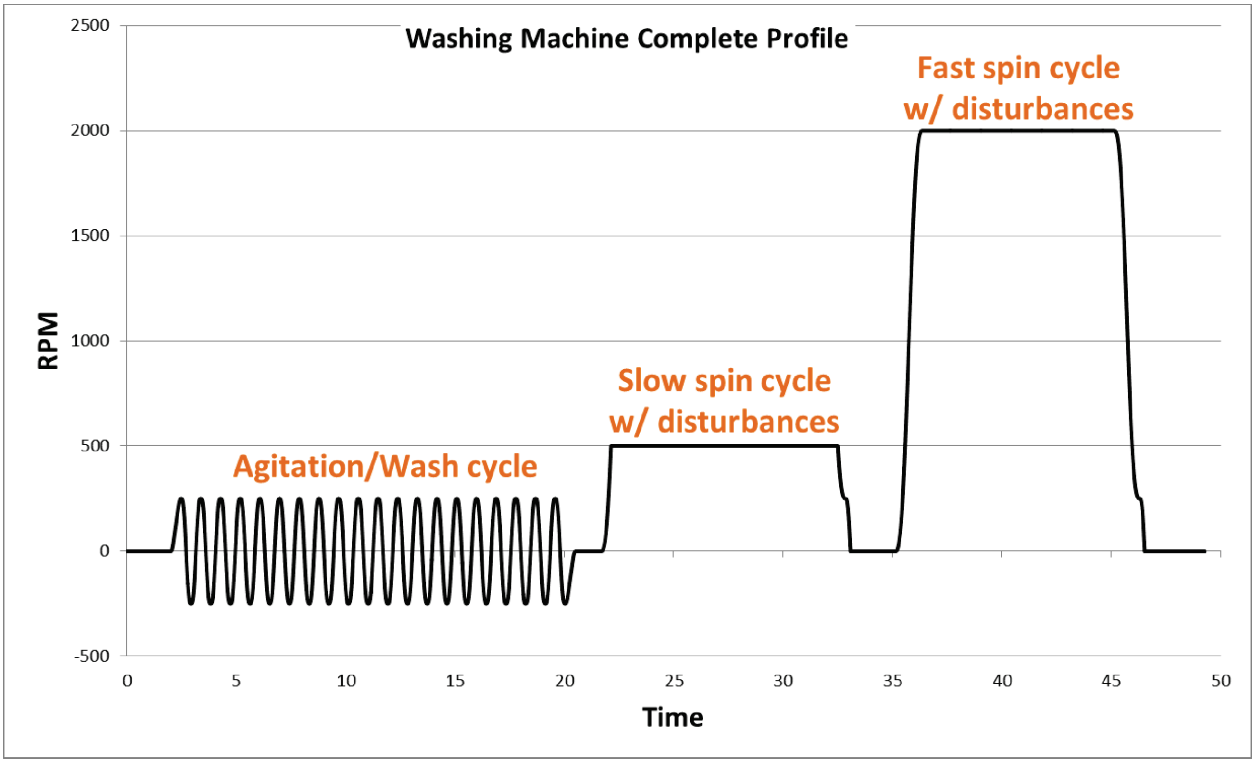


Figure 1-15. Washing Machine Profile

InstaSPIN-MOTION was applied to a washing machine application. The SpinTAC Plan trajectory planning feature was used to quickly build various states of motion (speed A to speed B) and tie them together with state based logic.

The washing machine application was run twice, once using a standard PI controller and once using LineStream's SpinTAC controller. The data was then plotted against the reference curve for comparison.

1.2.3.3.1 Agitation Cycle

During agitation, the motor switches between the 250 rpm and -250 rpm set points 20 times. The results, shown in Figure 1-16, that InstaSPIN-MOTION more closely matched the reference profile. Additionally, the maximum error for PI was 91 rpm (341 - 250 = 91 rpm), whereas the maximum error for InstaSPIN-MOTION was 30 rpm (280 - 250 = 30 rpm).

Copyright © 2013–2019, Texas Instruments Incorporated

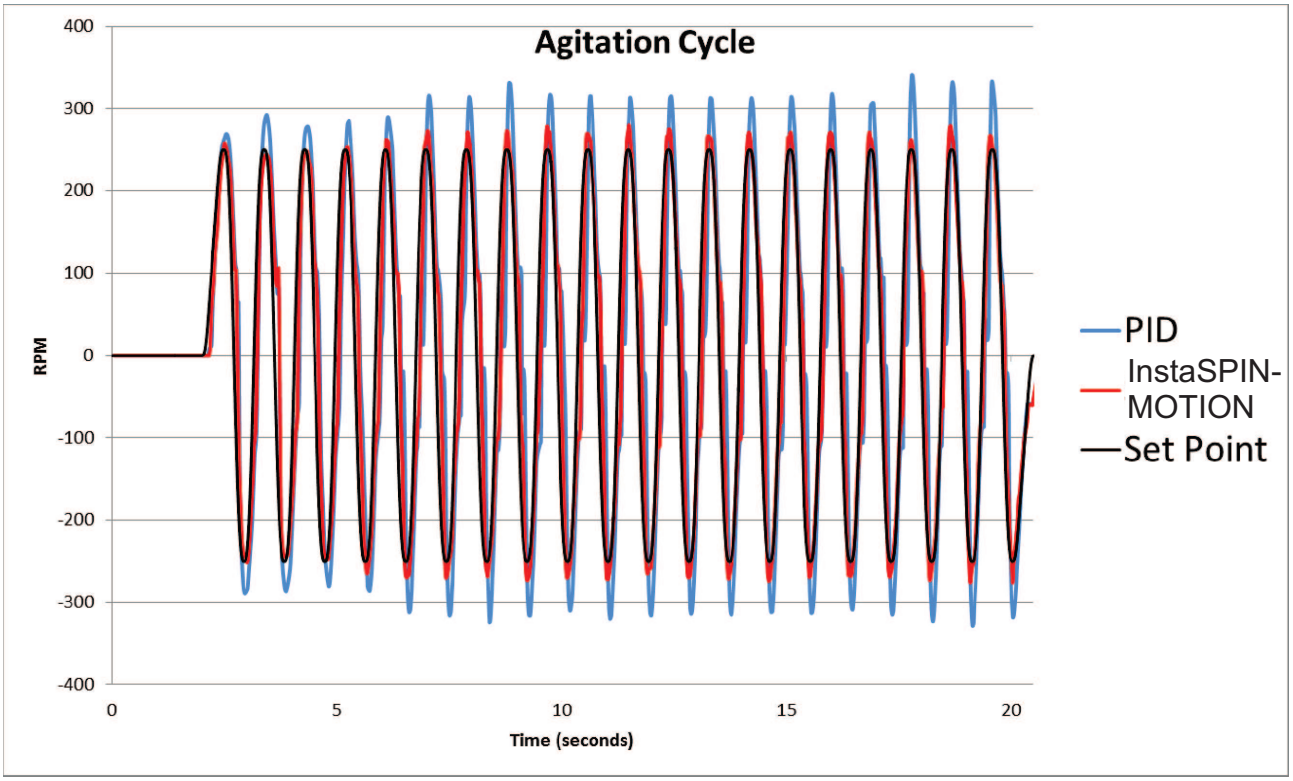


Figure 1-16. InstaSPIN-MOTION Minimizes Error

1.2.3.3.2 Spin Cycles

For the first spin cycle, the objective is to maintain 500 rpm, even when disturbances are introduced. Figure 7-7 shows that the InstaSPIN-MOTION recovered from disturbances more quickly and with less oscillation than the PI controller. Additionally, InstaSPIN-MOTION does not suffer from the overshoot and undershoot shown by the PI controller when it tries to reach the initial 500 rpm set point.

Copyright © 2013–2019, Texas Instruments Incorporated

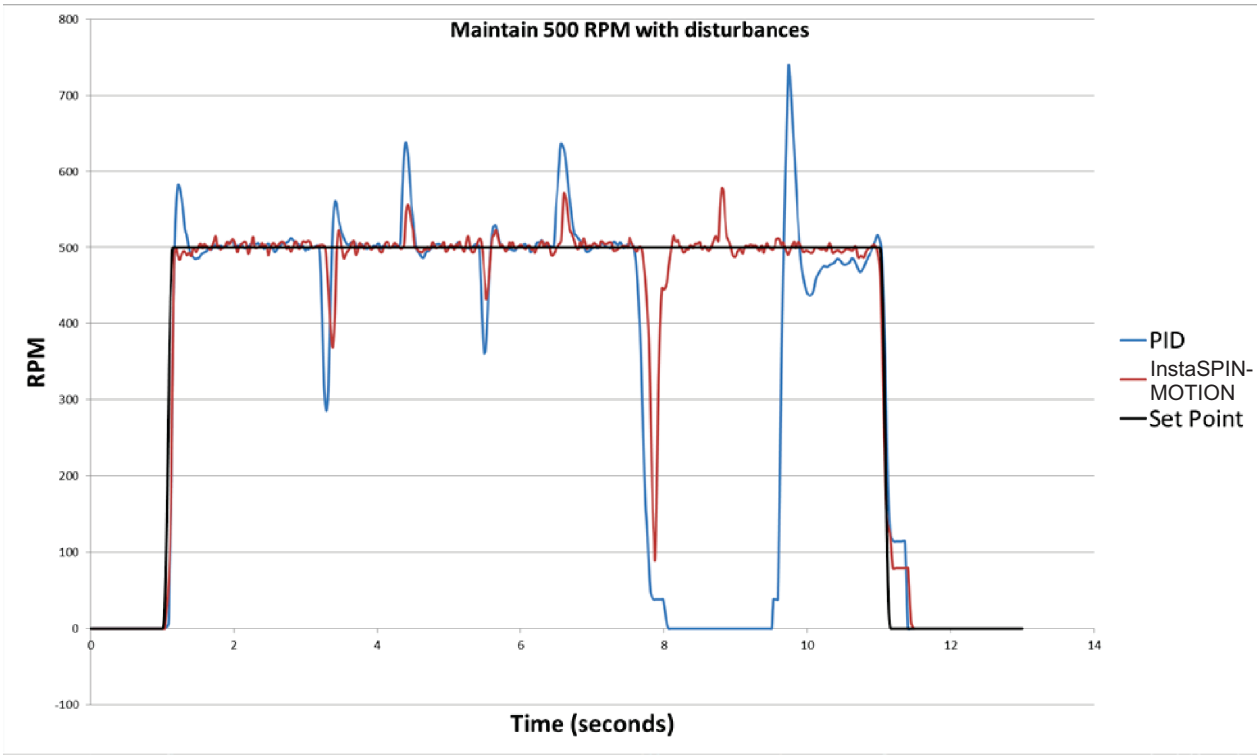


Figure 1-17. First Spin Cycle - 500 rpm

During the second spin cycle, shown in Figure 1-18, InstaSPIN-MOTION consistently recovered from disturbances at 2000 rpm more quickly and with less oscillation than the PI controller. Note that SpinTAC does not suffer from the overshoot and undershoot shown by the PI controller when it tries to reach the initial 2000 rpm set point.

Copyright © 2013–2019, Texas Instruments Incorporated

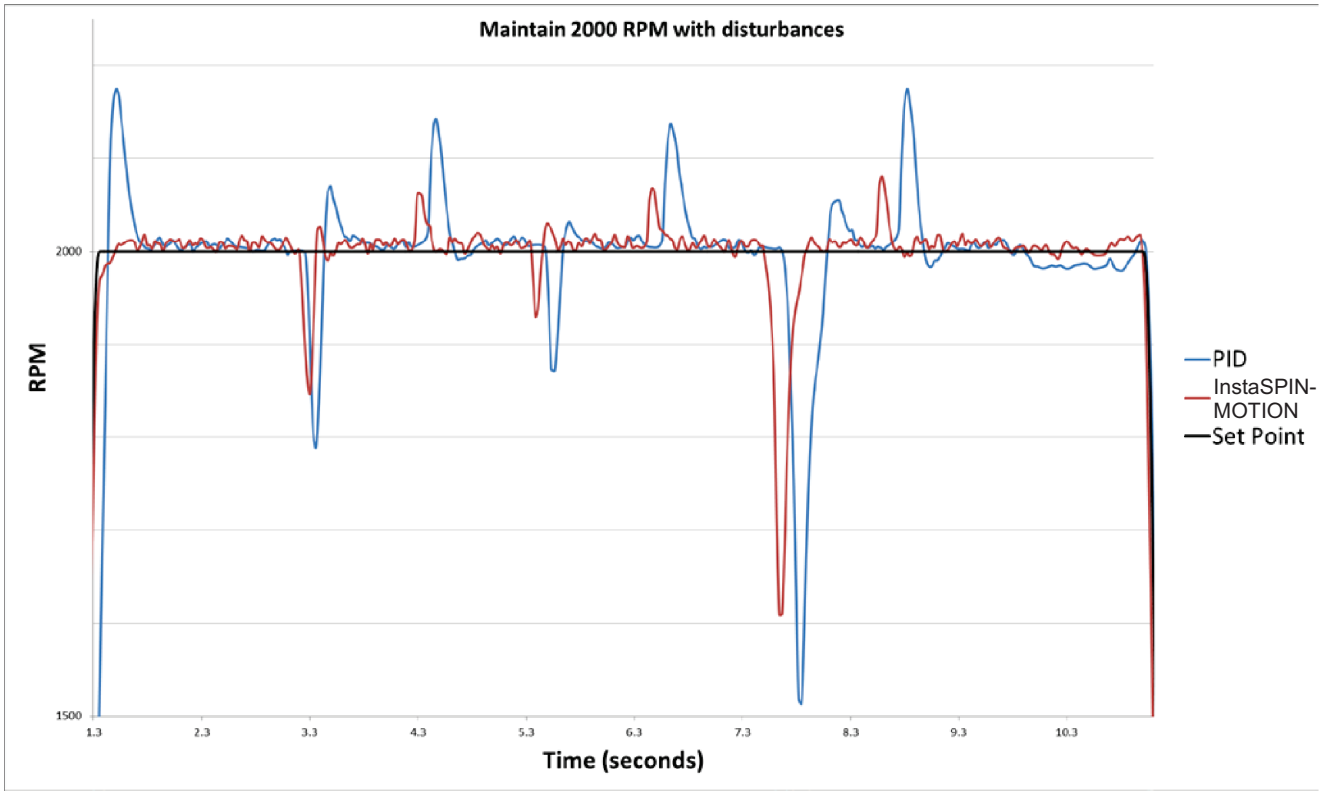


Figure 1-18. Second Spin Cycle - 2000 rpm

Additionally, the PI controller could not recover from the ramp disturbance at the 9.75 second mark. Instead, it shows a steady-state error of roughly 20 rpm.

1.2.3.3.3 InstaSPIN-MOTION Works Over the Entire Operating Range

Both the InstaSPIN-MOTION controller and the PI controller were tuned once, before executing the washing machine application. From the example, it is evident that InstaSPIN-MOTION's tuning works over the entire operating range. Whether the motor switches between the 250 rpm and -250 rpm, or maintains 500 rpm or 2000 rpm spin cycles, there is no need for new tuning sets.

Copyright © 2013–2019, Texas Instruments Incorporated

Quick Start Kits - TI Provided Software and Hardware

Several GUI-based applications are provided for a quick and easy way to evaluate InstaSPIN-FOC and InstaSPIN–MOTION software. Pre-configured graphical user interfaces (GUIs) are the quickest way to get started, providing a demonstration platform for the F2806xM and F2806xF devices. The universal GUI provides a GUI option for any MotorWare project that you can customize. You can use this GUI outside of Code Composer Studio or within the IDE. The *GUI Quick-Start Guides* will lead you through the details of the evaluation process. A simple overview of the pre-configured GUI is provided here: http://www.ti.com/tool/instaspinfocmotorwaregui

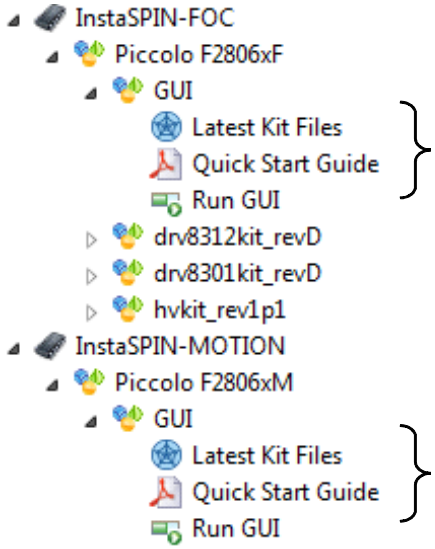
Kits currently available all use the same processor, TMDSCNCD28069MISO Piccolo F28069M (ROM) controlCARD, paired with one of the following 3-phase inverters:

- Low Voltage / Low Current: DRV8312
  - PN: DRV8312-69M-KIT http://www.ti.com/tool/DRV8312-69M-KIT
  - DRV8312 three-phase inverter integrated power module base board supporting up to 50V and 3.5A continuous with controlCARD interface
  - 1 NEMA17 BLDC/PMSM 55W Motor
  - For Position Control, purchase Anaheim Automation motor (with an encoder): BLY172D-24V-4000-2000SI
- Low Voltage / Medium Current: BoosterPack for LaunchPads
  - 6-24V, 14A continuous
  - PN: BOOSTXL-DRV8301 BoosterPack http://www.ti.com/tool/BOOSTXL-DRV8301
  - PN: InstaSPIN enabled LaunchPad: http://www.ti.com/launchpad
  - DRV8301 2.3A sink/1.7A source, three-phase inverter with integrated buck converter for 1.5-A external loads
  - NexFET™ Power MOSFETs
  - No motor or power supply included
- Low Voltage / High Current: DRV8301
  - PN: DRV8301-69M-KIT http://www.ti.com/tool/DRV8301-69M-KIT
  - DRV8301 2.3A sink/1.7A source, three-phase inverter with integrated buck converter for 1.5A external loads
  - No motor included
  - For Position Control, purchase http://www.ti.com/tool/lvservomtr
- High Voltage: hvmtrkit
  - PN: TMDSHVMTRINSPIN http://www.ti.com/tool/TMDSHVMTRINSPIN
  - Support for AC Induction, Permanent Magnet AC Synchronous, Brushless DC Motor
  - Motor driver stage capable of up to 10A@350Vdv-bus continuous
  - High voltage motors are available to order:
    - ACIM http://www.ti.com/tool/hvacimtr
    - BLDC http://www.ti.com/tool/hvbldcmtr
    - PMSM http://www.ti.com/tool/hvpmsmmtr
  - For Position Control, purchase http://www.ti.com/tool/hvpmsmmtr

All software and documentation is available in the MotorWare software download: http://www.ti.com/tool/motorware.

Copyright © 2013–2019, Texas Instruments Incorporated

Kit information is also available from Resource Explorer, an application within Code Composer Studio™ (CCStudio), the IDE for TI MCUs. Resource Explorer will display the first time CCStudio is used with a new workspace, to open it at a later time, select: *Help->Welcome to CCS*. Below are examples of the information that is available.



This document, along with all related documents for using InstaSPIN development kits, is available from Resource Explorer:

Copyright © 2013–2019, Texas Instruments Incorporated

Additional hardware information is available through controlSUITE™ libraries for C2000™ microcontrollers, a cohesive set of software infrastructure and software tools designed to minimize software development time. The controlSUITE libraries can be downloaded from ti.com/tool/controlsuite.

Topic **Page**

2.1 Evaluating InstaSPIN-FOC and InstaSPIN-MOTION..... 53

Copyright © 2013–2019, Texas Instruments Incorporated

2.1 Evaluating InstaSPIN-FOC and InstaSPIN-MOTION

The GUI provides a quick and easy way to evaluate InstaSPIN-FOC and InstaSPIN–MOTION. The InstaSPIN-FOC and InstaSPIN-MOTION Quick Start Guides will lead you through the details of the evaluation process. A simple overview is provided here:

Step 1. Identify the motor parameters (Figure 2-1).

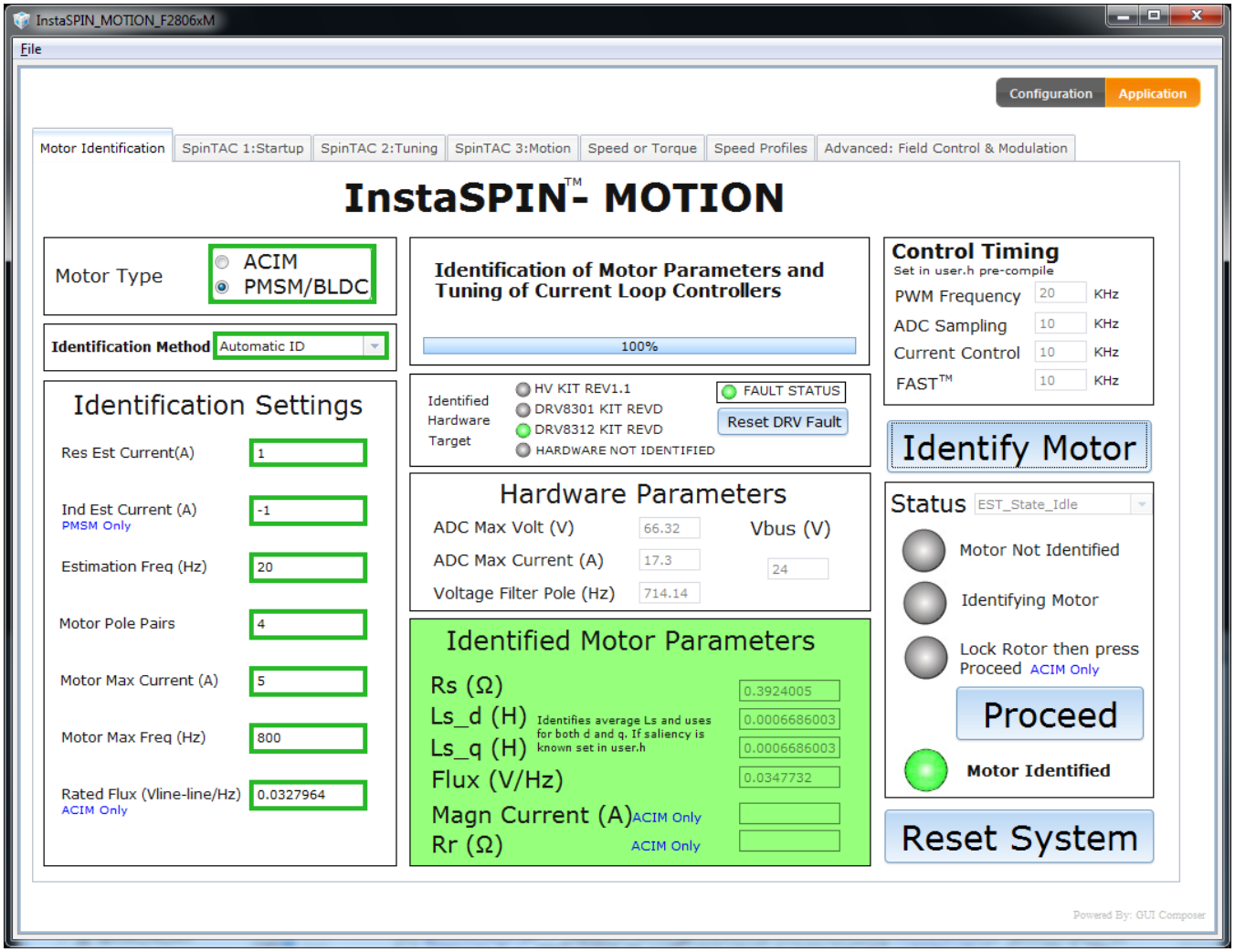


Figure 2-1. InstaSPIN-MOTION GUI Using *Motor Identification* Tab

Step 2. Tune the torque and/speed control (Figure 10-5).

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 2-2. InstaSPIN-MOTION GUI Using *Speed or Torque* Tab

Step 3. Identify system inertia (Figure 2-3).

Copyright © 2013–2019, Texas Instruments Incorporated

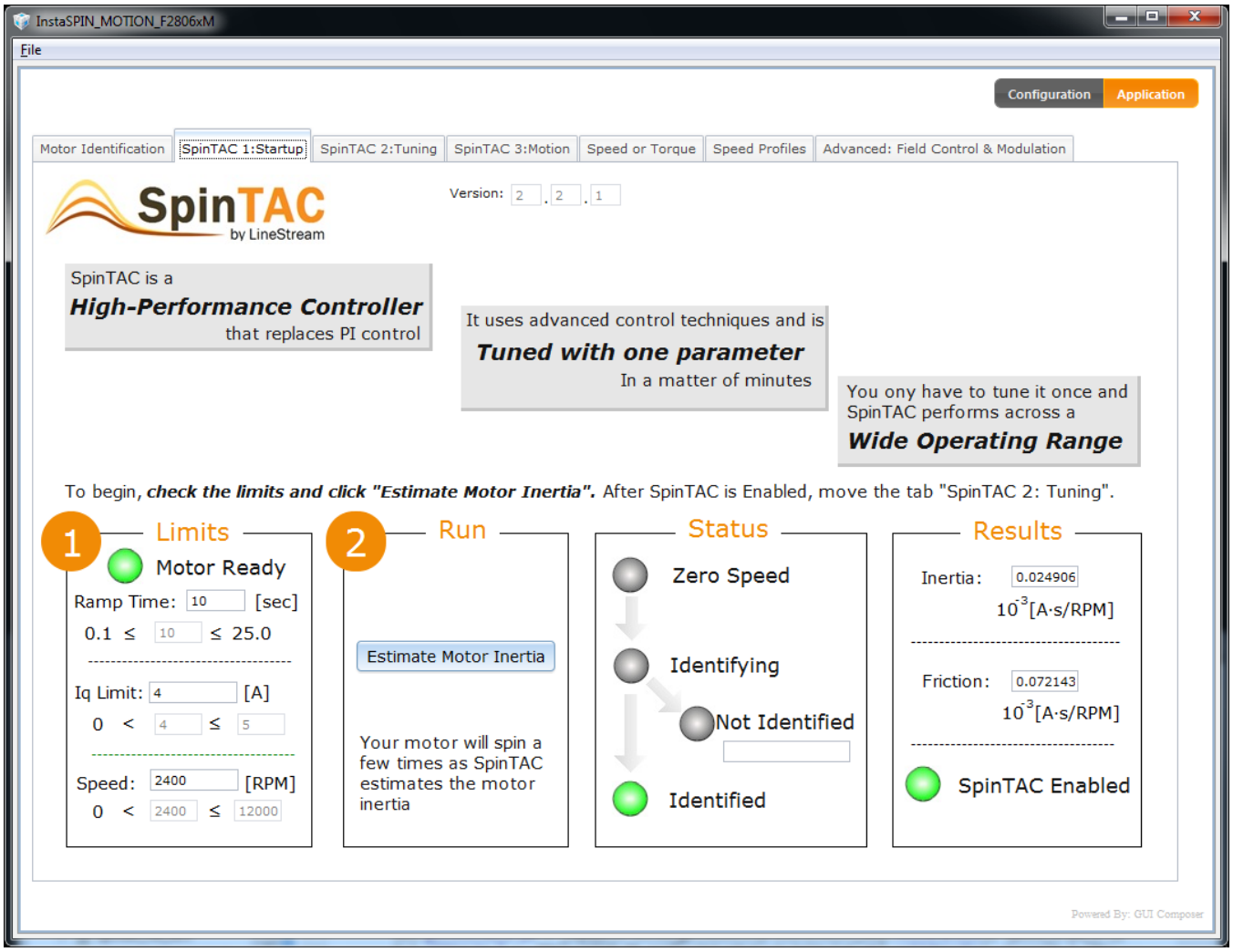


Figure 2-3. InstaSPIN-MOTION GUI Using *SpinTAC 1:Startup* Tab

Step 4. Tune the disturbance-rejecting speed controller (Figure 2-4) This replaces the speed controller in Step 2.

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 2-4. InstaSPIN-MOTION GUI *SpinTAC 2:Tuning* Tab

Step 5. Set the target speed and select the profile type (Figure 2-5).

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 2-5. InstaSPIN-MOTION GUI Using *SpinTAC 3:Motion* Tab

Spending time with the InstaSPIN Quick Start Kits and Quick Start Guides is a good investment of time to become familiar with the software and to reference the hardware schematics for the design of your board. The example code (labs) are configured to run on each of these kits. Whether you are interested in code for InstaSPIN-FOC or InstaSPIN-MOTION, you will find the examples you need to get started fast with your project. For example software and documentation, see the MotorWare *InstaSPIN Projects and Labs User's Guide* in the MotorWare software download (http://www.ti.com/tool/motorware).

Copyright © 2013–2019, Texas Instruments Incorporated

InstaSPIN and MotorWare

The MotorWare™ library is a cohesive set of software and technical resources designed to minimize motor control system development time.

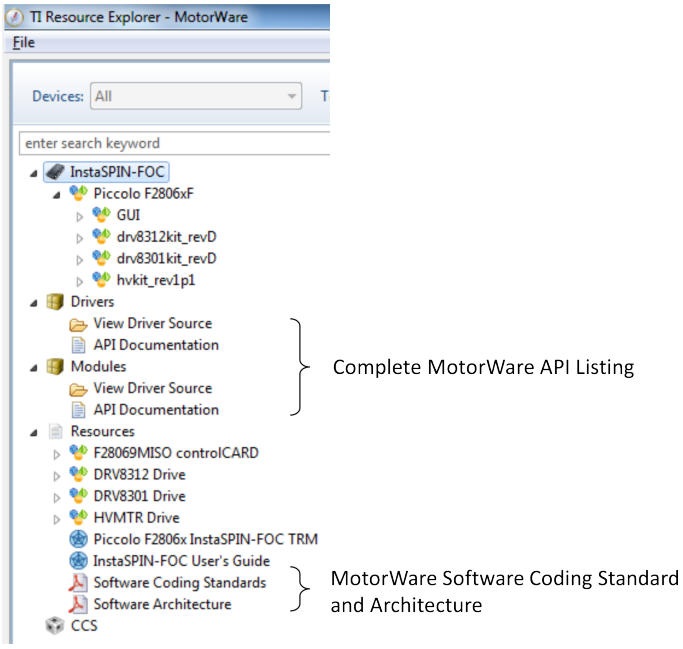
From device-specific drivers and support software to complete system examples and technical training, MotorWare software provides support for every stage of development and evaluation.

MotorWare software has been developed to enable easy integration of best-in-class motor control techniques.

The software has been designed to enable:

- Cross TI MCU support
- Modular and portable across MCU, power electronics and control techniques
- Object Oriented software design
- API based

InstaSPIN-FOC and InstaSPIN-MOTION motor control solutions are delivered within MotorWare. For a latest and complete listing of the API functions, MotorWare's Software Coding Standards and Architecture, see the Resource Explorer found within CCStudio.



Copyright © 2013–2019, Texas Instruments Incorporated

Topic Page

3.1 MotorWare Directory Structure 60
3.2 MotorWare Object-Oriented Design 65
3.3 InstaSPIN-FOC API 68
3.4 InstaSPIN-MOTION and the SpinTAC API 161
3.5 SpinTAC API 169

Copyright © 2013–2019, Texas Instruments Incorporated

3.1 MotorWare Directory Structure

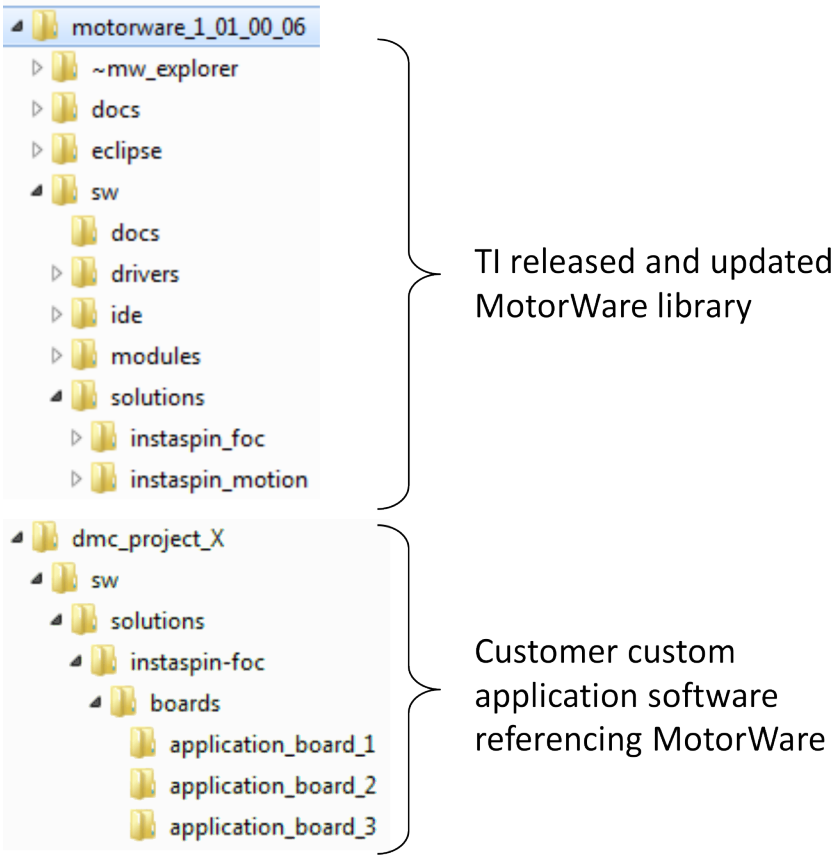
The MotorWare directories are structured to contain all code needed to run every motor control project.

File references in the Code Composer Studio (CCStudio) projects are relative. The relative directory links provides the ability to open a project and compile the first time. The MotorWare directory structure was created to provide an easy way to locate headers, libraries, and source code.

Four folders make up the core of the MotorWare directory structure:

- drivers - Peripheral driver API code.
- ide - Generic linker files used by CCStudio
- modules - Functions used in motor control
- solutions - Contains CCStudio projects to operate software solutions on motor example kits.

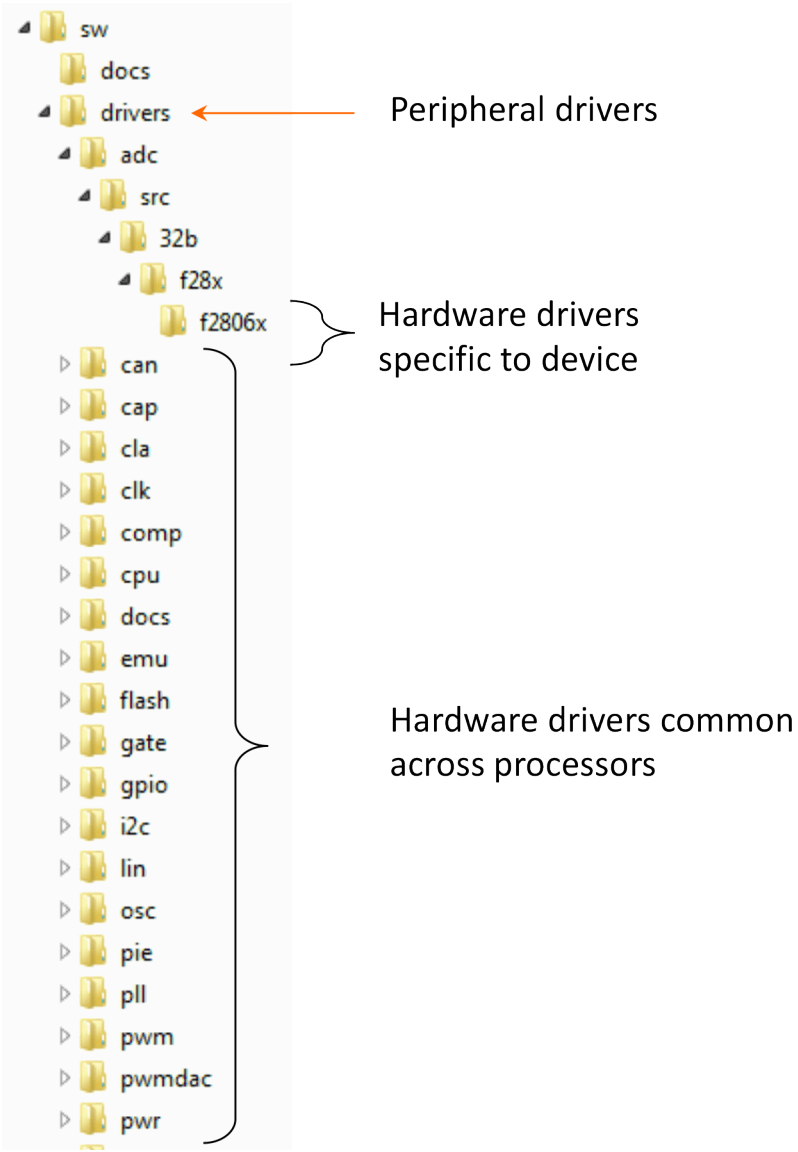
When integrating TI's MotorWare software with your application, it is recommended to create a separate MotorWare directory structure with board specific files for multiple projects. Your software would then reference the files within TI's MotorWare directory. This is optional, but recommended to simplify using future updates that are planned by TI, see the screenshot below. For the actual contents of the directory structure, see the most recent release of MotorWare software.



Copyright © 2013–2019, Texas Instruments Incorporated

3.1.1 MotorWare – drivers

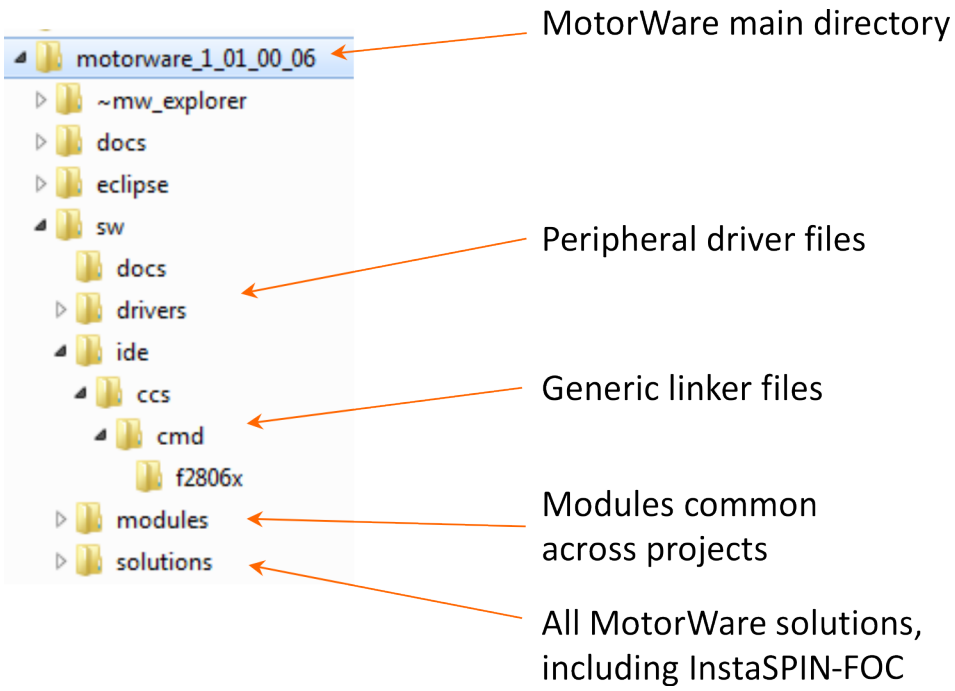
The driver directory contains peripheral APIs for configuring a specific processor.



Copyright © 2013–2019, Texas Instruments Incorporated

3.1.2 MotorWare – ide

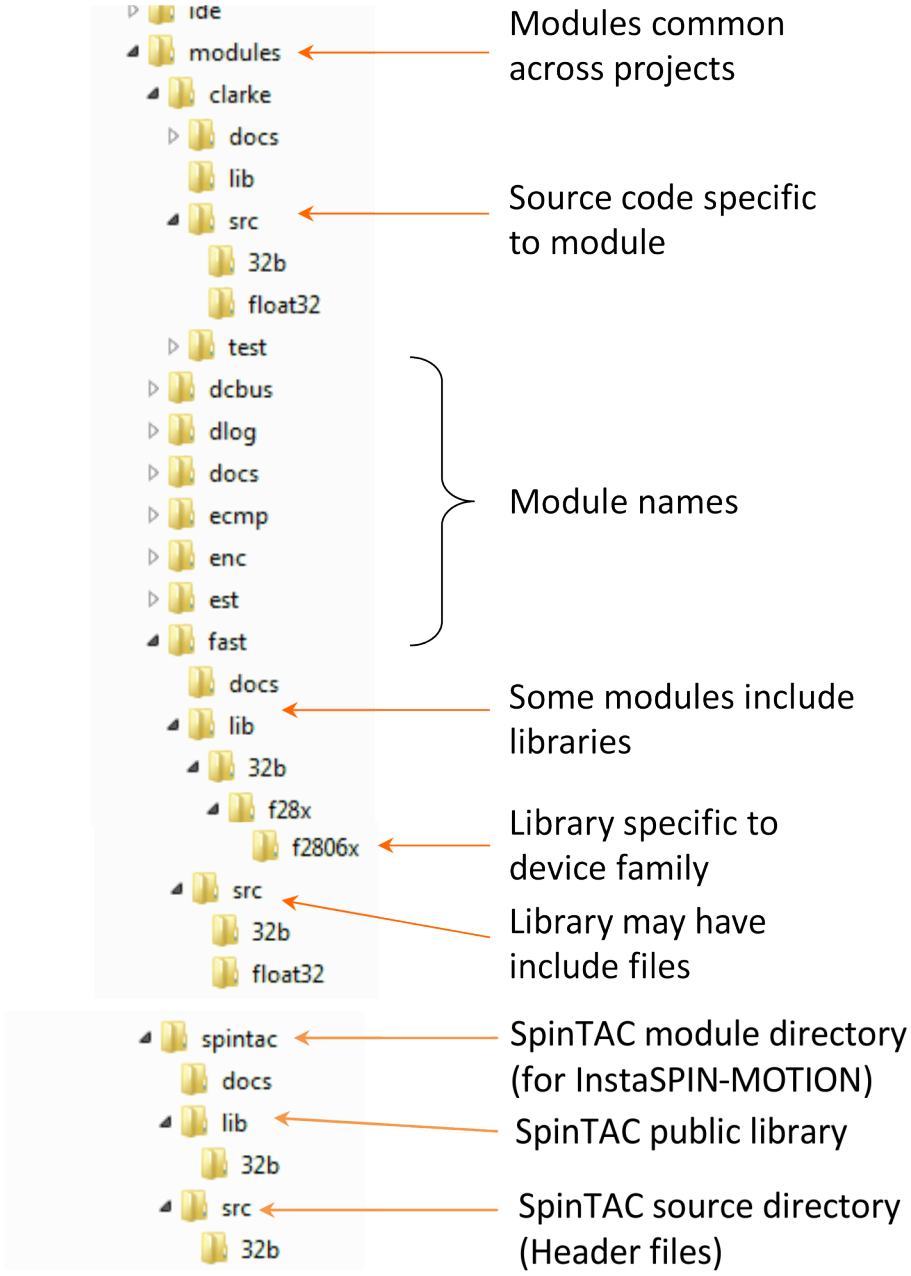
The IDE directory contains generic linker files needed by the compiler tools.



Copyright © 2013–2019, Texas Instruments Incorporated

3.1.3 MotorWare – modules

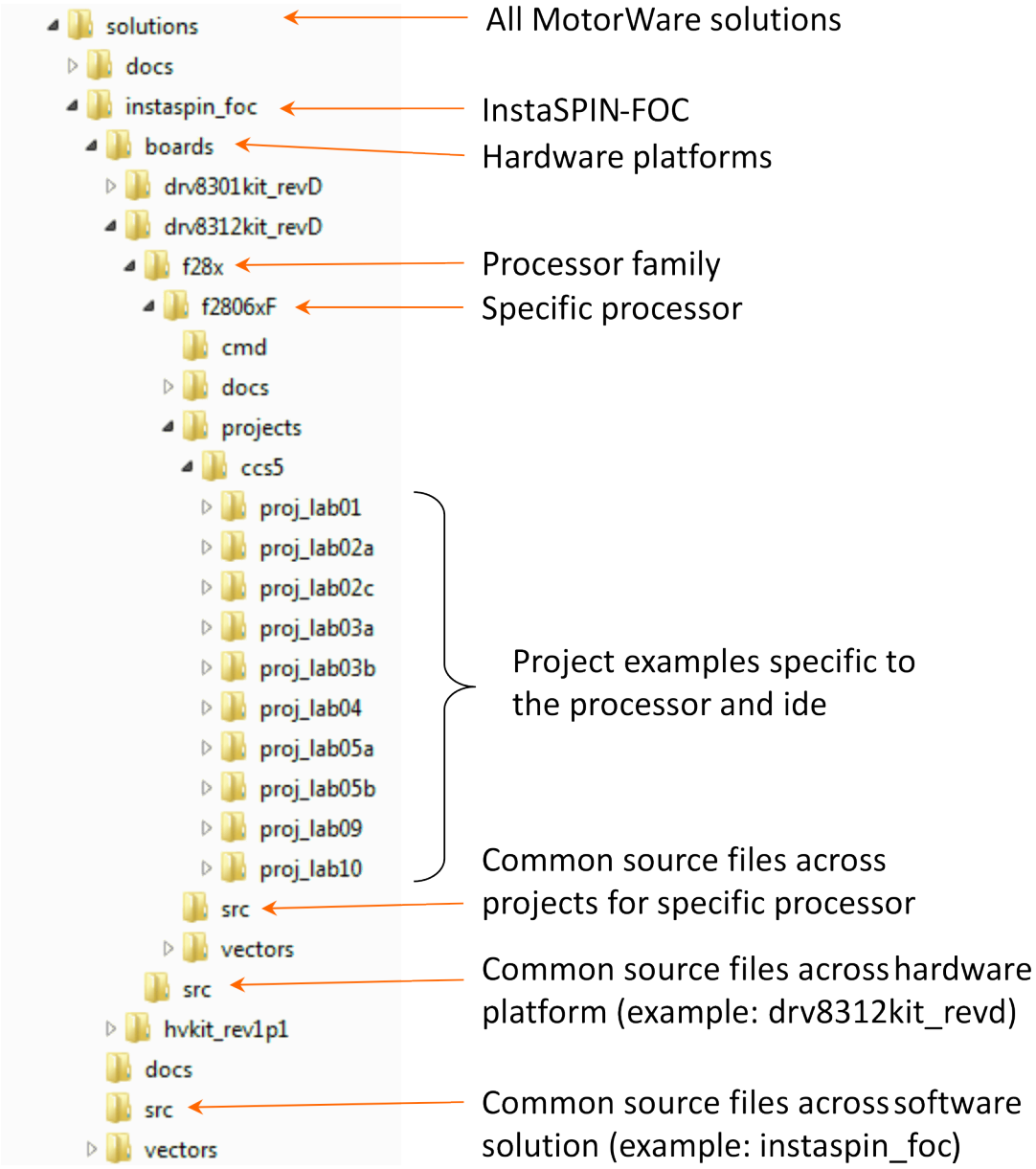
The modules directory contains algorithms that are common across processors and projects. If a module has a specific dependence to a processor it will have sub-directory for this dependence. The module will have source code and/or library functions.



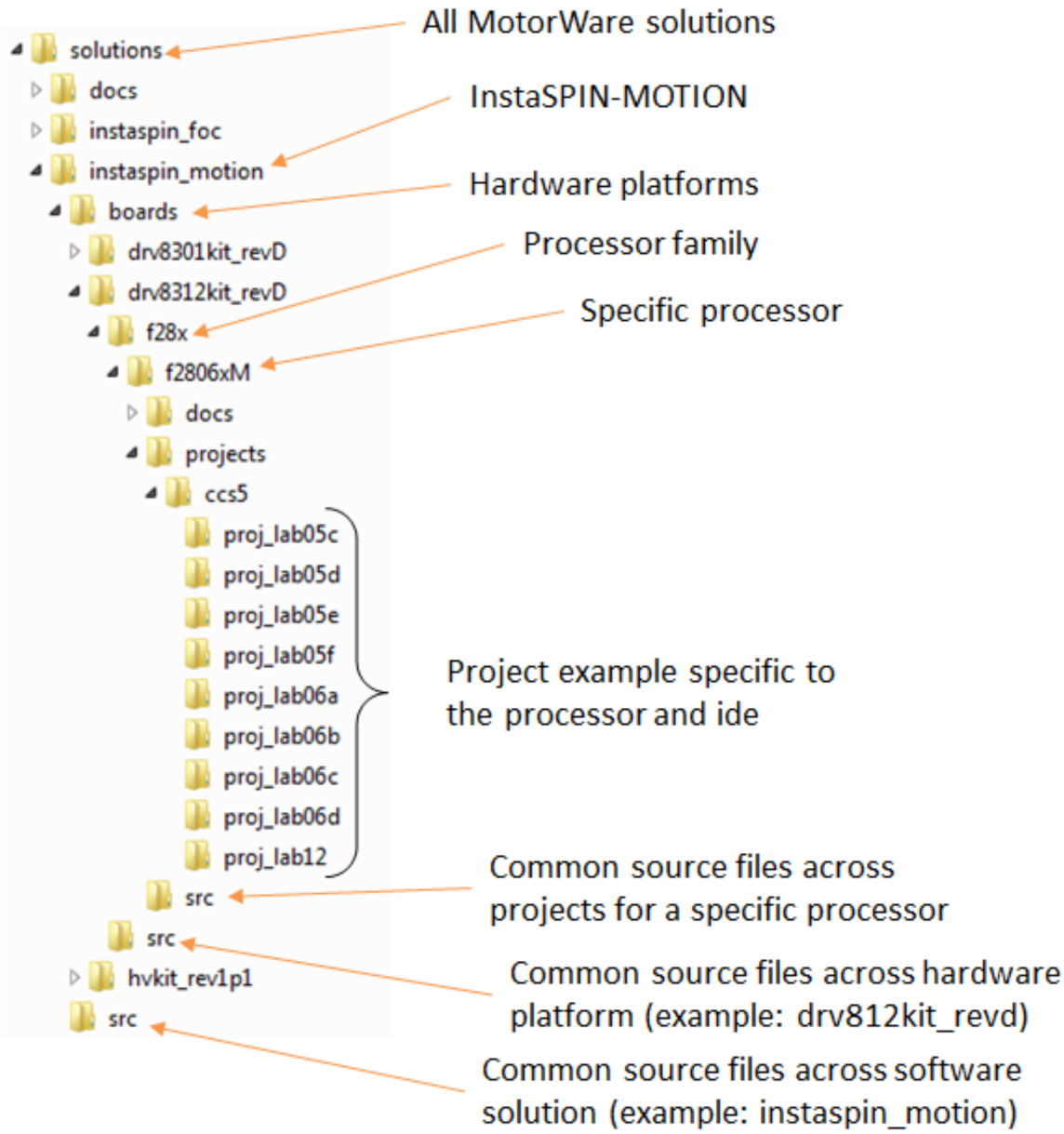
3.1.4 MotorWare – solutions

The solutions directory contains the complete solutions with complete CCStudio projects for specific targets. InstaSPIN-FOC and InstaSPIN-MOTION example code is common across multiple target devices. The hierarchical directory structure allows common source code that is not target dependent to be used with different processors and different target boards. Where there is source code that is target dependent or processor dependent, there are specific source code directories for these purposes. The figure below illustrates this structure.

Copyright © 2013–2019, Texas Instruments Incorporated



Copyright © 2013–2019, Texas Instruments Incorporated



3.2 MotorWare Object-Oriented Design

An object oriented approach has been used for MotorWare software. By using objects, the software is self-documenting and uses much less space in the main.c file. An object is really a structure that contains variables used by the object to perform its function. Associated with an object are methods which are function calls used to setup and run calculations for the object. As we go through the definitions of the object oriented software technique, the park transform object is used to show an example of how the objects are written.

Copyright © 2013–2019, Texas Instruments Incorporated

3.2.1 Objects

The object is a structure. An example of the park transform structure is shown below. Park.h is the file that contains the structure declaration.

```
typedef struct _PARK_Obj_{
 _iq sinTh; //!< the sine of the angle between the d,q and the alpha,beta
            coordinate systems
 _iq cosTh; //!< the cosine of the angle between the d,q and the alpha,beta
            coordinate systems
 } PARK_Obj;
```

Every object has a handle. The handle is a pointer to the object. A handle is very useful when passing objects between functions. Handles to objects also allow functions to work on only that object, or what is called re-entrant code. The handle declaration for the Park transform object is listed below.

```
typedef struct PARK_Obj *PARK_Handle;
```

3.2.2 Methods

Every object must do something. As it stands, an object is just a container of variables. For the object to perform calculations or even send and receive data between its variables, it must have methods. Methods are functions specific to an object that work on the variables the object contains. There are four main methods to every object in MotorWare and they are named as follows.

- Init method - Used only to create a handle to an object
- Set method - Sets internal variables of an object
- Get method - Returns internal variable values of an object
- Run method - Performs the calculation function of the object

3.2.2.1 Init method

The **init** method is only used to point a handle to an object. Code for the Park transform **init** method is shown next.

```
PARK_Handle PARK_init(void *pMemory,const size_t numBytes)
{
 PARK_Handle parkHandle;
 if(numBytes < sizeof(PARK_Obj))
  return((PARK_Handle)NULL);

 // assign the handle
 parkHandle = (PARK_Handle)pMemory;

 return(parkHandle);
}
```

The **init** method only takes two parameters, first the address of the object and second is the size (in bytes) of the object. After the object is created, the other methods are used.

Copyright © 2013–2019, Texas Instruments Incorporated

3.2.2.2 Set method

A **set** method puts a value into the variables that the object contains. In the example code for the Park transform below, the **set** function assigns sine and cosine values to sinTh and cosTh object variables.

```
static inline void PARK_setup(PARK_Handle parkHandle,const _iq angle_pu)
{
 PARK_Obj *park = (PARK_Obj *)parkHandle;

 park->sinTh = _IQsinPU(angle_pu);
 park->cosTh = _IQcosPU(angle_pu);

 return;
}
```

The set method takes as parameters the object handle and in the Park transform example, the angle θ. Set functions do not return any values.

3.2.2.3 Get method

Get methods return object variables. Only two variables are contained in the Park object. Because the two variables contained in Park are needed outside of the object, there are two **get** methods. One of the **get** methods is shown below.

```
static inline _iq PARK_getSinTh(PARK_Handle parkHandle)
{
 PARK_Obj *park = (PARK_Obj *)parkHandle;

 return(park->sinTh);
}
```

The **get** method returns only the variable that the method is named for. In the example code above for the Park object **get** method, the sinTh variable is returned. A handle to the object is the only variable that is passed to the **get** method. Only one variable is returned by a **get** method.

3.2.2.4 Run method

Run methods perform calculations of the object variables. In the case of embedded software, **run** methods might also operate a peripheral or some other hardware. The Park **run** method calculates the Park transform of the input vector {I_d, I_q } and then returns the output vector {I_α, I_β}. The code for the Park **run** method is shown below.

```
static inline void PARK_run(PARK_Handle parkHandle,const MATH_vec2
                *pInVec,MATH_vec2 *pOutVec)
{
 PARK_Obj *park = (PARK_Obj *)parkHandle;
 _iq sinTh = park->sinTh;
 _iq cosTh = park->cosTh;
 _iq value_0 = pInVec->value[0];
 _iq value_1 = pInVec->value[1];
 pOutVec->value[0] = _IQmpy(value_0,cosTh) + _IQmpy(value_1,sinTh);
 pOutVec->value[1] = _IQmpy(value_1,cosTh) - _IQmpy(value_0,sinTh);
 return;
} // end of PARK_run() function
```

In **run** methods the first parameter is the handle to the object, the subsequent parameters are input and output variables when in single quantities or pointers to vectors. Nothing is returned from a **run** method.

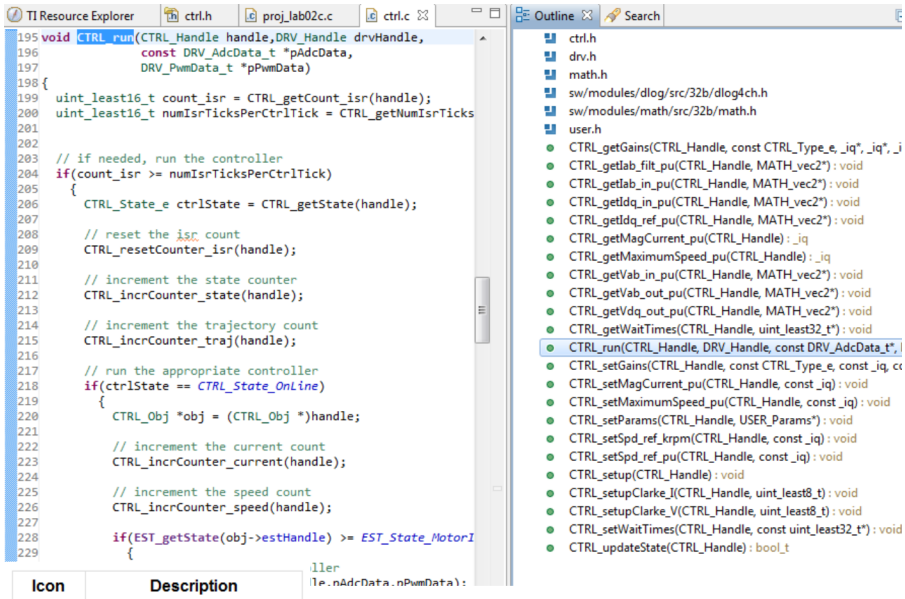
Copyright © 2013–2019, Texas Instruments Incorporated

3.3 InstaSPIN-FOC API

All of the functionality of InstaSPIN-FOC is accessible through an extensive API. This API remains the same whether InstaSPIN-FOC is in ROM or user memory. In this section, we will review the most commonly used functions that provide access to variables and enable your application to implement system control. These functions are used in the lab example projects at the end of this guide. For the latest and complete listing of the API functions, see the Resource Explorer found within CCStudio.

Another resource for the API functions that is especially useful during software development is the Outline View within CCStudio. This is feature provides navigation across the multiple files with a complete hyperlink listing of all the symbols within the file you are using that is part of a CCStudio project. Access this view from the CCStudio menu: *Window->Show View->Outline*.

Below is a screen capture using this view with ctrl.c. Notice that by clicking on the function name in the Outline window, the cursor in the Source window highlights the related code. You can quickly navigate functions and all symbols within a file, which is especially useful for the large number of API functions in InstaSPIN-FOC.



Icon	Description
	Class
	Namespace
	Macro Definition
	Enum
	Enumerator
	Variable
	Field private
	Field protected
	Field public
	Include
	Method private
	Method protected
	Method public
	Struct
	Type definition
	Union
	Function

When using Outline View with ctrl.h you will notice some functions do not have source code listed. These functions will have a circle with a green and white pattern next to the filename (instead of a solid green circle). This indicates it is one of the few files that must remain in ROM since it is a function that interfaces directly to the FAST estimator. An example of this can be seen in the figure below where the function ***CTRL_initCtrl()*** does not have source code but the function ***CTRL_isError()*** does.

Copyright © 2013–2019, Texas Instruments Incorporated

```
1133 //! \brief      Initializes a specified controller
1134 //! \details    Initializes all handles required for field
1135 //!             interface.  Returns a handle to the CTRL ob
1136 //! \param[in]  ctrlNumber  The controller number
1137 //! \param[in]  estNumber   The estimator number
1138 //! \return     The controller (CTRL) object handle
1139 CTRL_Handle CTRL_initCtrl(const uint_least8_t ctrlNumber, c
1140
1141
1142 //! \brief     Determines if there is a controller error
1143 //! \param[in] handle  The controller (CTRL) handle
1144 //! \return    A boolean value denoting if there is a contr
1145 inline bool_t CTRL_isError(CTRL_Handle handle)
1146 {
1147   CTRL_State_e ctrlState = CTRL_getState(handle);
1148   bool_t state = FALSE;
```

```
CTRL_getWaitTimes(CTRL_Handle, uint_least32_t*) : void
CTRL_incrCounter_current(CTRL_Handle) : void
CTRL_incrCounter_isr(CTRL_Handle) : void
CTRL_incrCounter_speed(CTRL_Handle) : void
CTRL_incrCounter_state(CTRL_Handle) : void
CTRL_incrCounter_traj(CTRL_Handle) : void
CTRL_initCtrl(const uint_least8_t, const uint_least8_t) : CTRL_Handle
CTRL_isError(CTRL_Handle) : bool_t
CTRL_resetCounter_current(CTRL_Handle) : void
CTRL_resetCounter_isr(CTRL_Handle) : void
CTRL_resetCounter_speed(CTRL_Handle) : void
CTRL_resetCounter_state(CTRL_Handle) : void
CTRL_resetCounter_traj(CTRL_Handle) : void
```

The block diagram shown in Figure 3-1 provides a good overview of the InstaSPIN-FOC system's functions and variables as they relate to user memory and ROM. Notice the key functions: CTRL_run, CTRL_setup, EST_run, HAL_run, HAL_acqAdcInt, and HAL_readAdcData. Also, the variables show in the diagram are all available. For example, the Ki gain for PI used for Id and Iq can be read with the function CTRL_getKi and set with CTRL_setKi. The intention is to give full access to all functions and variables.

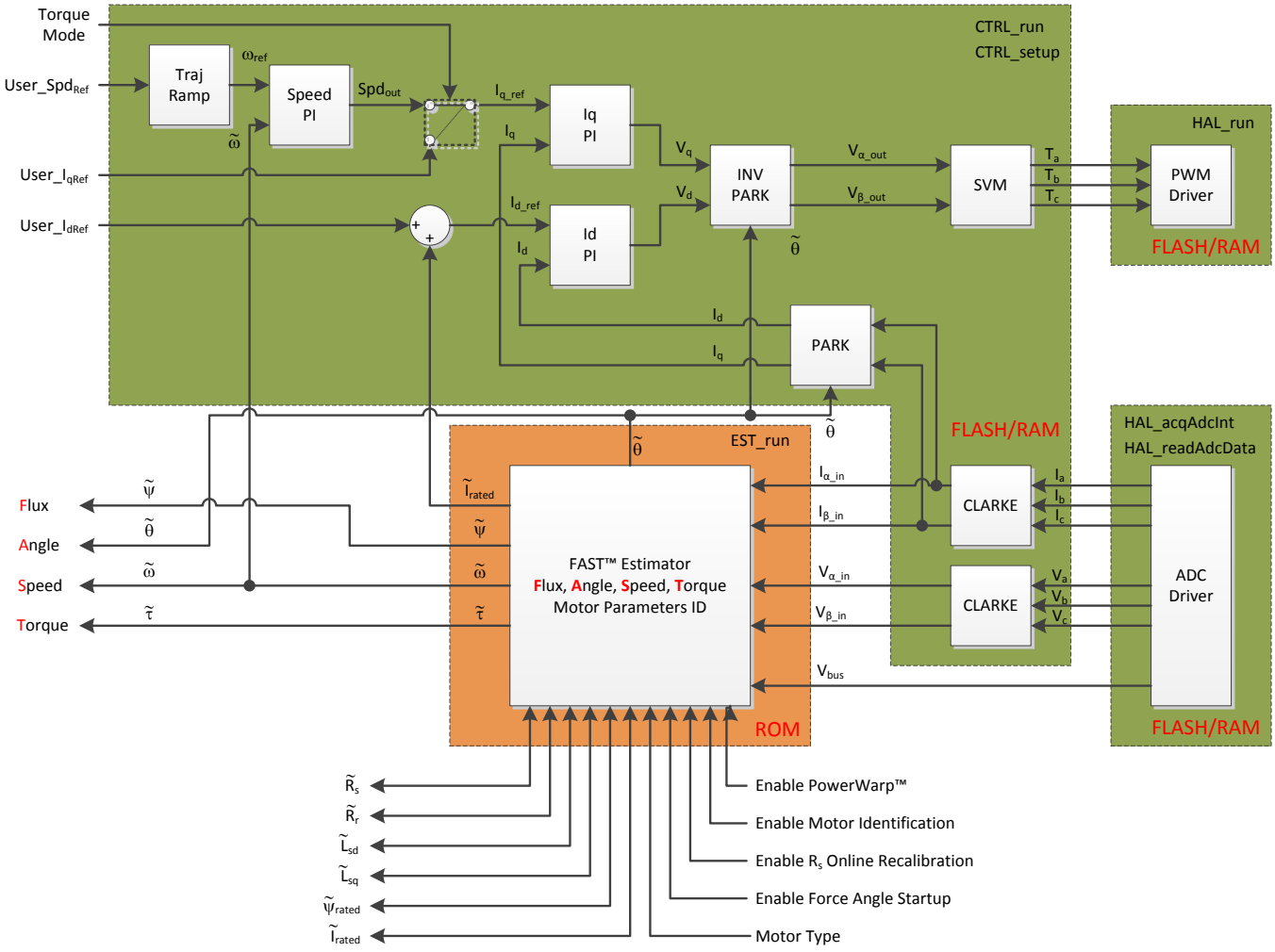


Figure 3-1. Block Diagram of InstaSPIN-FOC in User Memory, with Exception of FAST in ROM

The API can be grouped into the following four categories:

- Controller – ctrl.c (software that can be moved to user memory)
- Estimator – FAST library (FAST estimator in ROM)
- HAL – hal.c (Hardware Abstraction Layer)

Copyright © 2013–2019, Texas Instruments Incorporated

- User – user.c (user settings)

The commonly used functions from each of these groups are listed in the next sections.

For the F2802xF devices, the API has an additional library in user memory:

- Public Library – fast_public.lib (software that has to be loaded in user memory)

The commonly used functions from each of these groups are listed in the next sections.

3.3.1 Controller API Functions – ctrl.c, ctrl.h, CTRL_obj.h

3.3.1.1 CTRL Enumerations and Structures

CTRL_Obj

Defines the controller (CTRL) object. The controller object implements all of the FOC algorithms and the estimator functions.

```
typedef struct _CTRL_Obj_
{
  CTRL_Version       version;             //!< the controller version
  CTRL_State_e       state;               //!< the current state of the controller
  CTRL_State_e       prevState;           //!< the previous state of the
controller
  CTRL_ErrorCode_e   errorCode;           //!< the error code for the controller
  CLARKE_Handle      clarkeHandle_I; //!< the handle for the current Clarke
transform
  CLARKE_Obj         clarke_I;            //!< the current Clarke transform object
  CLARKE_Handle      clarkeHandle_V; //!< the handle for the voltage Clarke
transform
  CLARKE_Obj         clarke_V;            //!< the voltage Clarke transform object
  EST_Handle         estHandle;           //!< the handle for the parameter
estimator
  PARK_Handle        parkHandle;          //!< the handle for the Park object
  PARK_Obj           park;                //!< the Park transform object
  PID_Handle         pidHandle_Id;        //!< the handle for the Id PID
controller
  PID_Obj            pid_Id;              //!< the Id PID controller object
  PID_Handle         pidHandle_Iq;        //!< the handle for the Iq PID
controller
  PID_Obj            pid_Iq;              //!< the Iq PID controller object
  PID_Handle         pidHandle_spd;   //!< the handle for the speed PID controller
  PID_Obj            pid_spd;             //!< the speed PID controller object
  IPARK_Handle       iparkHandle;     //!< the handle for the inverse Park
transform
  IPARK_Obj          ipark;               //!< the inverse Park transform object
  SVGEN_Handle       svgenHandle;     //!< the handle for the space vector
generator
  SVGEN_Obj          svgen;               //!< the space vector generator object
  TRAJ_Handle        trajHandle_Id;   //!< the handle for the Id trajectory
generator
  TRAJ_Obj           traj_Id;             //!< the Id trajectory generator object
  TRAJ_Handle        trajHandle_spd;  //!< handle for the speed trajectory
generator
  TRAJ_Obj           traj_spd;            //!< the speed trajectory generator
object
  TRAJ_Handle        trajHandle_spdMax;  //!< handle for max speed traj generator
  TRAJ_Obj           traj_spdMax;     //!< the max speed trajectory generator
object
  MOTOR_Params       motorParams;         //!< the motor parameters
  uint_least32_t     waitTimes[CTRL_numStates];
//!< an array of wait times for each state, estimator clock counts
  uint_least32_t     counter_state;           //!< the state counter
  uint_least16_t     numIsrTicksPerCtrlTick;  //!< # of isr ticks per controller
tick
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
  uint_least16_t     numCtrlTicksPerCurrentTick;
//!< # of controller ticks per current controller tick
  uint_least16_t     numCtrlTicksPerSpeedTick;
//!< # of controller ticks per speed controller tick
  uint_least16_t     numCtrlTicksPerTrajTick;
//!< # of controller ticks per trajectory tick
  uint_least32_t     ctrlFreq_Hz;         //!< Defines the controller frequency,
Hz
  uint_least32_t     trajFreq_Hz;         //!< Defines the trajectory frequency,
Hz
  _iq                trajPeriod_sec;      //!< Defines the trajectory period, sec
  float_t            ctrlPeriod_sec;      //!< Defines the controller period, sec
  _iq                maxVsMag_pu; //!< the maximum voltage vector that is
allowed, pu
  MATH_vec2          Iab_in;              //!< the Iab input values
  MATH_vec2          Iab_filt;            //!< the Iab filtered values
  MATH_vec2          Idq_in;              //!< the Idq input values
  MATH_vec2          Vab_in;              //!< the Vab input values
  _iq                spd_out;             //!< the speed output value
  MATH_vec2          Vab_out;             //!< the Vab output values
  MATH_vec2          Vdq_out;             //!< the Vdq output values
  float_t            Rhf;                 //!< the Rhf value
  float_t            Lhf;                 //!< the Lhf value
  float_t            RoverL;              //!< the R/L value
  _iq                Kp_Id;               //!< the desired Kp_Id value
  _iq                Kp_Iq;               //!< the desired Kp_Iq value
  _iq                Kp_spd;              //!< the desired Kp_spd value
  _iq                Ki_Id;               //!< the desired Ki_Id value
  _iq                Ki_Iq;               //!< the desired Ki_Iq value
  _iq                Ki_spd;              //!< the desired Ki_spd value
  _iq                Kd_Id;               //!< the desired Kd_Id value
  _iq                Kd_Iq;               //!< the desired Kd_Iq value
  _iq                Kd_spd;              //!< the desired Kd_spd value
  _iq                Ui_Id;               //!< the desired Ui_Id value
  _iq                Ui_Iq;               //!< the desired Ui_Iq value
  _iq                Ui_spd;              //!< the desired Ui_spd value
  MATH_vec2          Idq_ref;             //!< the Idq reference values, pu
  _iq                IdRated;             //!< the Id rated current, pu
  _iq                spd_ref;             //!< the speed reference, pu
  _iq                spd_max;             //!< the maximum speed, pu
  uint_least16_t     counter_current;     //!< the isr counter
  uint_least16_t     counter_isr;         //!< the isr counter
  uint_least16_t     counter_speed;       //!< the speed counter
  uint_least16_t     counter_traj;        //!< the traj counter
  bool             flag_enableCtrl;       //!< a flag to enable the controller
  bool             flag_enableDcBusComp;
//!< a flag to enable the DC bus compensation in the controller
  bool             flag_enablePowerWarp;  //!< a flag to enable PowerWarp
  bool             flag_enableOffset;
//!< a flag to enable offset estimation after idle state
  bool             flag_enableSpeedCtrl;  //!< a flag to enable the speed
controller
  bool             flag_enableUserMotorParams;
//!< a flag to use known motor parameters from user.h file
} CTRL_Obj;
```


Copyright © 2013–2019, Texas Instruments Incorporated

CTRL_State_e

Enumeration for the controller states

```
typedef enum {
  CTRL_State_Error=0,          //!< the controller error state
  CTRL_State_Idle,             //!< the controller idle state
  CTRL_State_OffLine,          //!< the controller offline state
  CTRL_State_OnLine,           //!< the controller online state
  CTRL_numStates               //!< the number of controller states
} CTRL_State_e;
```

CTRL_ErrorCode_e

Enumeration for the error codes

```
typedef enum
{
  CTRL_ErrorCode_NoError=0,        //!< no error error code
  CTRL_ErrorCode_IdClip,           //!< Id clip error code
  CTRL_ErrorCode_EstError,         //!< estimator error code
  CTRL_numErrorCodes               //!< the number of controller error codes
} CTRL_ErrorCode_e;
```

CTRL_TargetProc_e

Enumeration for the target processors

```
typedef enum
{
  CTRL_TargetProc_2806x=0,   //!< 2806x processor
  CTRL_TargetProc_2805x,     //!< 2805x processor
  CTRL_TargetProc_2803x,     //!< 2803x processor
  CTRL_TargetProc_2802x,     //!< 2802x processor
  CTRL_TargetProc_Unknown    //!< Unknown processor
} CTRL_TargetProc_e;
```

CTRL_Type_e

Enumeration for the target processors

```
typedef enum
{
  CTRL_Type_PID_spd=0,             //!< PID Speed controller
  CTRL_Type_PID_Id,                //!< PID Id controller
  CTRL_Type_PID_Iq                 //!< PID Iq controller
} CTRL_Type_e;
```

CTRL_Version

Defines the controller (CTRL) version number

```
typedef struct _CTRL_Version_
{
  uint16_t rsvd;                 //!< reserved value
  CTRL_TargetProc_e targetProc;  //!< the target processor
  uint16_t major;                //!< the major release number
  uint16_t minor;                //!< the minor release number
} CTRL_Version;
```


Copyright © 2013–2019, Texas Instruments Incorporated

3.3.1.2 CTRL State Control and Error Handling

CTRL_initCtrl() ***ctrlHandle = CTRL_initCtrl(ctrlNumber,estNumber)***

Initializes a specified controller

ctrlNumber: The FOC (CTRL) controller number – The number of the FOC controller.

estNumber: The estimator (EST) number – The number of the InstaSPIN estimator.

Return: The controller (CTRL) object handle – The handle that is returned to point to the specific estimator and controller.

CTRL_updateState () ***bool CTRL_updateState(CTRL_Handle handle)***

Feeds back whether or not the controller state has changed

Handle: The controller (CTRL) handle

Return: A Boolean value denoting if the state has changed (true) or not (false)

CTRL_isError () ***inline bool CTRL_isError(CTRL_Handle handle)***

Determines if there is a controller error

Handle: The controller (CTRL) handle

Return: A Boolean value denoting if there is a controller error (true) or not (false)

CTRL_checkForErrors ()
inline void CTRL_checkForErrors(CTRL_Handle handle)

Checks for error with the estimator and, if found, sets the controller state to the error state

Handle: The controller (CTRL) handle

Errors: CTRL_State_Error, CTRL_ErrorCode_EstError

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.1.3 CTRL Get Functions

CTRL_getCount_current ()

inline uint_least16_t CTRL_getCount_current(CTRL_Handle handle)

Gets the current loop count

Handle: The controller (CTRL) handle

Return: The current loop count, obj->counter_current

CTRL_getCount_isr ()

inline uint_least16_t CTRL_getCount_isr(CTRL_Handle handle)

Gets the isr count

Handle: The controller (CTRL) handle

Return: The isr count, obj->counter_isr

CTRL_getCount_speed ()

inline uint_least16_t CTRL_getCount_speed(CTRL_Handle handle)

Gets the speed loop count

Handle: The controller (CTRL) handle

Return: The speed loop count, obj->counter_speed

CTRL_getCount_state ()

inline uint_least32_t CTRL_getCount_state(CTRL_Handle handle)

Gets the state count

Handle: The controller (CTRL) handle

Return: The state count, obj->counter_state

CTRL_getCount_traj ()

inline uint_least32_t CTRL_getCount_state(CTRL_Handle handle)

Gets the trajectory loop count

Handle: The controller (CTRL) handle

Return: The trajectory loop count, obj->counter_traj

CTRL_getCtrlFreq () *inline uint_least32_t CTRL_ getCtrlFreq(CTRL_Handle handle)*

Gets the controller execution frequency

Handle: The controller (CTRL) handle

Return: The controller execution frequency, Hz, obj->ctrlFreq_Hz

CTRL_getCtrlPeriod_sec ()

inline float_t CTRL_getCtrlPeriod_sec(CTRL_Handle handle)

Copyright © 2013–2019, Texas Instruments Incorporated

Gets the controller execution period

Handle: The controller (CTRL) handle

Return: The controller execution period, sec, obj->ctrlPeriod_sec

CTRL_getErrorCode ()

inline CTRL_ErrorCode_e CTRL_getErrorCode(CTRL_Handle handle)

Gets the error code from the controller (CTRL) object

Handle: The controller (CTRL) handle

Return: The error code, obj->errorCode

CTRL_getEstHandle ()

inline EST_Handle CTRL_getEstHandle(CTRL_Handle handle)

Gets the estimator handle for a given controller

Handle: The controller (CTRL) handle

Return: The estimator handle for the given controller, obj->estHandle

CTRL_getFlag_enableCtrl ()

inline bool CTRL_getFlag_enableCtrl(CTRL_Handle handle)

Gets the enable controller flag value from the estimator

Handle: The controller (CTRL) handle

Return: The enable controller flag value, obj->flag_enableCtrl

CTRL_getFlag_enableDcBusComp ()

inline bool CTRL_getFlag_enableDcBusComp(CTRL_Handle handle)

Gets the enable DC bus compensation flag value from the estimator

Handle: The controller (CTRL) handle

Return: The enable DC bus compensation flag value, obj-> flag_enableDcBusComp

CTRL_getFlag_enablePowerWarp ()

inline bool CTRL_getFlag_enablePowerWarp(CTRL_Handle handle)

Gets the PowerWarp enable flag value from the estimator

Handle: The controller (CTRL) handle

Return: The PowerWarp enable flag value, obj-> flag_enablePowerWarp

CTR CTRL_getFlag_enableOffset ()

inline bool CTRL_getFlag_enableOffset(CTRL_Handle handle)

Gets the enable offset flag value from the controller

Handle: The controller (CTRL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

Return: The enable offset flag value, obj-> flag_enableOffset

CTRL_getFlag_enableSpeedCtrl ()

inline bool CTRL_getFlag_enableSpeedCtrl(CTRL_Handle handle)

Gets the enable speed control flag value from the controller

Handle: The controller (CTRL) handle

Return: The enable speed control flag value, obj-> flag_enableSpeedCtrl

CTRL_getFlag_enableUserMotorParams ()

inline bool CTRL_getFlag_enableSpeedCtrl(CTRL_Handle handle)

Gets the enable user motor parameters flag value in the estimator

Handle: The controller (CTRL) handle

Return: The enable user motor parameters flag value:

- true = Use the user motor parameters from user.h
- false = Perform motor parameter estimation

CTRL_getGains ()

void CTRL_getGains(CTRL_Handle handle, const CTRL_Type_e ctrlType, iq *pKp,_iq *pKi,_iq *pKd)

Updates Kp, Ki, and Kd in the controller object

Handle: The controller (CTRL) handle

ctrlType: The controller type

pKp: The pointer for the Kp value, pu

pKi: The pointer for the Ki value, pu

pKd: The pointer for the Kd value, pu

CTRL_getlab_filt_pu ()

void CTRL_getlab_filt_pu (CTRL_Handle handle,MATH_vec2 *plab_filt_pu)

Updates the alpha/beta filtered current vector values in the controller object

Handle: The controller (CTRL) handle

plab_filt_pu: The vector for the alpha/beta filtered current vector values, pu

CTRL_getlab_filt_addr ()

inline MATH_vec2 *CTRL_getlab_filt_addr(CTRL_Handle handle)

Gets the alpha/beta filtered current vector memory address from the controller

Handle: The controller (CTRL) handle

Return: The alpha/beta filtered current vector memory address, obj->Iab_filt

CTRL_getlab_in_addr ()

inline MATH_vec2 *CTRL_getlab_in_addr(CTRL_Handle handle)

Copyright © 2013–2019, Texas Instruments Incorporated

Gets the alpha/beta current input vector memory address from the controller

Handle: The controller (CTRL) handle

Return: The alpha/beta current input vector memory address, obj-> Iab_in

CTRL_getIab_in_pu ()

void CTRL_getIab_in_pu(CTRL_Handle handle,MATH_vec2 *pIab_in_pu);

Gets the alpha/beta current input vector values from the controller

Handle: The controller (CTRL) handle

pIab_in_pu: The vector for the alpha/beta current input vector values, pu

CTRL_getId_in_pu ()

inline _iq CTRL_getId_in_pu(CTRL_Handle handle)

Gets the direct current input value from the controller

Handle: The controller (CTRL) handle

Return: The direct current input value, pu, obj->Idq_in.value[0]

CTRL_getId_ref_pu ()

inline _iq CTRL_getId_ref_pu(CTRL_Handle handle)

Gets the direct current (Id) reference value from the controller

Handle: The controller (CTRL) handle

Return: The direct current reference value, pu, obj-> Idq_ref.value[0]

CTRL_getIdq_in_addr ()

inline MATH_vec2 *CTRL_getIdq_in_addr(CTRL_Handle handle)

Gets the direct/quadrature current input vector memory address from the controller

Handle: The controller (CTRL) handle

Return: The direct/quadrature current input vector memory address, obj-> Idq_in

CTRL_getIdq_in_pu ()

void CTRL_getIdq_in_pu(CTRL_Handle handle,MATH_vec2 *pIdq_in_pu);

Gets the direct/quadrature current input vector values from the controller

Handle: The controller (CTRL) handle

pIdq_in_pu: The vector for the direct/quadrature input current vector values, pu

CTRL_getIdq_ref_pu ()

void CTRL_getIdq_ref_pu(CTRL_Handle handle,MATH_vec2 *pIdq_ref_pu);

Gets the direct/quadrature current reference vector values from the controller

Handle: The controller (CTRL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

pIdq_ref_pu: The vector for the direct/quadrature current reference vector values, pu

CTRL_getIdRated_pu ()

inline _iq CTRL_getIdRated_pu(CTRL_Handle handle)

Gets the Id rated current value from the controller

Handle: The controller (CTRL) handle

Return: The Id rated current value, pu, obj-> IdRated

CTRL_getIq_in_pu ()

inline _iq CTRL_getIq_in_pu(CTRL_Handle handle)

Gets the quadrature current input value from the controller

Handle: The controller (CTRL) handle

Return: The quadrature current input value, pu, obj-> Idq_in.value[1]

CTRL_getIq_ref_pu ()

inline _iq CTRL_getIq_ref_pu(CTRL_Handle handle)

Gets the quadrature current (Iq) reference value from the controller

Handle: The controller (CTRL) handle

Return: The quadrature current reference value, pu, obj-> Idq_ref.value [1]

CTRL_getKi ()

_iq CTRL_getKi (CTRL_Handle handle,const CTRL_Type_e ctrlType)

Gets the controller state

Handle: The controller (CTRL) handle

ctrlType: The controller type

Return: The Ki value

CTRL_getKd ()

_iq CTRL_getKd (CTRL_Handle handle,const CTRL_Type_e ctrlType)

Gets the proportional gain (Kd) value from the specified controller

Handle: The controller (CTRL) handle

ctrlType: The controller type

Return: The Kd value

CTRL_getKp ()

_iq CTRL_getKp (CTRL_Handle handle,const CTRL_Type_e ctrlType)

Gets the controller state

Handle: The controller (CTRL) handle

ctrlType: The controller type

Return: The Kp value

Copyright © 2013–2019, Texas Instruments Incorporated

CTRL_getLhf ()

inline float_t CTRL_getLhf(CTRL_Handle handle)

Gets the high frequency inductance (Lhf) value from the controller

Handle: The controller (CTRL) handle

Return: Return: The Lhf value, obj->Lhf

CTRL_getMagCurrent_pu ()

_iq CTRL_getMagCurrent_pu(CTRL_Handle handle)

Gets the magnetizing current value from the controller

Handle: The controller (CTRL) handle

Return: The magnetizing current value

CTRL_getMaxVsMag_pu ()

inline _iq CTRL_getMaxVsMag_pu(CTRL_Handle handle)

Gets the maximum voltage vector

Handle: The controller (CTRL) handle

Return: The maximum voltage vector (value between 0 and 4/3)

CTRL_getMaximumSpeed_pu ()

_iq CTRL_getMaximumSpeed_pu(CTRL_Handle handle);

Gets the maximum speed value from the controller

Handle: The controller (CTRL) handle

Return: The maximum voltage vector (value between 0 and 4/3)

CTRL_getMotorRatedFlux ()

inline float_t CTRL_getMotorRatedFlux(CTRL_Handle handle)

Gets the motor rated flux from the controller

Handle: The controller (CTRL) handle

Return: The motor rated flux, V*sec, obj->motorParams.ratedFlux_VpHz

CTRL_getMotorType ()

inline MOTOR_Type_e CTRL_getMotorType(CTRL_Handle handle)

Gets the motor type from the controller

Handle: The controller (CTRL) handle

Return: The motor type, obj-> motorParams.type

CTRL_getNumCtrlTicksPerCurrentTick ()

inline uint_least16_t CTRL_getNumCtrlTicksPerCurrentTick(CTRL_Handle handle)

Gets the number of controller clock ticks per current controller clock tick

Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The controller (CTRL) handle

Return: The number of controller clock ticks per estimator clock tick, obj->numCtrlTicksPerCurrentTick

CTRL_getNumCtrlTicksPerSpeedTick ()

inline uint_least16_t CTRL_getNumCtrlTicksPerSpeedTick(CTRL_Handle handle)

Gets the number of controller clock ticks per speed controller clock tick

Handle: The controller (CTRL) handle

Return: The number of controller clock ticks per speed clock tick, obj->numCtrlTicksPerSpeedTick

CTRL_getNumCtrlTicksPerTrajTick ()

inline uint_least16_t CTRL_getNumCtrlTicksPerTrajTick(CTRL_Handle handle)

Gets the number of controller clock ticks per trajectory clock tick

Handle: The controller (CTRL) handle

Return: The number of controller clock ticks per trajectory clock tick, obj->numCtrlTicksPerTrajTick

CTRL_getNumIsrTicksPerCtrlTick ()

inline uint_least16_t CTRL_getNumIsrTicksPerCtrlTick(CTRL_Handle handle)

Gets the number of Interrupt Service Routine (ISR) clock ticks per controller clock tick

Handle: The controller (CTRL) handle

Return: The number of Interrupt Service Routine (ISR) clock ticks per controller clock tick, obj->numIsrTicksPerCtrlTick

CTRL_getRefValue_pu ()

inline _iq CTRL_getRefValue_pu(CTRL_Handle handle,const CTRL_Type_e ctrlType)

Gets the reference value from the specified controller

Handle: The controller (CTRL) handle

ctrlType: The controller type

Return: The reference value, pu

CTRL_getRhf ()

inline float_t CTRL_getRhf(CTRL_Handle handle)

Gets the high frequency resistance (Rhf) value from the controller

Handle: The controller (CTRL) handle

Return: The Rhf value, obj->Rhf

CTRL_getRoverL ()

inline float_t CTRL_getRoverL(CTRL_Handle handle)

Gets the R/L value from the controller

Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The controller (CTRL) handle

Return: The the R/L value, obj-> RoverL

CTRL_getSpd_max_pu () *i*

nline _iq CTRL_getSpd_max_pu(CTRL_Handle handle)

Gets the maximum speed value from the controller

Handle: The controller (CTRL) handle

Return: The maximum speed value, pu, obj-> spd_max

CTRL_getSpd_out_addr ()

inline _iq *CTRL_getSpd_out_addr(CTRL_Handle handle)

Gets the output speed memory address from the controller

Handle: The controller (CTRL) handle

Return: The output speed memory address, obj-> spd_out

CTRL_getSpd_out_pu ()

inline _iq CTRL_getSpd_out_pu(CTRL_Handle handle)

Gets the output speed value from the controller

Handle: The controller (CTRL) handle

Return: The output speed value, pu, obj-> spd_out

CTRL_getSpd_ref_pu ()

inline _iq CTRL_getSpd_ref_pu(CTRL_Handle handle)

Gets the output speed reference value from the controller

Handle: The controller (CTRL) handle

Return: The output speed reference value, pu, obj-> spd_ref

CTRL_getSpd_int_ref_pu ()

inline _iq CTRL_getSpd_int_ref_pu(CTRL_Handle handle)

Gets the output speed intermediate reference value from the controller

Handle: The controller (CTRL) handle

Return: The output speed intermediate reference value, pu, obj-> trajHandle_spd

CTRL_getState () *CTRL_State_e CTRL_getState(CTRL_Handle handle)*

Gets the controller state

Handle: The controller (CTRL) handle

Return: The controller state

Copyright © 2013–2019, Texas Instruments Incorporated

CTRL_getTrajFreq ()

inline uint_least32_t CTRL_getTrajFreq(CTRL_Handle handle)

Gets the trajectory execution frequency

Handle: The controller (CTRL) handle

Return: The trajectory execution frequency, Hz, obj->trajFreq_Hz

CTRL_getTrajPeriod_sec ()

inline _iq CTRL_getTrajPeriod_sec(CTRL_Handle handle)

Gets the trajectory execution frequency

Handle: The controller (CTRL) handle

Return: The trajectory execution period, sec, obj-> trajPeriod_sec

CTRL_getTrajStep ()

void CTRL_getTrajStep(CTRL_Handle handle);

Gets the trajectory step size

Handle: The controller (CTRL) handle

Return: The trajectory execution frequency, Hz

CTRL_getUi ()

inline _iq CTRL_getUi(CTRL_Handle handle,const CTRL_Type_e ctrlType)

Gets the integrator (Ui) value from the specified controller

Handle: The controller (CTRL) handle

Return: The Ui value

CTRL_getVab_in_pu ()

void CTRL_getVab_in_pu(CTRL_Handle handle,MATH_vec2 *pVab_in_pu);

Gets the alpha/beta voltage input vector values from the controller

Handle: The controller (CTRL) handle

pVab_in_pu: The vector for the alpha/beta voltage input vector values, pu

CTRL_getVab_out_addr ()

inline MATH_vec2 *CTRL_getVab_out_addr(CTRL_Handle handle)

Gets the alpha/beta voltage output vector memory address from the controller

Handle: The controller (CTRL) handle

Return: The alpha/beta voltage output vector memory address, &(obj->Vab_out)

CTRL_getVab_out_pu ()

void CTRL_getVab_out_pu(CTRL_Handle handle,MATH_vec2 *pVab_out_pu);

Gets the alpha/beta voltage output vector values from the controller

Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The controller (CTRL) handle

Return: The vector for the alpha/beta voltage output vector values, pu

CTRL_getVd_out_addr ()

inline _iq *CTRL_getVd_out_addr(CTRL_Handle handle)

Gets the direct voltage output value memory address from the controller

Handle: The controller (CTRL) handle

Return: The direct voltage output value memory address, &(obj->Vdq_out.value[0])

CTRL_getVd_out_pu ()

inline _iq CTRL_getVd_out_pu(CTRL_Handle handle)

Gets the direct voltage output value from the controller

Handle: The controller (CTRL) handle

Return: The direct voltage output value, pu, obj->Vdq_out.value[0])

CTRL_getVdq_out_addr ()

inline MATH_vec2 *CTRL_getVdq_out_addr(CTRL_Handle handle)

Gets the direct/quadrature voltage output vector memory address from the controller

Handle: The controller (CTRL) handle

Return: The direct/quadrature voltage output vector memory address, &(obj->Vdq_out)

CTRL_getVdq_out_pu ()

void CTRL_getVdq_out_pu(CTRL_Handle handle,MATH_vec2 *pVdq_out_pu);

Gets the direct/quadrature voltage output vector values from the controller

Handle: The controller (CTRL) handle

pVdq_out_pu: The vector for the direct/quadrature voltage output vector values, pu

CTRL_getVersion () *void CTRL_getVersion(CTRL_Handle handle,CTRL_Version *pVersion);*

Gets the controller version number

Handle: The controller (CTRL) handle

pVersion: A pointer to the version

CTRL_getVq_out_addr ()

inline _iq *CTRL_getVq_out_addr(CTRL_Handle handle)

Gets the quadrature voltage output value memory address from the controller

Handle: The controller (CTRL) handle

Return: The quadrature voltage output value memory address, &(obj->Vdq_out.value[1])

Copyright © 2013–2019, Texas Instruments Incorporated

CTRL_getVq_out_pu ()

inline _iq CTRL_getVq_out_pu(CTRL_Handle handle)

Gets the quadrature voltage output value from the controller

Handle: The controller (CTRL) handle

Return: The quadrature voltage output value, pu, obj->Vdq_out.value[1]

CTRL_getWaitTime ()

inline uint_least32_t CTRL_getWaitTime(CTRL_Handle handle,const CTRL_State_e ctrlState)

Gets the wait time for a given state

Handle: The controller (CTRL) handle

ctrlState: The controller state

Return: The wait time, controller clock counts, waitTimes[ctrlState]

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.1.4 CTRL Counter Functions

CTRL_incrCounter_current ()

inline void CTRL_incrCounter_current(CTRL_Handle handle)

Increments the current counter

Handle: The controller (CTRL) handle

counter_current: Member of CTRL object that is incremented

CTRL_incrCounter_isr ()

inline void CTRL_incrCounter_isr(CTRL_Handle handle)

Increments the ISR counter

Handle: The controller (CTRL) handle

counter_isr: Member of CTRL object that is incremented

CTRL_incrCounter_speed ()

inline void CTRL_incrCounter_speed(CTRL_Handle handle)

Increments the speed counter

Handle: The controller (CTRL) handle

counter_speed: Member of CTRL object that is incremented

CTRL_incrCounter_state ()

inline void CTRL_incrCounter_state(CTRL_Handle handle)

Increments the state counter

Handle: The controller (CTRL) handle

counter_state: Member of CTRL object that is incremented

CTRL_incrCounter_traj ()

inline void CTRL_incrCounter_traj(CTRL_Handle handle)

Increments the trajectory counter

Handle: The controller (CTRL) handle

counter_traj: Member of CTRL object that is incremented

CTRL_resetCounter_current ()

inline void CTRL_resetCounter_current(CTRL_Handle handle)

Resets the current counter to 0

Handle: The controller (CTRL) handle

counter_current: Member of CTRL object that is reset to 0

Copyright © 2013–2019, Texas Instruments Incorporated

CTRL_resetCounter_isr ()

inline void CTRL_resetCounter_isr(CTRL_Handle handle)

Resets the ISR counter to 0

Handle: The controller (CTRL) handle

counter_isr: Member of CTRL object that is reset to 0

CTRL_resetCounter_speed ()

inline void CTRL_resetCounter_speed(CTRL_Handle handle)

Resets the speed counter to 0

Handle: The controller (CTRL) handle

counter_speed: Member of CTRL object that is reset to 0

CTRL_resetCounter_state ()

inline void CTRL_resetCounter_state(CTRL_Handle handle)

Resets the state counter to 0

Handle: The controller (CTRL) handle

counter_state: Member of CTRL object that is reset to 0

CTRL_resetCounter_traj ()

inline void CTRL_resetCounter_traj(CTRL_Handle handle)

Resets the trajectory counter to 0

Handle: The controller (CTRL) handle

counter_traj: Member of CTRL object that is reset to 0

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.1.5 CTRL Set Functions

CTRL_setCtrlFreq_Hz ()

nline void CTRL_setCtrlFreq_Hz(CTRL_Handle handle,const uint_least32_t ctrlFreq_Hz)

Sets the controller frequency

Handle: The controller (CTRL) handle

ctrlFreq_Hz: The controller frequency, Hz

CTRL_setCtrlFreq_sec ()

inline void CTRL_setCtrlPeriod_sec(CTRL_Handle handle,const float_t ctrlPeriod_sec)

Sets the controller execution period

Handle: The controller (CTRL) handle

ctrlPeriod_sec: The controller execution period, sec

CTRL_setErrorCode ()

inline void CTRL_setErrorCode(CTRL_Handle handle,const CTRL_ErrorCode_e errorCode)

Sets the error code in the controller

Handle: The controller (CTRL) handle

errorCode: The error code

CTRL_setEstParams ()

void CTRL_setEstParams(EST_Handle estHandle,USER_Params *pUserParams);

Sets the default estimator parameters. Copies all scale factors that are defined in the file user.h and used by CTRL into the CTRL object.

estHandle: The estimator (EST) handle

pUserParams: The pointer to the user parameters

CTRL_setFlag_enableCtrl ()

void CTRL_setFlag_enableCtrl(CTRL_Handle handle,const bool_t state)

Enables the FOC controller (enables the motor controller)

Handle: The controller (CTRL) handle

State: The desired state:

- True -> Enable the controller
- False -> Disable the controller

CTRL_setFlag_enableDcBusComp ()

void CTRL_setFlag_enableDcBusComp (CTRL_Handle handle,bool_t state)

Sets the enable DC bus compensation flag value in the estimator. The DC bus

Copyright © 2013–2019, Texas Instruments Incorporated

compensation algorithm will compensate the Iq and Id PI controller's.

Handle: The controller (CTRL) handle

State: Boolean of desired state

CTRL_setFlag_enablePowerWarp ()

inline void CTRL_setFlag_enablePowerWarp(CTRL_Handle handle,const bool state)

Sets the PowerWarp enable flag value in the estimator. PowerWarp is only used when controlling an induction motor. PowerWarp adjusts field levels so that the least amount of power is used according to the load on the motor.

Handle: The controller (CTRL) handle

State: Boolean of desired state

CTRL_setFlag_enableOffset ()

void CTRL_setFlag_enableOffset(CTRL_Handle handle,const bool_t state)

Enable or disable the voltage and current offset calibration

Handle: The controller (CTRL) handle

State: The desired state:

- True -> Perform the offset calibration
- False -> Do not perform the offset

calibrationCTRL_setFlag_enableSpeedCtrl ()

void CTRL_setFlag_enableSpeedCtrl(CTRL_Handle handle,const bool_t state)

Enables speed control mode or enables torque control mode (connects the speed PI output to Iq)

Handle: The controller (CTRL) handle

State: The desired state:

- True -> Enable speed control (connect the speed PI output to the Iq input)
- False -> Disable speed control (dis-connect the speed PI from the Iq. Iq is available for direct input)

CTRL_setFlag_enableUserMotorParams ()

void CTRL_setFlag_enableUserMotorParams(CTRL_Handle handle,const bool_t state)

Sets the enable user motor parameters flag value in the estimator

Handle: The controller (CTRL) handle

State: The desired state:

- True -> Use the user motor parameters from user.h
- False -> Perform motor parameter estimation

CTRL_setGains ()

void CTRL_setGains(CTRL_Handle handle,const CTRL_Type_e ctrlType, const _iq Kp,const _iq Ki,const _iq Kd);

Copyright © 2013–2019, Texas Instruments Incorporated

Sets the gain values for the specified controller

Handle: The controller (CTRL) handle

ctrlType: The controller type

Kp: The Kp gain value, pu

Ki: The Ki gain value, pu

Kd: The Kd gain value, pu

CTRL_setIab_in_pu ()

inline void CTRL_setIab_in_pu(CTRL_Handle handle,const MATH_vec2 *pIab_in_pu)

Sets the alpha/beta current (Iab) input vector values in the controller

Handle: The controller (CTRL) handle

pIab_in_pu: The vector of the alpha/beta current input vector values, pu

CTRL_setIab_filt_pu ()

void CTRL_setIab_filt_pu(CTRL_Handle handle,const MATH_vec2 *pIab_filt_pu);

Sets the alpha/beta filtered current vector values in the controller

Handle: The controller (CTRL) handle

pIab_filt_pu: The vector of the alpha/beta filtered current vector values, pu

CTRL_setId_ref_pu ()

inline void CTRL_setId_ref_pu(CTRL_Handle handle,const _iq Id_ref_pu)

Sets the direct current (Id) reference value in the controller

Handle: The controller (CTRL) handle

Id_ref_pu: The quadrature current reference value, pu; obj->Idq_ref.value[0]

CTRL_setIdq_in_pu ()

inline void CTRL_setIdq_in_pu(CTRL_Handle handle,const MATH_vec2 *pIdq_in_pu)

Sets the direct/quadrature current (Idq) input vector values in the controller

Handle: The controller (CTRL) handle

pIdq_in_pu: The vector of the direct/quadrature current input vector values, pu; obj-> Idq_in.value [0,1]

CTRL_setIdq_ref_pu ()

inline void CTRL_setIdq_ref_pu(CTRL_Handle handle,const MATH_vec2 *pIdq_ref_pu)

Sets the direct/quadrature current (Idq) reference vector values in the controller

Handle: The controller (CTRL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

pIdq_ref_pu: The vector of the direct/quadrature current reference vector values, pu, obj-> Idq_ref.value[0,1]

CTRL_setIdRated_pu ()

inline void CTRL_setIdRated_pu(CTRL_Handle handle,const _iq IdRated_pu)

Sets the Id rated current value in the controller

Handle: The controller (CTRL) handle

IdRated_pu: The Id rated current value, pu, obj-> IdRated

CTRL_setIq_ref_pu ()

void CTRL_setIq_ref_pu (CTRL_Handle handle, const _iq IqRef_pu)

Sets the quadrature current (Iq) reference value in the controller

Handle: The controller (CTRL) handle

IqRef_pu: The quadrature current reference value, pu, Idq_ref.value[1]

CTRL_setKd ()

void CTRL_setKd (CTRL_Handle handle, const CTRL_Type_e ctrlType,const _iq Kd)

Sets the derivative gain (Kd) value for the specified controller (speed, Id, or Iq)

Handle: The controller (CTRL) handle

ctrlType: The controller type Kd: The Kd value

CTRL_setKi ()

void CTRL_setKi (CTRL_Handle handle, const CTRL_Type_e ctrlType,const _iq Ki)

Sets the integral gain (Ki) value for the specified controller (speed, Id, or Iq)

Handle: The controller (CTRL) handle

ctrlType: The controller type Ki: The Ki value

CTRL_setKp ()

void CTRL_setKp (CTRL_Handle handle, const CTRL_Type_e ctrlType,const _iq Kp)

Sets the proportional gain (Kp) value for the specified controller (speed, Id, or Iq)

Handle: The controller (CTRL) handle

ctrlType: The controller type

Kp: The Kp value

CTRL_setLhf ()

inline void CTRL_setLhf(CTRL_Handle handle,const float_t Lhf)

Sets the high frequency inductance (Lhf) value in the controller

Handle: The controller (CTRL) handle

ctrlType: The controller type

Lhf: The Lhf value

Copyright © 2013–2019, Texas Instruments Incorporated

CTRL_setMagCurrent_pu ()

void CTRL_setMagCurrent_pu(CTRL_Handle handle,const _iq magCurrent_pu);

Sets the magnetizing current value in the controller

Handle: The controller (CTRL) handle

magCurrent_pu: The magnetizing current value, pu

CTRL_setMaxVsMag_pu ()

inline void CTRL_setMaxVsMag_pu(CTRL_Handle handle,const _iq maxVsMag)

Sets the maximum voltage vector in the controller

Handle: The controller (CTRL) handle

maxVsMag: The maximum voltage vector (value between 0 and 4/3), obj->maxVsMag_pu

CTRL_setMaxAccel_pu ()

inline void CTRL_setMaxAccel_pu(CTRL_Handle handle,const _iq maxAccel_pu)

Sets the maximum acceleration of the speed controller. Sets the maximum acceleration rate of the speed reference.

Handle: The controller (CTRL) handle

maxAccel_pu: The maximum acceleration (value between 0 and 1), pu, obj->traj_spd.maxDelta

CTRL_setMaximumSpeed_pu ()

void CTRL_setMaximumSpeed_pu(CTRL_Handle handle,const _iq maxSpeed_pu);

Sets the maximum speed value in the controller.

Handle: The controller (CTRL) handle

maxSpeed_pu: The maximum speed value, pu

CTRL_setParams() *void CTRL_setParams(CTRL_Handle handle,USER_Params *pUserParams)*

Sets the default controller parameters. This function allows for updates in scale factors during real-time operation of the controller.

Handle: The controller (CTRL) handle

pUserParams: The pointer to the user parameters

CTRL_setNumCtrlTicksPerCurrentTick ()

inline void CTRL_setNumCtrlTicksPerCurrentTick(CTRL_Handle handle, const uint_least16_t numCtrlTicksPerCurrentTick)

Sets the number of controller clock ticks per current controller clock tick.

Handle: The controller (CTRL) handle

numCtrlTicksPerCurrentTick: The number of controller clock ticks per estimator clock tick, obj->numCtrlTicksPerCurrentTick

Copyright © 2013–2019, Texas Instruments Incorporated

CTRL_setNumCtrlTicksPerSpeedTick ()

inline void CTRL_setNumCtrlTicksPerSpeedTick(CTRL_Handle handle,,const uint_least16_t numCtrlTicksPerSpeedTick)numCtrlTicksPerCurrentTick)

Sets the number of controller clock ticks per speed controller clock tick.

Handle: The controller (CTRL) handle

numCtrlTicksPerSpeedTick: The number of controller clock ticks per speed clock tick, obj-> numCtrlTicksPerSpeedTick

CTRL_setNumCtrlTicksPerTrajTick ()

inline void CTRL_setNumCtrlTicksPerTrajTick(CTRL_Handle handle, const uint_least16_t numCtrlTicksPerTrajTick)

Sets the number of controller clock ticks per trajectory clock tick.

Handle: The controller (CTRL) handle

numCtrlTicksPerTrajTick: The number of controller clock ticks per trajectory clock tick, obj-> numCtrlTicksPerTrajTick

CTRL_setNumIsrTicksPerCtrlTick () *inline void CTRL_setNumIsrTicksPerCtrlTick(CTRL_Handle handle, const uint_least16_t numIsrTicksPerCtrlTick)*

Sets the number of Interrupt Service Routine (ISR) clock ticks per controller clock tick.

Handle: The controller (CTRL) handle

numIsrTicksPerCtrlTick: The number of ISR clock ticks per controller clock tick

CTRL_setRhf () *inline void CTRL_setRhf(CTRL_Handle handle,const float_t Rhf)*

Sets the high frequency resistance (Rhf) value in the controller

Handle: The controller (CTRL) handle

Rhf: The Rhf value, obj->Rhf

CTRL_setRoverL () *inline void CTRL_setRoverL(CTRL_Handle handle,const float_t RoverL)*

Sets the R/L value in the controller

Handle: The controller (CTRL) handle

RoverL: The R/L value, obj-> RoverL

CTRL_setSpdMax () *void CTRL_setSpdMax (CTRL_Handle handle, const _iq spdMax)*

Sets the PI speed reference value that is located in the controller

Handle: The controller (CTRL) handle

spdMax: The maximum allowed output of the speed controller

CTRL_setSpd_max_pu ()

inline void CTRL_setSpd_max_pu(CTRL_Handle handle,const _iq maxSpd_pu)

Copyright © 2013–2019, Texas Instruments Incorporated

Sets the maximum speed value in the controller

Handle: The controller (CTRL) handle

maxSpd_pu: The maximum speed value, pu

CTRL_setSpd_out_pu ()

inline void CTRL_setSpd_out_pu(CTRL_Handle handle,const _iq spd_out_pu)

Sets the output speed value in the controller

Handle: The controller (CTRL) handle

spd_out_pu: The output speed value, pu

CTRL_setSpd_ref_pu ()

void CTRL_setSpd_ref_pu(CTRL_Handle handle,const _iq spd_ref_pu);

Sets the output speed reference value in the controller

Handle: The controller (CTRL) handle

spd_ref_pu: The output speed reference value, pu

CTRL_setSpd_ref_krpm ()

void CTRL_setSpd_ref_krpm(CTRL_Handle handle,const _iq spd_ref_krpm)

Sets the PI speed reference value that is located in the controller

Handle: The controller (CTRL) handle

spd_ref_krpm: The output speed reference value, kilo-rpm

CTRL_setState ()

inline void CTRL_setState(CTRL_Handle handle,const CTRL_State_e state)

Sets the controller state

Handle: The controller (CTRL) handle

state: The new state

CTRL_setTrajFreq_Hz ()

inline void CTRL_setTrajFreq_Hz(CTRL_Handle handle,const uint_least32_t trajFreq_Hz)

Sets the trajectory execution frequency

Handle: The controller (CTRL) handle

trajFreq_Hz: The trajectory execution frequency, Hz, obj->trajFreq_Hz

CTRL_setTrajPeriod_sec ()

inline void CTRL_setTrajPeriod_sec(CTRL_Handle handle,const _iq trajPeriod_sec)

Sets the trajectory execution period

Handle: The controller (CTRL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

trajPeriod_sec: The trajectory execution period, sec, obj->trajPeriod_sec

CTRL_setUi ()

inline void CTRL_setUi(CTRL_Handle handle,const CTRL_Type_e ctrlType,const _iq Ui)

Sets the integrator (Ui) value in the specified controller (speed, Id, or Iq)

Handle: The controller (CTRL) handle

ctrlType: The controller type

Ui: Ui value

CTRL_setupClarke_I ()

void CTRL_setupClarke_I(CTRL_Handle handle,uint_least8_t numCurrentSensors);

Sets the number of current sensors. Different algorithms are used for calculating the Clarke transform when different number of currents are read in.

Handle: The controller (CTRL) handle

ctrlType: The controller type

numCurrentSensors: The number of current sensors

CTRL_setupClarke_V ()

void CTRL_setupClarke_V(CTRL_Handle handle,uint_least8_t numVoltageSensors);

Sets the number of voltage sensors. Different algorithms are used for calculating the Clarke transform when different number of voltages are read in.

Handle: The controller (CTRL) handle

ctrlType: The controller type

numVoltageSensors: The number of voltage sensors

CTRL_setupEstIdleState ()

void CTRL_setupEstIdleState(CTRL_Handle handle);

Sets up the controller and trajectory generator for the estimator idle state

Handle: The controller (CTRL) handle

Handle: The controller (CTRL) handle

:

CTRL_setupEstOnLineState ()

void CTRL_setupEstOnLineState(CTRL_Handle handle);

Sets up the controller and trajectory generator for the estimator idle state

Handle: The controller (CTRL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

CTRL_setUserMotorParams ()

void CTRL_setUserMotorParams(CTRL_Handle handle);

Sets the controller and estimator with motor parameters from the user.h file

Handle: The controller (CTRL) handle

CTRL_setVab_in_pu ()

inline void CTRL_setVab_in_pu(CTRL_Handle handle,const MATH_vec2 *pVab_in_pu)

Sets the alpha/beta voltage input vector values in the controller

Handle: The controller (CTRL) handle

CTRL_setVab_out_pu ()

inline void CTRL_setVab_out_pu(CTRL_Handle handle,const MATH_vec2 *pVab_out_pu)

Sets the alpha/beta voltage output vector values in the controller

Handle: The controller (CTRL) handle

CTRL_setVdq_out_pu ()

inline void CTRL_setVdq_out_pu(CTRL_Handle handle,const MATH_vec2 *pVdq_out_pu)

Sets the direct/quadrature voltage output vector values in the controller

Handle: The controller (CTRL) handle

pVdq_out_pu: The vector of direct/quadrature voltage output vector values, pu

CTRL_setWaitTimes () *void CTRL_setWaitTimes(CTRL_Handle handle,const uint_least32_t *pWaitTimes)*

Sets the wait times for the controller states

Handle: The controller (CTRL) handle

pWaitTimes: A pointer to a vector of wait times, controller clock counts

CTRL_setup () *void CTRL_setup(CTRL_Handle handle)*

Sets up the controllers

Handle: The controller (CTRL) handle

CTRL_setupCtrl () *void CTRL_setupCtrl(CTRL_Handle handle);*

Sets up the controller (CTRL) object and all of the subordinate objects (Runs the InstaSPIN state machine)

Handle: The controller (CTRL) handle

CTRL_setupEst () *void CTRL_setupEst(CTRL_Handle handle);*

Copyright © 2013–2019, Texas Instruments Incorporated

Sets up the controller (CTRL) object and all of the subordinate objects (Runs the InstaSPIN state machine)

Handle: The controller (CTRL) handle

CTRL_setupTraj () *void CTRL_setupTraj(CTRL_Handle handle);*

Sets up the trajectory (TRAJ) object

Handle: The controller (CTRL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.1.6 CTRL Run and Compute Functions

CTRL_angleDelayComp ()

inline _iq CTRL_angleDelayComp(CTRL_Handle handle, const _iq angle_pu)

Runs angle delay compensation. This function takes the decimation rates to calculate a phase delay, and it compensates the delay. This allows the voltage output to do a better correction of the error.

Handle: The controller (CTRL) handle

angle_pu: The angle delayed

Return: The phase delay compensated angle, angleComp_pu

CTRL_computePhasor ()

inline void CTRL_computePhasor(const _iq angle_pu,MATH_vec2 *pPhasor)

Computes a phasor for a given angle

angle_pu: The angle, pu

pPhasor: The pointer to the phasor vector values

CTRL_doCurrentCtrl ()

inline bool CTRL_doCurrentCtrl(CTRL_Handle handle)

Determines if the current controllers should be run

Handle: The controller (CTRL) handle

Return: The value denoting that the current controllers should be run (true) or not (false), result

CTRL_doSpeedCtrl ()

inline bool CTRL_doSpeedCtrl(CTRL_Handle handle)

Determines if the speed controller should be executed

Handle: The controller (CTRL) handle

Return: A Boolean value denoting if the speed controller should be executed (true) or not (false)

CTRL_run()

void CTRL_run(CTRL_Handle handle,HAL_Handle halHandle,const HAL_AdcData_t *pAdcData, HAL_PwmData_t *pPwmData)

Runs the motor controller calculations, must be called at the ISR rate

Handle: The controller (CTRL) handle

halHandle: The driver (HAL) handle

pAdcData: The pointer to the ADC data in “HAL_AdcData_t” type format

pPwmData: The pointer to the PWM data in “HAL_AdcData_t” type format

CTRL_runTraj ()

void CTRL_runTraj(CTRL_Handle handle)

Runs the trajectory

Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The controller (CTRL) handle

CTRL_runOffLine ()

inline void CTRL_runOffLine(CTRL_Handle handle,HAL_Handle halHandle, const HAL_AdcData_t *pAdcData,HAL_PwmData_t *pPwmData)

Runs the offline controller

Handle: The controller (CTRL) handle

halHandle: The hardware abstraction layer (HAL) handle

pAdcData: The pointer to the ADC data

pPwmData: The pointer to the PWM data

CTRL_runOnLine ()

inline void CTRL_runOnLine(CTRL_Handle handle,const HAL_AdcData_t *pAdcData,HAL_PwmData_t *pPwmData)

Runs the online controller

Handle: The controller (CTRL) handle

pAdcData: The pointer to the ADC data

pPwmData: The pointer to the PWM data

CTRL_runOnLine_User ()

inline void CTRL_runOnLine_User(CTRL_Handle handle, const HAL_AdcData_t *pAdcData,HAL_PwmData_t *pPwmData)

Runs the online controller

Handle: The controller (CTRL) handle

pAdcData: The pointer to the ADC data

pPwmData: The pointer to the PWM data

CTRL_useZeroIq_ref ()

inline bool CTRL_useZeroIq_ref(CTRL_Handle handle)

Determines if a zero Iq current reference should be used in the controller

Handle: The controller (CTRL) handle

Return: A Boolean value denoting if a zero Iq current reference should be used (true) or not (false)

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.2 Estimator API Functions – FAST Library – est.h, est_states.h

3.3.2.1 EST Enumerations and Structures

EST_RsOnLineFilterType_e

Enumeration for the Rs online filter types

```
typedef enum
{
  EST_RsOnLineFilterType_Current=0,        < Current Filter
  EST_RsOnLineFilterType_Voltage           < Voltage Filter
} EST_RsOnLineFilterType_e;
```

EST_ErrorCode_e

Enumeration for the estimator error codes

```
typedef enum
{
  EST_ErrorCode_NoError=0,               < no error error code
  EST_ErrorCode_Flux_OL_ShiftOverFlow,   < flux open loop shift overflow error
code
  EST_ErrorCode_FluxError,               < flux estimator error code
  EST_ErrorCode_Dir_ShiftOverFlow,       < direction shift overflow error code
  EST_ErrorCode_Ind_ShiftOverFlow,       < inductance shift overflow error code
  EST_numErrorCodes                      < the number of estimator error codes
} EST_ErrorCode_e;
```

EST_State_e

Enumeration for the estimator states

```
typedef enum
{
  EST_State_Error=0,            < error
  EST_State_Idle,               < idle
  EST_State_RoverL,             < R/L estimation
  EST_State_Rs,                 < Rs estimation state
  EST_State_RampUp,             < ramp up the speed
  EST_State_IdRated,            < control Id and estimate the rated flux
  EST_State_RatedFlux_OL,       < estimate the open loop rated flux
  EST_State_RatedFlux,          < estimate the rated flux
  EST_State_RampDown,           < ramp down the speed
  EST_State_LockRotor,          < lock the rotor
  EST_State_Ls,                 < stator inductance estimation state
  EST_State_Rr,                 < rotor resistance estimation state
  EST_State_MotorIdentified,    < motor identified state
  EST_State_OnLine,             < online parameter estimation
  EST_numStates                 < the number of estimator states
} EST_State_e;
```


Copyright © 2013–2019, Texas Instruments Incorporated

3.3.2.2 EST Set Functions

EST_setRsOnLineId_pu ()

extern void EST_setRsOnLineId_pu(EST_Handle handle,const _iq Id_pu);

Gets the Id value used for online stator resistance estimation in per unit (pu), IQ24

Handle: The estimator (EST) handle

Return: The Id value, pu

EST_setAngle_pu ()

extern void EST_setAngle_pu(EST_Handle handle,const _iq angle_pu);

Sets the angle value in the estimator in per unit (pu), IQ24

This function overwrites the estimated angle with a user's provided angle. The set value should be between 0x00000000 or _IQ(0.0) to 0x00FFFFFF or _IQ(1.0). The following example shows how to overwrite the estimated angle:

```
_iq Overwrite_Flux_Angle_pu = _IQ(0.5);
EST_setAngle_pu(handle, Overwrite_Flux_Angle_pu);
```

This function is not recommended for general use, since this will automatically generate an axis misalignment between the rotor flux axis and the control signals driving the motor. The use of this function is recommended for advanced users interested in doing open loop startup algorithms that need to bypass the estimator.

Handle: The estimator (EST) handle

angle_pu: The angle value, pu

EST_setDcBus_pu ()

extern void EST_setDcBus_pu(EST_Handle handle,const _iq dcBus_pu);

Sets the DC bus voltage in the estimator in per unit (pu), IQ24

Handle: The estimator (EST) handle

dcBus_pu: The DC bus voltage, pu

EST_setDir_qFmt ()

extern void EST_setDir_qFmt(EST_Handle handle,const uint_least8_t dir_qFmt);

Sets the direction Q format in the estimator

Handle: The estimator (EST) handle

dir_qFmt: The direction Q format

EST_setFe_neg_max_pu ()

extern void EST_setFe_neg_max_pu(EST_Handle handle,const _iq fe_neg_max_pu);

Sets maximum negative electrical frequency from the estimator

Handle: The estimator (EST) handle

fe_neg_max_pu: The maximum negative electrical frequency, Hz

Copyright © 2013–2019, Texas Instruments Incorporated

EST_setFe_pos_min_pu ()

extern void EST_setFe_pos_min_pu(EST_Handle handle,const _iq fe_pos_min_pu);

Sets minimum positive electrical frequency from the estimator

Handle: The estimator (EST) handle

fe_pos_min_pu: The minimum positive electrical frequency, Hz

EST_setFlag_enableFluxControl ()

extern void EST_setFlag_enableFluxControl(EST_Handle handle,const bool state);

Sets the enable flux control flag in the estimator

Handle: The estimator (EST) handle

State: The desired flag state, on (1) or off (0)

EST_setFlag_enableForceAngle ()

void EST_setFlag_enableForceAngle (EST_Handle handle,const bool_t state)

Sets the enable force angle flag in the estimator

Enable or disable the DC measurement of Rs at startup of the motor

Handle: The estimator (EST) handle

State: The desired flag state, on (1) or off (0)

- TRUE: Enable forced angle. The estimated angle will be bypassed if the flux frequency falls below a threshold defined by:

```
#define USER_ZEROSPEEDLIMIT (0.001)
```

  in user.h. A typical value of this frequency is 0.001 of the full scale frequency defined in:

```
#define USER_IQ_FULL_SCALE_FREQ_Hz (500.0)
```

  Forced angle algorithm, when active, that is, when the rotor flux electrical frequency falls below the threshold, will be forcing a rotating angle at a frequency set by the following define:

```
#define USER_FORCE_ANGLE_FREQ_Hz  (1.0)
```

- FALSE: Disable forced angle. The estimator will never be bypassed by any forced angle algorithm.

EST_setFlag_enableRsOnLine ()

void EST_setFlag_enableRsOnLine(EST_Handle handle,const bool_t state)

Enables or disables the Rs online estimation in the estimator handle

Handle: The estimator (EST) handle

State: The desired flag state, on (1) or off (0)

EST_setFlag_enableRsRecalc ()

void EST_setFlag_enableRsRecalc(EST_Handle handle,const bool_t state)

Sets the enable stator resistance (Rs) re-calculation flag in the estimator

Copyright © 2013–2019, Texas Instruments Incorporated

Enable or disable the DC measurement of Rs at startup of the motor

Handle: The estimator (EST) handle

State: The desired flag state, on (1) or off (0)

EST_setFlag_estComplete () *extern void EST_setFlag_estComplete(EST_Handle handle,const bool state);*

Sets the estimation complete flag in the estimator

Handle: The estimator (EST) handle

State: The desired flag state, true (1) or false (0)

EST_setFlag_updateRs ()

void EST_setFlag_updateRs(EST_Handle handle,const bool_t state)

Sets the update stator resistance (Rs) flag in the estimator. Copies the Rs value from the Rs online estimator to the Rs value used by the InstaSpin angle estimator

Handle: The estimator (EST) handle

State: The desired flag state, on (1) or off (0)

EST_setForceAngleDelta_pu ()

extern void EST_setForceAngleDelta_pu(EST_Handle handle,const _iq angleDelta_pu);

Sets the force angle delta value in the estimator in per unit (pu), IQ24

This function sets a forced angle delta, which represents the increments to be added to or subtracted from the forced angle. The higher this value is, the higher frequency will be generated when the angle is forced (estimated angle is bypassed when in forced angle mode). By default the forced angle frequency is set in user.h. The following example shows how to set a forced angle frequency from Hertz (Hz) to per unit:

```
 #define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_EST_TICK  (1)
 #define USER_PWM_FREQ_kHz         (15.0)
 #define USER_ISR_FREQ_Hz          (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_EST_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_EST_TICK)
 #define USER_FORCE_ANGLE_FREQ_Hz  (1.0)

 _iq delta_hz_to_pu_sf = _IQ(1.0/(float_t)USER_EST_FREQ_Hz);
 _iq Force_Angle_Freq_Hz = _IQ(USER_FORCE_ANGLE_FREQ_Hz);
 _iq Force_Angle_Delta_pu = _IQmpy(Force_Angle_Freq_Hz, delta_hz_to_pu_sf);

 EST_setForceAngleDelta_pu(handle, Force_Angle_Delta_pu);
```

Handle: The estimator (EST) handle

angleDelta_pu: The force angle delta value, pu

EST_setFreqB0_lp_pu ()

extern void EST_setFreqB0_lp_pu(EST_Handle handle,const _iq b0_lp_pu);

Sets the low pass filter numerator value in the frequency estimator in per unit (pu), IQ30

Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The estimator (EST) handle

b0_lp_pu: The low pass filter numerator value, pu

EST_setFreqBeta_lp_pu ()

extern void EST_setFreqBeta_lp_pu(EST_Handle handle,const _iq beta_lp_pu);

Sets the value used to set the low pass pole location in the frequency estimator in per unit (pu), IQ30

Handle: The estimator (EST) handle

beta_lp_pu: The value used to set the filter pole location, pu

EST_setFullScaleCurrent ()

extern void EST_setFullScaleCurrent(EST_Handle handle,const float_t fullScaleCurrent);

Sets the full scale current in the estimator in Amperes (A)

Handle: The estimator (EST) handle

fullScaleCurrent: The full scale current, A

EST_setFullScaleFlux ()

extern void EST_setFullScaleFlux(EST_Handle handle,const float_t fullScaleFlux);

Sets the full scale flux value used in the estimator in Volts*seconds (V.s)

Handle: The estimator (EST) handle

fullScaleFlux: The full scale flux value, V*sec

EST_setFullScaleFreq ()

extern void EST_setFullScaleFreq(EST_Handle handle,const float_t fullScaleFreq);

Sets the full scale frequency in the estimator in Hertz (Hz)

Handle: The estimator (EST) handle

fullScaleFreq: The full scale frequency, Hz

EST_setFullScaleInductance ()

extern void EST_setFullScaleInductance(EST_Handle handle,const float_t fullScaleInductance);

Sets the full scale inductance in the estimator in Henries (H).

Handle: The estimator (EST) handle

fullScaleInductance: The full scale inductance, Henry

EST_setFullScaleResistance () *extern void EST_setFullScaleResistance(EST_Handle handle,const float_t fullScaleResistance);*

Sets the full scale resistance in the estimator in Ohms.

Handle: The estimator (EST) handle

Copyright © 2013–2019, Texas Instruments Incorporated

fullScaleResistance: The full scale resistance, Ohm

EST_setFullScaleVoltage ()

extern void EST_setFullScaleVoltage(EST_Handle handle,const float_t fullScaleVoltage);

Sets the full scale resistance in the estimator in Volts (V).

Handle: The estimator (EST) handle

fullScaleVoltage: The full scale voltage, V

EST_setIdle ()

extern void EST_setIdle(EST_Handle handle);

Sets the estimator to idle

Handle: The estimator (EST) handle

EST_setIdle_all ()

extern void EST_setIdle_all(EST_Handle handle);

Sets the estimator and all of the subordinate estimators to idle

Handle: The estimator (EST) handle

EST_setId_ref_pu ()

extern void EST_setId_ref_pu(EST_Handle handle,const _iq Id_ref_pu);

Sets the direct current (Id) reference value in the estimator in per unit (pu), IQ24

Handle: The estimator (EST) handle

Id_ref_pu: The Id reference value, pu

EST_setIdRated_pu ()

extern void EST_setIdRated_pu(EST_Handle handle,const _iq IdRated_pu);

Sets the Id rated current value in the estimator in per unit (pu), IQ24

Handle: The estimator (EST) handle

IdRated_pu: The Id rated current value, pu

EST_setIq_ref_pu ()

extern void EST_setIq_ref_pu(EST_Handle handle,const _iq Iq_ref_pu);

Sets the quadrature current (Iq) reference value in the estimator in per unit (pu), IQ24

Handle: The estimator (EST) handle

Iq_ref_pu: The Iq reference value, pu

EST_setLs_d_pu ()

extern void EST_setLs_d_pu(EST_Handle handle,const _iq Ls_d_pu);

Sets the direct stator inductance value in the estimator in per unit (pu), IQ30

The internal direct inductance (Ls_d) used by the estimator can be changed in real time by calling this function. An example showing how this is done is shown here:

```
#define USER_MOTOR_Ls_d  (0.012)

float_t fullScaleInductance = EST_getFullScaleInductance(handle);
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
 float_t Ls_coarse_max = _IQ30toF(EST_getLs_coarse_max_pu(handle));
 int_least8_t lShift =
ceil(log(USER_MOTOR_Ls_d/(Ls_coarse_max*fullScaleInductance))/log(2.0));
 uint_least8_t Ls_qFmt = 30 - lShift;
 float_t L_max = fullScaleInductance * pow(2.0,lShift);
 _iq Ls_d_pu = _IQ30(USER_MOTOR_Ls_d / L_max);

 EST_setLs_d_pu(handle, Ls_d_pu);
 EST_setLs_qFmt(handle, Ls_qFmt);
```

Handle: The estimator (EST) handle

Ls_d_pu: The direct stator inductance value, pu

EST_setLs_delta_pu ()

extern void EST_setLs_delta_pu(EST_Handle handle,const _iq Ls_delta_pu);

Sets the delta stator inductance value during fine estimation

Handle: The estimator (EST) handle

Ls_delta_pu: The delta stator inductance value, pu

EST_setLs_dq_pu () *extern void EST_setLs_dq_pu(EST_Handle handle,const MATH_vec2 *pLs_dq_pu);*

Sets the direct/quadrature stator inductance vector values in the estimator in per unit (pu), IQ30

The internal direct and quadrature inductances (Ls_d and Ls_q) used by the estimator can be changed in real time by calling this function. An example showing how this is done is shown here:

```
 #define USER_MOTOR_Ls_d  (0.012)
 #define USER_MOTOR_Ls_q  (0.027)

 float_t fullScaleInductance = EST_getFullScaleInductance(handle);
 float_t Ls_coarse_max = _IQ30toF(EST_getLs_coarse_max_pu(handle));
 int_least8_t lShift =
ceil(log(USER_MOTOR_Ls_d/(Ls_coarse_max*fullScaleInductance))/log(2.0));
 uint_least8_t Ls_qFmt = 30 - lShift;
 float_t L_max = fullScaleInductance * pow(2.0,lShift);
 MATH_vec2 Ls_dq_pu;

 Ls_dq_pu.value[0] = _IQ30(USER_MOTOR_Ls_d / L_max);
 Ls_dq_pu.value[1] = _IQ30(USER_MOTOR_Ls_q / L_max);

 EST_setLs_dq_pu(handle, &Ls_dq_pu);
 EST_setLs_qFmt(handle, Ls_qFmt);
```

Handle: The estimator (EST) handle

pLs_dq_pu: The pointer to the direct/quadrature stator inductance vector values, pu

EST_setLs_q_pu () *extern void EST_setLs_q_pu(EST_Handle handle,const _iq Ls_q_pu);*

Sets the quadrature stator inductance value in the estimator in per unit (pu), IQ30

The internal quadrature inductance (Ls_q) used by the estimator can be changed in real time by calling this function. An example showing how this is done is shown here:

```
 #define USER_MOTOR_Ls_q  (0.027)

 float_t fullScaleInductance = EST_getFullScaleInductance(handle);
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
 float_t Ls_coarse_max = _IQ30toF(EST_getLs_coarse_max_pu(handle));
 int_least8_t lShift =
ceil(log(USER_MOTOR_Ls_q/(Ls_coarse_max*fullScaleInductance))/log(2.0));
 uint_least8_t Ls_qFmt = 30 - lShift;
 float_t L_max = fullScaleInductance * pow(2.0,lShift);
 _iq Ls_d_pu = _IQ30(USER_MOTOR_Ls_q / L_max);

 EST_setLs_q_pu(handle, Ls_q_pu);
 EST_setLs_qFmt(handle, Ls_qFmt);
```

Handle: The estimator (EST) handle

Ls_q_pu: The quadrature stator inductance value, pu

EST_setLs_qFmt () *extern void EST_setLs_qFmt(EST_Handle handle,const uint_least8_t Ls_qFmt);*

Sets the stator inductance Q format in the estimator in 8 bit unsigned integer (uint_least8_t)

Updating the internal inductance also requires to update the Q format variable, which is used to extend the covered range. This qFmt (Q Format) variable creates a floating point using fixed point math. It is important to notice that the inductance Q Format set by calling EST_setLs_qFmt() will be used by both per unit inductance calculations Ls_d and Ls_q. An example showing how this Q Format is set is shown below:

```
 #define USER_MOTOR_Ls_d  (0.012)
 #define USER_MOTOR_Ls_q  (0.027)

 float_t fullScaleInductance = EST_getFullScaleInductance(handle);
 float_t Ls_coarse_max = _IQ30toF(EST_getLs_coarse_max_pu(handle));
 int_least8_t lShift =
ceil(log(USER_MOTOR_Ls_d/(Ls_coarse_max*fullScaleInductance))/log(2.0));
 uint_least8_t Ls_qFmt = 30 - lShift;
 float_t L_max = fullScaleInductance * pow(2.0,lShift);
 MATH_vec2 Ls_dq_pu;

 Ls_dq_pu.value[0] = _IQ30(USER_MOTOR_Ls_d / L_max);
 Ls_dq_pu.value[1] = _IQ30(USER_MOTOR_Ls_q / L_max);

 EST_setLs_dq_pu(handle, &Ls_dq_pu);
 EST_setLs_qFmt(handle, Ls_qFmt);
```

Handle: The estimator (EST) handle

Ls_qFmt: The stator inductance Q format

EST_setMaxAccel_pu ()

extern void EST_setMaxAccel_pu(EST_Handle handle,const _iq maxAccel_pu);

Sets the maximum acceleration value in the estimator in per unit (pu), IQ24

Handle: The estimator (EST) handle

maxAccel_pu: The maximum acceleration value, pu

EST_setMaxAccel_est_pu ()

extern void EST_setMaxAccel_est_pu(EST_Handle handle,const _iq maxAccel_pu);

Sets the maximum estimation acceleration value in the estimator in per unit (pu), IQ24.

Handle: The estimator (EST) handle

Copyright © 2013–2019, Texas Instruments Incorporated

maxAccel_pu: The maximum estimation acceleration value, pu

EST_setMaxCurrentSlope_pu ()

void EST_setMaxCurrentSlope_pu (EST_Handle handle, const _iq maxCurrentSlope_pu)

Sets the maximum current slope value in the estimator in per unit (pu), IQ24.

Handle: The estimator (EST) handle

maxCurrentSlope_pu: The maximum current slope value, pu

EST_setMaxCurrentSlope_PowerWarp_pu ()

extern void EST_setMaxCurrentSlope_PowerWarp_pu(EST_Handle handle,const _iq maxCurrentSlope_pu);

Sets the maximum PowerWarp current slope value used in the estimator in per unit (pu), IQ24

Handle: The estimator (EST) handle

maxCurrentSlope_pu: The maximum current slope value, pu

EST_setRr_pu ()

extern void EST_setRr_pu(EST_Handle handle,const _iq Rr_pu);

Sets the rotor resistance value in the estimator in per unit (pu), IQ30.

Handle: The estimator (EST) handle

Rr_pu: The rotor resistance value, pu

EST_setRr_qFmt ()

extern void EST_setRr_qFmt(EST_Handle handle,uint_least8_t Rr_qFmt);

Sets the rotor resistance Q format in the estimator in 8 bit unsigned integer (uint_least8_t)

Handle: The estimator (EST) handle

Rr_qFmt: The rotor resistance Q format

EST_setRs_delta_pu ()

extern void EST_setRs_delta_pu(EST_Handle handle,const _iq Rs_delta_pu);

Sets the delta stator resistance value

Handle: The estimator (EST) handle

Rs_delta_pu: The delta stator resistance value, pu

EST_setRsOnLine_pu ()

extern void EST_setRsOnLine_pu(EST_Handle handle,const _iq Rs_pu);

Sets the stator resistance value in the online stator resistance estimator in per unit (pu), IQ30.

Handle: The estimator (EST) handle

Rs_pu: The stator resistance value, pu

Copyright © 2013–2019, Texas Instruments Incorporated

EST_setRsOnLine_qFmt ()

extern void EST_setRsOnLine_qFmt(EST_Handle handle,const uint_least8_t Rs_qFmt);

Sets the stator resistance Q format in the online stator resistance estimator in 8 bit unsigned integer (uint_least8_t)

Handle: The estimator (EST) handle

Rs_qFmt: The stator resistance Q format

EST_setRsOnLineFilterParams ()

extern void EST_setRsOnLineFilterParams(EST_Handle handle,const EST_RsOnLineFilterType_e filterType, const _iq filter_0_b0,const _iq filter_0_a1,const _iq filter_0_y1, const _iq filter_1_b0,const _iq filter_1_a1,const _iq filter_1_y1);

Sets the online stator resistance filter parameters in per unit (pu), IQ24

Handle: The estimator (EST) handle

filterType: The filter type

filter_0_b0: The filter 0 numerator coefficient value for z^0

filter_0_a1: The filter 0 denominator coefficient value for z^(-1)

filter_0_y1: The filter 0 output value at time sample n=-1

filter_1_b0: The filter 1 numerator coefficient value for z^0

filter_1_a1: The filter 1 denominator coefficient value for z^(-1)

filter_1_y1: The filter 1 output value at time sample n=-1

EST_setRsOnLineId_mag_pu ()

extern void EST_setRsOnLineId_mag_pu(EST_Handle handle,const _iq Id_mag_pu);

Sets the Id magnitude value used for online stator resistance estimation in per unit (pu), IQ24

Handle: The estimator (EST) handle

Id_mag_pu: The Id magnitude value, pu

EST_setRs_pu ()

extern void EST_setRs_pu(EST_Handle handle,const _iq Rs_pu);

Sets the stator resistance value used in the estimator in per unit (pu), IQ30

Handle: The estimator (EST) handle

Rs_pu: The stator resistance value, pu

EST_setRs_qFmt ()

extern void EST_setRs_qFmt(EST_Handle handle,uint_least8_t Rs_qFmt);

Sets the stator resistance Q format in the estimator in 8 bit unsigned integer (uint_least8_t)

Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The estimator (EST) handle

Rs_qFmt: The stator resistance Q format

EST_updateId_ref_pu ()

extern void EST_updateId_ref_pu(EST_Handle handle,_iq *pId_ref_pu);

Updates the Id reference value used for online stator resistance estimation in per unit (pu), IQ24.

Handle: The estimator (EST) handle

pId_ref_pu: The pointer to the Id reference value, pu

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.2.3 EST Get Functions

EST_get_krpm_to_pu_sf ()

extern _iq EST_get_krpm_to_pu_sf(EST_Handle handle);

Gets the krpm to pu scale factor in per unit (pu), IQ24. This function is needed when a user needs to scale a value of the motor speed from kpm (kilo revolutions per minute) to a per units value.

This scale factor is calculated and used as shown below:

```
 #define USER_MOTOR_NUM_POLE_PAIRS       (2)
#define USER_IQ_FULL_SCALE_FREQ_Hz      (500.0)

 _iq scale_factor = _IQ(USER_MOTOR_NUM_POLE_PAIRS * 1000.0 / (60.0 *
USER_IQ_FULL_SCALE_FREQ_Hz));

 _iq Speed_krpm = EST_getSpeed_krpm(handle);
 _iq Speed_krpm_to_pu_sf = EST_get_krpm_to_pu_sf(handle);
 _iq Speed_pu = _IQmpy(Speed_krpm,Speed_krpm_to_pu_sf);
```

Handle: The estimator (EST) handle

Return: The krpm to pu scale factor. This value is in IQ24

EST_get_pu_to_krpm_sf ()

extern _iq EST_get_pu_to_krpm_sf(EST_Handle handle);

Gets the pu to krpm scale factor in per unit (pu), IQ24. This function is needed when a user needs to scale a value of the motor speed from per units to krpm (kilo revolutions per minute) value.

This scale factor is calculated and used as shown below:

```
 #define USER_MOTOR_NUM_POLE_PAIRS       (2)
#define USER_IQ_FULL_SCALE_FREQ_Hz      (500.0)

 _iq scale_factor = IQ(60.0 * USER_IQ_FULL_SCALE_FREQ_Hz /
(USER_MOTOR_NUM_POLE_PAIRS * 1000.0));

 _iq Speed_pu = EST_getFm_pu(handle);
 _iq Speed_pu_to_krpm_sf = EST_get_pu_to_krpm_sf(handle);
 _iq Speed_krpm = _IQmpy(Speed_krpm,Speed_krpm_to_pu_sf);
```

Handle: The estimator (EST) handle

Return: The krpm to pu scale factor. This value is in IQ24

EST_getAngle_pu () *_iq EST_getAngle_pu(EST_Handle handle)*

Gets the angle value from the estimator in per unit (pu), IQ24. This function returns a per units value of the rotor flux angle. This value wraps around at 1.0, so the return value is between 0x00000000 or _IQ(0.0) to 0x00FFFFFF or _IQ(1.0). An example of using this angle is shown:

```
_iq Rotor_Flux_Angle_pu = EST_getAngle_pu(handle);
```

Handle: The estimator (EST) handle

Return: The flux angle value, pu

EST_getDcBus_pu () *_iq EST_getDcBus_pu(EST_Handle handle)*

Copyright © 2013–2019, Texas Instruments Incorporated

Gets the DC bus value from the estimator in per unit (pu), IQ24. This value is originally passed as a parameter when calling function EST_run(). A similar function can be simply reading what has been read and scaled by the ADC converter on pAdcData->dcBus. This value is used by the libraries internally to calculate one over dcbus, which is a value used to compensate the proportional gains of the current controllers. The following example shows how to use this function to calculate a DC bus value in kilo volts:

```
#define USER_IQ_FULL_SCALE_VOLTAGE_V (300.0)

 _iq Vbus_pu = EST_getDcBus_pu(handle);
 _iq Vbus_pu_to_kV_sf = _IQ(USER_IQ_FULL_SCALE_VOLT
_iq Vbus_kV = _IQmpy(Vbus_pu,Vbus_pu_to_kV_sf);
```

Handle: The estimator (EST) handle

Return: The DC bus value, pu

EST_ErrorCode_e EST_getErrorCode ()

extern EST_ErrorCode_e EST_getErrorCode(EST_Handle handle);

Gets the error code from the estimator

Handle: The estimator (EST) handle

Return: The error code

EST_getFe ()

extern int32_t EST_getFe(EST_Handle handle);

Gets the electrical frequency of the motor in Hertz (Hz). This frequency, in Hz, is the frequency of currents and voltages going into the motor. In order to get the speed of the motor, it is better to use EST_getFm().

Handle: The estimator (EST) handle

Return: The electrical frequency, Hz

EST_getFe_pu ()

extern _iq EST_getFe_pu(EST_Handle handle);

Gets the electrical frequency of the motor in per unit (pu), IQ24. Similar to EST_getFe() function, this function returns the electrical frequency of the motor in per units. In order to convert the electrical frequency from per units to Hz, the user needs to multiply the returned value by the following scale factor:

```
_iq Full_Scale_Freq_Elec_Hz = _IQ(USER_IQ_FULL_SCALE_FREQ_Hz);
_iq Freq_Elec_Hz = _IQmpy(EST_getFe_pu(handle),Full_Scale_Freq_Elec_Hz);
```

Handle: The estimator (EST) handle

Return: The electrical frequency, pu

EST_getFlag_enableForceAngle ()

extern bool EST_getFlag_enableForceAngle(EST_Handle handle);

Gets the enable force angle flag value from the estimator.

Handle: The estimator (EST) handle

Return: The value of the flag, in Boolean type, bool

- TRUE: Forced angle is enabled, and the estimated angle will be bypassed if the flux frequency falls below a threshold defined by:

```
#define USER_ZEROSPEEDLIMIT (0.001)
```


Copyright © 2013–2019, Texas Instruments Incorporated

A typical value of this frequency is 0.001 of the full scale frequency defined in:

```
#define USER_IQ_FULL_SCALE_FREQ_Hz (500.0)
```

Forced angle algorithm, when active, that is, when the rotor flux electrical frequency falls below the threshold, will be forcing a rotating angle at a frequency set by the following define:

```
#define USER_FORCE_ANGLE_FREQ_Hz  (1.0)
```

- FALSE: Disable forced angle. The estimator will never be bypassed by any forced angle algorithm.

EST_getFlag_enableRsOnLine ()

extern bool EST_getFlag_enableRsOnLine(EST_Handle handle);

Gets the value of the flag which enables online stator resistance (Rs) estimation

Handle: The estimator (EST) handle

Return: The enable online Rs flag value

- true Rs online recalibration algorithm is enabled. The estimator will run a set of functions related to rs online which recalculates the stator resistance while the motor is rotating. This algorithm is useful when motor heats up, and hence stator resistance increases.
- false Rs online recalibration algorithm is disabled, and no updates to Rs will be made even if the motor heats up. Low speed performace, and startup performance with full torque might be affected if stator resistance changes due to motor heating up. The stator resistance will be fixed, and equal to the value returned by: EST_getRs_Ohm().

EST_getFlag_enableRsRecalc ()

extern bool EST_getFlag_enableRsRecalc(EST_Handle handle);

Gets the value of the flag which enables online stator resistance (Rs) estimation

Handle: The estimator (EST) handle

Return: The enable online Rs flag value

- true Rs online recalibration algorithm is enabled. The estimator will run a set of functions related to rs online which recalculates the stator resistance while the motor is rotating. This algorithm is useful when motor heats up, and hence stator resistance increases.
- false Rs online recalibration algorithm is disabled, and no updates to Rs will be made even if the motor heats up. Low speed performace, and startup performance with full torque might be affected if stator resistance changes due to motor heating up. The stator resistance will be fixed, and equal to the value returned by: EST_getRs_Ohm().

EST_getFlag_estComplete ()

extern bool EST_getFlag_estComplete(EST_Handle handle);

Gets the value of the flag which denotes when the estimation is complete. This flag is set to true every time the EST_run() function is run. This flag can be reset to false by using the following example:

```
bool estComplete_Flag = EST_getFlag_estComplete(handle);
```

Handle: The estimator (EST) handle

Copyright © 2013–2019, Texas Instruments Incorporated

Return: The enable online Rs flag value

- true - The estimator has been run at least once since last time EST_setFlag_estComplete(handle, false) was called.
- false - The estimator has not been run since last time EST_setFlag_estComplete(handle, false) was called.

EST_getFlag_updateRs ()

extern bool EST_getFlag_updateRs(EST_Handle handle);

Gets the value of the flag which enables the updating of the stator resistance (Rs) value. When the online resistance estimator is enabled, the update flag allows the online resistance to be copied to the resistance used by the estimator model. If the update flag is not set to true the online resistance estimation will not be used by the estimator model, and if the resistance changes too much due to temperature increase, the model may not work as expected.

```
bool update_Flag = EST_getFlag_updateRs(handle);
```

Handle: The estimator (EST) handle

Return: The update Rs flag value

- true - The stator resistance estimated by the Rs OnLine module will be copied to the stator resistance used by the module, so of the motor's temperature changes, the estimated angle will be calculated based on the most up to date stator resistance
- false - The stator resistance estimated by the Rs OnLine module may or may not be updated depending on the enable flag, but will not be used in the motor's model used to generate the estimated speed and angle.

EST_getFlux_VpHz ()

int32_t EST_getFlux_VpHz(EST_Handle handle)

Gets the flux value in V/Hz

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. This functions returns a precise value only after the motor has been identified, which can be checked by the following code example:

```
if(EST_isMotorIdentified(handle))
  {
    // once the motor has been identified, get the flux
    float_t Flux_VpHz = EST_getFlux_VpHz(handle);
  }
```

Handle: The estimator (EST) handle

Return: The flux value, V/Hz

EST_getFlux_Wb ()

int32_t EST_getFlux_Wb(EST_Handle handle)

Gets the flux value in Weber

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 Webers, or Wb, or Volts * Seconds (V.s). This functions returns a precise value only after the motor has been identified, which can be checked by the following code example:

```
if(EST_isMotorIdentified(handle))
  {
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
    // once the motor has been identified, get the flux
    float_t Flux_Wb = EST_getFlux_Wb(handle);
  }
```

Handle: The estimator (EST) handle

Return: The flux value, Weber

EST_getFlux_pu () *extern _iq EST_getFlux_pu(EST_Handle handle);*

Gets the flux value in per unit (pu), IQ24

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 per units. This functions returns a precise value only after the motor has been identified, which can be checked by the following code example:

```
if(EST_isMotorIdentified(handle))
  {
    // once the motor has been identified, get the flux
    _iq Flux_pu = EST_getFlux_pu(handle);
  }
```

For some applications it is important to get this value in per units, since it is much faster to process especially when the architecture of the microcontroller does not have a floating point processing unit. In order to translate this per units value into a scaled value in _iq, it is important to consider a scale factor to convert this flux in per units to the required units. The following example shows how to scale a per units value to Wb and V/Hz in IQ for faster processing:

```
float_t FullScaleFlux = (USER_IQ_FULL_SCALE_VOLTAGE_V/(float_t)USER_EST_FREQ_Hz);
 float_t maxFlux =
(USER_MOTOR_RATED_FLUX*((USER_MOTOR_TYPE==MOTOR_Type_Induction)?0.05:0.7));
 float_t lShift = -ceil(log(FullScaleFlux/maxFlux)/log(2.0));
 _iq gFlux_pu_to_Wb_sf = _IQ(FullScaleFlux/(2.0*MATH_PI)*pow(2.0,lShift));
 _iq gFlux_pu_to_VpHz_sf = _IQ(FullScaleFlux*pow(2.0,lShift));
 // The value of gFlux_pu_to_Wb_sf and gFlux_pu_to_VpHz_sf can be calculated once
at the beginning of the
 // code and stored as global variables

 _iq Flux_Wb;
 _iq Flux_VpHz;
 _iq Flux_pu = EST_getFlux_pu(handle);

 Flux_Wb = _IQmpy(Flux_pu, gFlux_pu_to_Wb_sf);
 Flux_VpHz = _IQmpy(Flux_pu, gFlux_pu_to_VpHz_sf);
```

Handle: The estimator (EST) handle

Return: The flux value, pu

EST_getFm () *extern int32_t EST_getFm(EST_Handle handle);*

Gets the mechanical frequency of the motor in Hertz (Hz). This frequency, in Hz, is the mechanical frequency of the motor. If the motor is a permanent magnet motor, the mechanical frequency will be equal to the electrical frequency, since it is a synchronous motor. In the case of AC induction motors, the mechanical frequency will be equal to the electrical frequency minus the slip frequency. The following code example shows how to use this function to calculate revolutions per minute (RPM) in floating point:

```
#define USER_MOTOR_NUM_POLE_PAIRS  (2)

 float_t Mechanical_Freq_Hz = EST_getFm(handle);
 float_t hz_to_rpm_sf = 60.0/USER_MOTOR_NUM_POLE_PAIRS;
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
float_t Speed_RPM = Mechanical_Freq_Hz * hz_to_rpm_sf;
```

Handle: The estimator (EST) handle

Return: The mechanical frequency, Hz

EST_getFm_pu () *extern _iq EST_getFm_pu(EST_Handle handle);*

Gets the mechanical frequency of the motor in per unit (pu), IQ24. Similar to EST_getFe_pu() function, this function returns the mechanical frequency of the motor in per units. In order to convert the mechanical frequency from per units to kHz (to avoid saturation of IQ24), the user needs to multiply the returned value by the following scale factor:

```
#define USER_IQ_FULL_SCALE_FREQ_Hz  (500.0)

 _iq pu_to_khz_sf = _IQ(USER_IQ_FULL_SCALE_FREQ_Hz/1000.0);
 _iq khz_to_krpm_sf = _IQ(60.0/USER_MOTOR_NUM_POLE_PAIRS);
 _iq Mechanical_Freq_kHz = _IQmpy(EST_getFm_pu(handle),pu_to_khz_sf);
 _iq Speed_kRPM = _IQmpy(Mechanical_Freq_kHz,khz_to_krpm_sf);
```

Handle: The estimator (EST) handle

Return: The mechanical frequency, pu

EST_getForceAngleDelta_pu ()

extern _iq EST_getForceAngleDelta_pu(EST_Handle handle);

Gets the force angle delta value from the estimator in per unit (pu), IQ24. This function returns a valid value only after initializing the controller object by calling CTRL_setParams() function. The force angle delta represents the increments to be added to or subtracted from the forced angle. The higher this value is, the higher frequency will be generated when the angle is forced (estimated angle is bypassed when in forced angle mode). By default the forced angle frequency is set in user.h. The following example shows how to convert delta in per units to kilo Hertz (kHz).

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_EST_TICK  (1)
 #define USER_PWM_FREQ_kHz         (15.0)
 #define USER_ISR_FREQ_Hz          (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_EST_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_EST_TICK)

 _iq delta_pu_to_kHz_sf = _IQ((float_t)USER_EST_FREQ_Hz/1000.0);
 _iq Force_Angle_Delta_pu = EST_getForceAngleDelta_pu(handle);
 _iq Force_Angle_Freq_kHz = _IQmpy(Force_Angle_Delta_pu, delta_pu_to_kHz_sf);
```

Note that kHz is preferred to avoid overflow of IQ24 variables.

Handle: The estimator (EST) handle

Return: The force angle delta, pu. Minimum value of _IQ(0.0) and maximum of _IQ(1.0).

EST_getForceAngleStatus ()

extern bool EST_getForceAngleStatus(EST_Handle handle);

Gets the status of the force angle operation in the estimator. The status can only change to active when forced angle mode has been enabled by calling the following function: EST_setFlag_enableForceAngle(handle, true); Forced angle mode will be active when the electrical frequency of the motor falls below the defined threshold in user.h: #define USER_ZEROSPEEDLIMIT (0.001) details A manual check of forced angle status can be

Copyright © 2013–2019, Texas Instruments Incorporated

done using the following code example:

```
_iq fe_pu = EST_getFe_pu(handle);
 bool is_forced_angle_active;
 if(_IQabs(fe_pu) < _IQ(USER_ZEROSPEEDLIMIT))
   {
     is_forced_angle_active = true;
   }
 else
   {
     is_forced_angle_active = false;
   }
```

Note that kHz is preferred to avoid overflow of IQ24 variables.

Handle: The estimator (EST) handle

Return: A Boolean value denoting whether the angle has been forced (true) or not (false)

- true - The last iteration of the estimator used a forced angle to run the park and inverse park transforms. The estimator was also run in parallel to the forced angle, but the estimator output was not used.

\retval

- false - Forced angle mode is either disabled, or the electrical frequency did not fall below the predetermined threshold. The estimator output was used to run the park and inverse park transforms.

EST_getFreqB0_lp_pu ()

extern _iq EST_getFreqB0_lp_pu(EST_Handle handle);

Gets the low pass filter numerator value in the frequency estimator in per unit (pu), IQ30

Handle: The estimator (EST) handle

Return: The low pass filter numerator value, pu

EST_getFreqBeta_lp_pu ()

extern _iq EST_getFreqBeta_lp_pu(EST_Handle handle);

Gets the value used to set the pole location in the low-pass filter of the frequency estimator in per unit (pu), IQ30

Handle: The estimator (EST) handle

Return: The value used to set the filter pole location, pu

EST_getFslip ()

extern int32_t EST_getFslip(EST_Handle handle);

Gets the slip frequency of the motor in Hertz (Hz).

Handle: The estimator (EST) handle

Return: The slip frequency, Hz

Handle: The estimator (EST) handle

Return: The krpm to pu scale factor. This value is in IQ24

Copyright © 2013–2019, Texas Instruments Incorporated

EST_getFslip_pu () *extern _iq EST_getFslip_pu(EST_Handle handle);*

Gets the slip frequency of the motor in per unit (pu), IQ24

Similar to EST_getFe_pu() function, this function returns the slip frequency of the motor in per units. In order to convert the slip frequency from from per units to Hz, the user needs to multiply the returned value by the following scale factor:

```
#define USER_IQ_FULL_SCALE_FREQ_Hz (500.0)

 _iq Full_Scale_Freq_Elec_Hz = _IQ(USER_IQ_FULL_SCALE_FREQ_Hz);

 _iq Freq_Slip_Hz = _IQmpy(EST_getFslip_pu(handle),Full_Scale_Freq_Elec_Hz);
```

Handle: The estimator (EST) handle

Return: The slip frequency, pu

EST_getFullScaleCurrent ()

extern int32_t EST_getFullScaleCurrent(EST_Handle handle);

Gets the full scale current value used in the estimator in Amperes (A)

The value returned by this function is the same as the value defined in user.h. When users require to display a value in real world units; i.e., in Amperes, this value is used to convert the per unit values of currents into Amperes. The following example shows two different ways of doing this conversion, one using floating point, and the other one using IQ math. Example using floating point:

```
float_t pu_to_amps_sf = EST_getFullScaleCurrent(handle);
 _iq Id_rated_pu = EST_getIdRated_pu(handle);
 float_t Id_rated_A = _IQtoF(Id_rated_pu) * pu_to_amps_sf;

    Example using fixed point:

 #define USER_IQ_FULL_SCALE_CURRENT_A (10.0)

 _iq pu_to_amps_sf = _IQ(USER_IQ_FULL_SCALE_CURRENT_A);
 _iq Id_rated_pu = EST_getIdRated_pu(handle);
 _iq Id_rated_A = _IQmpy(Id_rated_pu, pu_to_amps_sf);
```

Handle: The estimator (EST) handle

Return: The full scale current value, A

EST_getFullScaleFlux ()

extern int32_t EST_getFullScaleFlux(EST_Handle handle);

Gets the full scale flux value used in the estimator in Volts per Hertz (V/Hz)

Handle: The estimator (EST) handle

Return: The full scale flux value

EST_getFullScaleFreq ()

extern int32_t EST_getFullScaleFreq(EST_Handle handle);

Gets the full scale frequency value used in the estimator in Hertz (Hz).

Full-scale frequency can be used as a scale factor to convert values from per units to Hertz. The following example shows how to use this function to convert frequency from per units to Hz using floating point math:

Copyright © 2013–2019, Texas Instruments Incorporated

```
float_t Mechanical_Frequency_pu = _IQtoF(EST_getFm_pu(handle));
 float_t pu_to_hz_sf = EST_getFullScaleFreq(handle);
 float_t Mechanical_Frequency_hz = Mechanical_Frequency_pu * pu_to_hz_sf
```

For faster execution, this function call can be avoided by using a definition of the full scale frequency that resides in user.h. The following example shows the same functionality but using fixed point math for faster execution:

```
#define USER_IQ_FULL_SCALE_FREQ_Hz (500.0)

 _iq Mechanical_Frequency_pu = EST_getFm_pu(handle);
 _iq pu_to_khz_sf = _IQ(USER_IQ_FULL_SCALE_FREQ_Hz/1000.0);
 _iq Mechanical_Frequency_khz = _IQmpy(Mechanical_Frequency_pu, pu_to_khz_sf);
```

Handle: The estimator (EST) handle

Return: The full scale frequency value, Hz

EST_getFullScaleInductance ()

extern int32_t EST_getFullScaleInductance(EST_Handle handle);

Gets the full scale inductance value used in the estimator in Henries (H).

There are different ways of getting the inductance used by the estimator. This function helps when converting an inductance from per units to H. However, the returned value is in floating point format, so utilizing this full scale value to convert per units to H is not the most efficient way. Two examples are provided below, showing a floating point per units to H conversion, and a fixed point per units to H conversion for faster execution. Floating point example:

```
uint_least8_t Ls_qFmt = EST_getLs_qFmt(handle);
 float_t fullScaleInductance = EST_getFullScaleInductance(handle);
 float_t Ls_d_pu = _IQ30toF(EST_getLs_d_pu(handle));
 float_t pu_to_h_sf = fullScaleInductance * pow(2.0, 30 - Ls_qFmt);
 float_t Ls_d_H = Ls_d_pu * pu_to_h_sf;
```

Another example is to avoid using floating point math for faster execution. In this example the full scale inductance value is calculated using pre-compiler math based on user's parameters in user.h:

```
#define VarShift(var,nshift) (((nshift) < 0) ? ((var)>>(-
(nshift))) : ((var) <<(nshift)))

 #define MATH_PI (3.14159265358979323846264338327 95)
 #define USER_IQ_FULL_SCALE_VOLTAGE_V (300.0)
 #define USER_IQ_FULL_SCALE_CURRENT_A (10.0)
 #define USER_VOLTAGE_FILTER_POLE_Hz  (335.648)
 #define USER_VOLTAGE_FILTER_POLE_rps (2.0 * MATH_PI *
USER_VOLTAGE_FILTER_POLE_Hz)

 uint_least8_t Ls_qFmt = EST_getLs_qFmt(handle);
 _iq fullScaleInductance =
_IQ(USER_IQ_FULL_SCALE_VOLTAGE_V/(USER_IQ_FULL_SCALE_CURRENT_A *
USER_VOLTAGE_FILTER_POLE_rps));
 _iq Ls_d_pu = _IQ30toIQ(EST_getLs_d_pu(handle));
 _iq pu_to_h_sf = VarShift(fullScaleInductance, 30 - Ls_qFmt);
 _iq Ls_d_H = _IQmpy(Ls_d_pu, pu_to_h_sf);
```

Handle: The estimator (EST) handle

Return: The full scale resistance value, Henry

EST_getFullScaleResistance ()

extern int32_t EST_getFullScaleResistance(EST_Handle handle);

Copyright © 2013–2019, Texas Instruments Incorporated

Gets the full scale resistance value used in the estimator in Ohms There are different ways of getting the resistance used by the estimator. This function helps when converting resistance from per units to Ohms. However, the returned value is in floating point format, so utilizing this full scale value to convert per units to Ohms is not the most efficient way. Two examples are provided below, showing a floating point per units to Ohms conversion, and a fixed point per units to Ohms conversion for faster execution. Floating point example:

```
uint_least8_t Rs_qFmt = EST_getRs_qFmt(handle);
 float_t fullScaleResistance = EST_getFullScaleResistance(handle);
 float_t Rs_pu = _IQ30toF(EST_getRs_pu(handle));
 float_t pu_to_ohms_sf = fullScaleResistance * pow(2.0, 30 - Rs_qFmt);
 float_t Rs_Ohms = Rs_pu * pu_to_ohms_sf;
```

Another example is to avoid using floating point math for faster execution. In this example the full scale resistance value is calculated using pre-compiler math based on user's parameters in user.h:

```
#define VarShift(var,nshift) (((nshift) < 0) ? ((var)>>(-
(nshift))) : ((var)<<(nshift)))

 #define USER_IQ_FULL_SCALE_VOLTAGE_V (300.0)
 #define USER_IQ_FULL_SCALE_CURRENT_A (10.0)

 uint_least8_t Rs_qFmt = EST_getRs_qFmt(handle);
 _iq fullScaleResistance =
_IQ(USER_IQ_FULL_SCALE_VOLTAGE_V/USER_IQ_FULL_SCALE_CURRENT_A);
 _iq Rs_pu = _IQ30toIQ(EST_getRs_pu(handle));
 _iq pu_to_ohms_sf = VarShift(fullScaleResistance, 30 - Rs_qFmt);
 _iq Rs_Ohms = _IQmpy(Rs_pu, pu_to_ohms_sf);
```

Handle: The estimator (EST) handle

Return: The full scale resistance value, Ohm

EST_getFullScaleVoltage ()

extern int32_t EST_getFullScaleVoltage(EST_Handle handle);

Gets the full-scale voltage value used in the estimator in Volts (V).

The value returned by this function is the same as the value defined in user.h. When users require to display a value in real world units; i.e., in Volts, this value is used to convert the per unit values of voltage into Volts. The following example shows two different ways of doing this conversion, one using floating point, and the other one using IQ math.

Example using floating point:

```
float_t pu_to_v_sf = EST_getFullScaleVoltage(handle);
 _iq DcBus_pu = EST_getDcBus_pu(handle);
 float_t DcBus_V = _IQtoF(DcBus_pu) * pu_to_v_sf;
```

Example using fixed point:

```
#define USER_IQ_FULL_SCALE_VOLTAGE_V (300.0)

 _iq pu_to_kv_sf = _IQ(USER_IQ_FULL_SCALE_VOLTAGE_V/1000.0);
 _iq DcBus_pu = EST_getDcBus_pu(handle);
 _iq DcBus_kV = _IQmpy(DcBus_pu, pu_to_kv_sf);
```

Handle: The estimator (EST) handle

Return: The full scale resistance value, Ohm

EST_getIdRated () *float_t EST_getIdRated(EST_Handle handle)*

Copyright © 2013–2019, Texas Instruments Incorporated

Gets the Id rated current value from the estimator

Handle: The estimator (EST) handle

Return: The Id rated current value, A

EST_getIdRated_pu ()

extern _iq EST_getIdRated_pu(EST_Handle handle);

Gets the Id rated current value from the estimator in per unit (pu), IQ24.

Handle: The estimator (EST) handle

Return: The Id rated current value, pu

EST_getIdRated_indEst_pu ()

_iq EST_getIdRated_indEst_pu(EST_Handle handle)

Gets the Id rated current value used for induction estimation

Handle: The estimator (EST) handle

Return: The Id rated value, pu

EST_getIdRated_ratedFlux_pu ()

extern _iq EST_getIdRated_ratedFlux_pu(EST_Handle handle);

Gets the Id current value used for flux estimation of induction motors in per unit (pu), IQ24.

Handle: The estimator (EST) handle

Return: The Id rated value, pu

EST_getLr_H ()

extern int32_t EST_getLr_H(EST_Handle handle);

Gets the rotor inductance value in Henries (H).

Handle: The estimator (EST) handle

Return: The Id rated value, pu

EST_getLr_pu ()

extern _iq EST_getLr_pu(EST_Handle handle);

Gets the rotor inductance value in per unit (pu), IQ30.

The per units value of the rotor inductance can be used as an alternative way of calculating the rotor inductance of an induction motor using fixed point math. An example showing how this is done is shown here:

```
#define VarShift(var,nshift) (((nshift) < 0) ? ((var)>>(-
(nshift))) : ((var)<<(nshift)))

 #define MATH_PI (3.14159265358979323846264338327795)
 #define USER_IQ_FULL_SCALE_VOLTAGE_V (300.0)
 #define USER_IQ_FULL_SCALE_CURRENT_A (10.0)
 #define USER_VOLTAGE_FILTER_POLE_Hz  (335.648)
 #define USER_VOLTAGE_FILTER_POLE_rps (2.0 * MATH_PI *
USER_VOLTAGE_FILTER_POLE_Hz)
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
 uint_least8_t Lr_qFmt = EST_getLr_qFmt(handle);
 _iq fullScaleInductance =
_IQ(USER_IQ_FULL_SCALE_VOLTAGE_V/(USER_IQ_FULL_SCALE_CURRENT_A *
USER_VOLTAGE_FILTER_POLE_rps));
 _iq Lr_pu = _IQ30toIQ(EST_getLr_pu(handle));
 _iq pu_to_h_sf = VarShift(fullScaleInductance, 30 - Lr_qFmt);
 _iq Lr_H = _IQmpy(Lr_pu, pu_to_h_sf);
```

Handle: The estimator (EST) handle

Return: The rotor inductance value, pu

EST_getLr_qFmt () *extern uint_least8_t EST_getLr_qFmt(EST_Handle handle);*

Gets the rotor inductance Q format in 8 bit unsigned integer (uint_least8_t).

When the motor is identified by the estimator, the Q format is used to have a wider range of the identified parameter. This Q format is the difference between the actual Q format used for the identification and IQ30 which is used internaly during identification of the motor parameters. To understand how this Q format can be used in user's code, please refer to the following example, which converts a per units value read from the estimator to Henries:

```
#define VarShift(var,nshift) (((nshift) < 0) ? ((var)>>(-
(nshift))) : ((var)<<(nshift)))

 #define MATH_PI (3.14159265358979323846264338327795)
 #define USER_IQ_FULL_SCALE_VOLTAGE_V (300.0)
 #define USER_IQ_FULL_SCALE_CURRENT_A (10.0)
 #define USER_VOLTAGE_FILTER_POLE_Hz  (335.648)
 #define USER_VOLTAGE_FILTER_POLE_rps (2.0 * MATH_PI *
USER_VOLTAGE_FILTER_POLE_Hz)

 uint_least8_t Lr_qFmt = EST_getLr_qFmt(handle);
 _iq fullScaleInductance =
_IQ(USER_IQ_FULL_SCALE_VOLTAGE_V/(USER_IQ_FULL_SCALE_CURRENT_A *
USER_VOLTAGE_FILTER_POLE_rps));
 _iq Lr_pu = _IQ30toIQ(EST_getLr_pu(handle));
 _iq pu_to_h_sf = VarShift(fullScaleInductance, 30 - Lr_qFmt);
 _iq Lr_H = _IQmpy(Lr_pu, pu_to_h_sf);
```

Handle: The estimator (EST) handle

Return: The rotor inductance value Q format

EST_getLs_d_H () *float_t EST_getLs_d_H(EST_Handle handle)*

Gets the direct stator inductance value in Henries

Handle: The estimator (EST) handle

Return: The direct stator inductance value, Henry

EST_getLs_d_pu () *extern _iq EST_getLs_d_pu(EST_Handle handle);*

Gets the direct stator inductance value in per unit (pu), IQ30

The per units value of the direct stator inductance can be used as an alternative way of calculating the direct stator inductance of a permanent magnet motor using fixed point math. An example showing how this is done is shown here:

```
#define VarShift(var,nshift) (((nshift) < 0) ? ((var)>>(-
(nshift))) : ((var)<<(nshift)))
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
 #define MATH_PI (3.14159265358979323846264338327795)
 #define USER_IQ_FULL_SCALE_VOLTAGE_V (300.0)
 #define USER_IQ_FULL_SCALE_CURRENT_A (10.0)
 #define USER_VOLTAGE_FILTER_POLE_Hz  (335.648)
 #define USER_VOLTAGE_FILTER_POLE_rps (2.0 * MATH_PI *
USER_VOLTAGE_FILTER_POLE_Hz)

 uint_least8_t Ls_qFmt = EST_getLs_qFmt(handle);
 _iq fullScaleInductance =
_IQ(USER_IQ_FULL_SCALE_VOLTAGE_V/(USER_IQ_FULL_SCALE_CURRENT_A *
USER_VOLTAGE_FILTER_POLE_rps));
 _iq Ls_d_pu = _IQ30toIQ(EST_getLs_d_pu(handle));
 _iq pu_to_h_sf = VarShift(fullScaleInductance, 30 - Ls_qFmt);
 _iq Ls_d_H = _IQmpy(Ls_d_pu, pu_to_h_sf);
```

Handle: The estimator (EST) handle

Return: The direct stator inductance value, pu

EST_getLs_delta_pu ()

extern _iq EST_getLs_delta_pu(EST_Handle handle);

Gets the delta stator inductance value in the stator inductance estimator

Handle: The estimator (EST) handle

Return: The delta stator inductance value, pu

EST_getLs_dq_pu () *extern void EST_getLs_dq_pu(EST_Handle handle,MATH_vec2 *pLs_dq_pu);*

Gets the direct/quadrature stator inductance vector values from the estimator in per unit (pu), IQ30 Both direct and quadrature stator inductances can be read from the estimator by using this function call and passing a pointer to a structure where these two values will be stored.

Handle: The estimator (EST) handle

pLs_dq_pu: The pointer for the direct/quadrature stator inductance vector values, pu

EST_getLs_q_H () *float_t EST_getLs_q_H(EST_Handle handle)*

Gets the stator inductance value in the quadrature coordinate direction in Henries

Handle: The estimator (EST) handle

Return: The stator inductance value, Henry

EST_getLs_q_pu () *extern _iq EST_getLs_q_pu(EST_Handle handle);*

Gets the stator inductance value in the quadrature coordinate direction in per unit (pu), IQ30

The per units value of the quadrature stator inductance can be used as an alternative way of calculating the quadrature stator inductance of a permanent magnet motor using fixed point math. An example showing how this is done is shown here:

```
#define VarShift(var,nshift) (((nshift) < 0) ? ((var)>>(-
(nshift))) : ((var)<<(nshift)))

 #define MATH_PI (3.14159265358979323846264338327795)
 #define USER_IQ_FULL_SCALE_VOLTAGE_V (300.0)
 #define USER_IQ_FULL_SCALE_CURRENT_A (10.0)
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
 #define USER_VOLTAGE_FILTER_POLE_Hz  (335.648)
 #define USER_VOLTAGE_FILTER_POLE_rps (2.0 * MATH_PI *
USER_VOLTAGE_FILTER_POLE_Hz)

 uint_least8_t Ls_qFmt = EST_getLs_qFmt(handle);
 _iq fullScaleInductance =
_IQ(USER_IQ_FULL_SCALE_VOLTAGE_V/(USER_IQ_FULL_SCALE_CURRENT_A *
USER_VOLTAGE_FILTER_POLE_rps));
 _iq Ls_q_pu = _IQ30toIQ(EST_getLs_q_pu(handle));
 _iq pu_to_h_sf = VarShift(fullScaleInductance, 30 - Ls_qFmt);
 _iq Ls_q_H = _IQmpy(Ls_q_pu, pu_to_h_sf);
```

Handle: The estimator (EST) handle

Return: The stator inductance value, pu

EST_getLs_qFmt () *extern uint_least8_t EST_getLs_qFmt(EST_Handle handle);*

Gets the stator inductance Q format in 8 bit unsigned integer (uint_least8_t).

When the motor is identified by the estimator, the Q format is used to have a wider range of the identified parameter. This Q format is the difference between the actual Q format used for the identification and IQ30 which is used internaly during identification of the motor parameters. To understand how this Q format can be used in user's code, please refer to the following example, which converts a per units value read from the estimator to Henries:

```
#define VarShift(var,nshift) (((nshift) < 0) ? ((var)>>(-
(nshift))) : ((var)<<(nshift)))

 #define MATH_PI (3.14159265358979323846264338327795)
 #define USER_IQ_FULL_SCALE_VOLTAGE_V (300.0)
 #define USER_IQ_FULL_SCALE_CURRENT_A (10.0)
 #define USER_VOLTAGE_FILTER_POLE_Hz  (335.648)
 #define USER_VOLTAGE_FILTER_POLE_rps (2.0 * MATH_PI *
USER_VOLTAGE_FILTER_POLE_Hz)

 uint_least8_t Ls_qFmt = EST_getLs_qFmt(handle);
 _iq fullScaleInductance =
_IQ(USER_IQ_FULL_SCALE_VOLTAGE_V/(USER_IQ_FULL_SCALE_CURRENT_A *
USER_VOLTAGE_FILTER_POLE_rps));
 _iq Ls_q_pu = _IQ30toIQ(EST_getLs_q_pu(handle));
 _iq pu_to_h_sf = VarShift(fullScaleInductance, 30 - Ls_qFmt);
 _iq Ls_q_H = _IQmpy(Ls_q_pu, pu_to_h_sf);
```

Handle: The estimator (EST) handle

Return: The stator inductance Q format

EST_getLs_max_pu ()

extern _iq EST_getLs_max_pu(EST_Handle handle);

Gets the maximum stator inductance value from the stator inductance estimator

Handle: The estimator (EST) handle

Return: The maximum stator inductance value, pu

EST_getLs_min_pu ()

extern _iq EST_getLs_min_pu(EST_Handle handle);

Gets the minimum stator inductance value from the stator inductance estimator

Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The estimator (EST) handle

Return: The minimum stator inductance value, pu

EST_getLs_coarse_max_pu ()

extern _iq EST_getLs_coarse_max_pu(EST_Handle handle);

Gets the maximum stator inductance value during coarse estimation in the stator inductance estimator

Handle: The estimator (EST) handle

Return: The maximum stator inductance value, pu

EST_getMaxAccel_pu ()

extern _iq EST_getMaxAccel_pu(EST_Handle handle);

Gets the maximum acceleration value used in the estimator in per unit (pu), IQ24

The maximum acceleration is a setting of the trajectory module, which sets the speed reference. The acceleration returned by this function call is used after the motor has been identified. This value represents how the speed reference is increased or decreased from an initial value to a target value. The following example shows how convert the returned value of this function to kilo Hertz per second (kHz/s) and kilo RPM per second (kRPM/s):

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK (10)
 #define USER_PWM_FREQ_kHz                 (15.0)
 #define USER_ISR_FREQ_Hz                  (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_TRAJ_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_TRAJ_TICK)
 #define USER_IQ_FULL_SCALE_FREQ_Hz        (500.0)
 #define USER_MOTOR_NUM_POLE_PAIRS         (4)

 _iq pu_to_khzps_sf = _IQ((float_t)USER_TRAJ_FREQ_Hz * USER_IQ_FULL_SCALE_FREQ_Hz
/ 1000.0);
 _iq khzps_to_krpmps_sf = _IQ(60.0 / (float_t)USER_MOTOR_NUM_POLE_PAIRS);

 _iq Accel_pu = EST_getMaxAccel_pu(handle);
 _iq Accel_kilo_hz_per_sec = _IQmpy(Accel_pu, pu_to_khzps_sf);
 _iq Accel_kilo_rpm_per_sec = _IQmpy(Accel_kilo_hz_per_sec, khzps_to_krpmps_sf);
```

The default value is set by a user's defined value in user.h, and the default value in per units is calculated internally as follows:

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK (10)
 #define USER_PWM_FREQ_kHz                 (15.0)
 #define USER_ISR_FREQ_Hz                  (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_TRAJ_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_TRAJ_TICK)
 #define USER_IQ_FULL_SCALE_FREQ_Hz        (500.0)
 #define USER_MAX_ACCEL_Hzps               (20.0)

 _iq hzps_to_pu_sf = _IQ(1.0 / ((float_t)USER_TRAJ_FREQ_Hz *
USER_IQ_FULL_SCALE_FREQ_Hz));

 _iq Accel_hertz_per_sec = _IQ(USER_MAX_ACCEL_Hzps);
 _iq Accel_pu = _IQmpy(Accel_hertz_per_sec, hzps_to_pu_sf);
```


Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The estimator (EST) handle

Return: The maximum acceleration value, pu

EST_getMaxAccel_est_pu ()

extern _iq EST_getMaxAccel_est_pu(EST_Handle handle);

Gets the maximum estimation acceleration value used in the estimator in per unit (pu), IQ24

The maximum acceleration is a setting of the trajectory module, which sets the speed reference. The acceleration returned by this function call is used during the motor identification process. This value represents how the speed reference is increased or decreased from an initial value to a target value. The following example shows how convert the returned value of this function to kilo Hertz per Second (kHz/s) and kilo RPM per second (kRPM/s):

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK (10)
 #define USER_PWM_FREQ_kHz                 (15.0)
 #define USER_ISR_FREQ_Hz                  (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_TRAJ_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_TRAJ_TICK)
 #define USER_IQ_FULL_SCALE_FREQ_Hz        (500.0)
 #define USER_MOTOR_NUM_POLE_PAIRS         (4)

 _iq pu_to_khzps_sf = _IQ((float_t)USER_TRAJ_FREQ_Hz * USER_IQ_FULL_SCALE_FREQ_Hz
/ 1000.0);
 _iq khzps_to_krpmps_sf = _IQ(60.0 / (float_t)USER_MOTOR_NUM_POLE_PAIRS);

 _iq est_Accel_pu = EST_getMaxAccel_est_pu(handle);
 _iq est_Accel_kilo_hz_per_sec = _IQmpy(est_Accel_pu, pu_to_khzps_sf);
 _iq est_Accel_kilo_rpm_per_sec = _IQmpy(est_Accel_kilo_hz_per_sec,
khzps_to_krpmps_sf);
```

The default value is set by a user's defined value in user.h, and the default value in per units is calculated internally as follows:

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK (10)
 #define USER_PWM_FREQ_kHz                 (15.0)
 #define USER_ISR_FREQ_Hz                  (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_TRAJ_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_TRAJ_TICK)
 #define USER_IQ_FULL_SCALE_FREQ_Hz        (500.0)
 #define USER_MAX_ACCEL_EST_Hzps           (2.0)

 _iq hzps_to_pu_sf = _IQ(1.0 / ((float_t)USER_TRAJ_FREQ_Hz *
USER_IQ_FULL_SCALE_FREQ_Hz));

 _iq est_Accel_hertz_per_sec = _IQ(USER_MAX_ACCEL_EST_Hzps);
 _iq est_Accel_pu = _IQmpy(est_Accel_hertz_per_sec, hzps_to_pu_sf);
```

Handle: The estimator (EST) handle

Return: The maximum estimation acceleration value, pu

EST_getMaxCurrentSlope_pu ()

extern _iq EST_getMaxCurrentSlope_pu(EST_Handle handle);

Copyright © 2013–2019, Texas Instruments Incorporated

Gets the maximum current slope value used in the estimator in per unit (pu), IQ24

Gets the slope of Id reference. The following example shows how to convert the returned value into kilo Amperes per second (kA/s):

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK (10)
 #define USER_PWM_FREQ_kHz                 (15.0)
 #define USER_ISR_FREQ_Hz                  (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_TRAJ_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_TRAJ_TICK)
 #define USER_IQ_FULL_SCALE_CURRENT_A      (10.0)

 _iq pu_to_kA_per_sec_sf = _IQ((float_t)USER_TRAJ_FREQ_Hz *
USER_IQ_FULL_SCALE_CURRENT_A / 1000.0);

 _iq currentSlope_pu = EST_getMaxCurrentSlope_pu(handle);
 _iq currentSlope_kAps = _IQmpy(currentSlope_pu, pu_to_kA_per_sec_sf);
```

The default value is set by a user's defined value in user.h, and the default value in per units is calculated internally as follows:

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK (10)
 #define USER_PWM_FREQ_kHz                 (15.0)
 #define USER_ISR_FREQ_Hz                  (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_TRAJ_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_TRAJ_TICK)
 #define USER_IQ_FULL_SCALE_CURRENT_A      (10.0)
 #define USER_MOTOR_RES_EST_CURRENT        (1.0)

 _iq A_per_sec_to_pu_sf = _IQ(1.0 / ((float_t)USER_TRAJ_FREQ_Hz *
USER_IQ_FULL_SCALE_CURRENT_A));

 _iq currentSlope_Aps = _IQ(USER_MOTOR_RES_EST_CURRENT);
 _iq currentSlope_pu = _IQmpy(currentSlope_Aps, A_per_sec_to_pu_sf);
```

Handle: The estimator (EST) handle

Return: The maximum current slope value, pu

EST_getMaxCurrentSlope_PowerWarp_pu ()

extern _iq EST_getMaxCurrentSlope_PowerWarp_pu(EST_Handle handle);

Gets the maximum PowerWarp current slope value used in the estimator in per unit (pu), IQ24

Gets the slope of Id reference change when efficient partial load is enabled. This mode only applies to induction motors. The following example shows how to convert the returned value into kilo Amperes per second (kA/s):

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK (10)
 #define USER_PWM_FREQ_kHz                 (15.0)
 #define USER_ISR_FREQ_Hz                  (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_TRAJ_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_TRAJ_TICK)
 #define USER_IQ_FULL_SCALE_CURRENT_A      (10.0)

 _iq pu_to_kA_per_sec_sf = _IQ((float_t)USER_TRAJ_FREQ_Hz *
USER_IQ_FULL_SCALE_CURRENT_A / 1000.0);
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
 _iq currentSlope_PowerWarp_pu = EST_getMaxCurrentSlope_PowerWarp_pu(handle);
 _iq currentSlope_PowerWarp_kAps = _IQmpy(currentSlope_PowerWarp_pu,
pu_to_kA_per_sec_sf);
```

The default value is set by a user's defined value in user.h, and the default value in per units is calculated internally as follows:

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK  (1)
 #define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK (10)
 #define USER_PWM_FREQ_kHz                 (15.0)
 #define USER_ISR_FREQ_Hz                  (USER_PWM_FREQ_kHz * 1000.0)
 #define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)
 #define USER_TRAJ_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_TRAJ_TICK)
 #define USER_IQ_FULL_SCALE_CURRENT_A      (10.0)
 #define USER_MOTOR_RES_EST_CURRENT        (1.0)

 _iq A_per_sec_to_pu_sf = _IQ(1.0 / ((float_t)USER_TRAJ_FREQ_Hz *
USER_IQ_FULL_SCALE_CURRENT_A));

 _iq currentSlope_PowerWarp_Aps = _IQ(0.3 * USER_MOTOR_RES_EST_CURRENT);
 _iq currentSlope_PowerWarp_pu = _IQmpy(currentSlope_PowerWarp_Aps,
A_per_sec_to_pu_sf);
```

Handle: The estimator (EST) handle

Return: The maximum PowerWarp current slope value, pu

EST_getOneOverDcBus_pu ()

extern _iq EST_getOneOverDcBus_pu(EST_Handle handle);

Gets the inverse of the DC bus voltage in per unit (pu), IQ24

Handle: The estimator (EST) handle

Return: The inverse of the DC bus voltage, pu

EST_getRr_Ohm ()

extern int32_t EST_getRr_Ohm(EST_Handle handle);

Gets the rotor resistance value in Ohms

Handle: The estimator (EST) handle

Return: The rotor resistance value, Ohm

EST_getRr_pu ()

extern _iq EST_getRr_pu(EST_Handle handle);

Gets the rotor resistance value in per unit (pu), IQ30

Handle: The estimator (EST) handle

Return: The rotor resistance value, pu

EST_getRr_qFmt ()

extern uint_least8_t EST_getRr_qFmt(EST_Handle handle);

Gets the rotor resistance Q format in 8 bit unsigned integer (uint_least8_t).

Handle: The estimator (EST) handle

Return: The rotor resistance Q format

Copyright © 2013–2019, Texas Instruments Incorporated

EST_getRs_delta_pu ()

extern _iq EST_getRs_delta_pu(EST_Handle handle);

Gets the delta stator resistance value from the stator resistance estimator

Handle: The estimator (EST) handle

Return: The delta stator resistance value, pu

EST_getRs_Ohm () *float_t EST_getRs_Ohm(EST_Handle handle)*

Gets the stator resistance value that is used by the angle estimator

Handle: The estimator (EST) handle

Return: The stator resistance value, Ohm

EST_getRs_pu () *extern _iq EST_getRs_pu(EST_Handle handle);*

Gets the stator resistance value in per unit (pu), IQ30

Handle: The estimator (EST) handle

Return: The stator resistance value, pu

EST_getRs_qFmt () *extern uint_least8_t EST_getRs_qFmt(EST_Handle handle);*

Gets the stator resistance Q format in 8 bit unsigned integer (uint_least8_t)

Handle: The estimator (EST) handle

Return: The stator resistance Q format

EST_getRs_qFmt () *extern void EST_getRsOnLineFilterParams(EST_Handle handle,const EST_RsOnLineFilterType_e filterType, _iq *pFilter_0_b0,_iq *pFilter_0_a1,_iq *pFilter_0_y1, _iq *pFilter_1_b0,_iq *pFilter_1_a1,_iq *pFilter_1_y1);*

Gets the online stator resistance filter parameters in per unit (pu), IQ24

Handle: The estimator (EST) handle

filterType: The filter type

pFilter_0_b0: The pointer for the filter 0 numerator coefficient value for z^0

pFilter_0_a1: The pointer for the filter 0 denominator coefficient value for z^(-1)

pFilter_0_y1: The pointer for the filter 0 output value at time sample n=-1

pFilter_1_b0: The pointer for the filter 1 numerator coefficient value for z^0

pFilter_1_a1: The pointer for the filter 1 denominator coefficient value for z^(-1)

pFilter_1_y1: The pointer for the filter 1 output value at time sample n=-1

EST_getRsOnLine_Ohm ()

extern int32_t EST_getRsOnLine_Ohm(EST_Handle handle)

Gets the online stator resistance value

Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The estimator (EST) handle

Return: The online stator resistance value, Ohm

EST_getRsOnLine_pu ()

extern _iq EST_getRsOnLine_pu(EST_Handle handle);

Gets the online stator resistance value in per unit (pu), IQ30

Handle: The estimator (EST) handle

Return: The online stator resistance Q format

EST_getRsOnLineId_mag_pu ()

extern _iq EST_getRsOnLineId_mag_pu(EST_Handle handle);

Gets the Id magnitude value used for online stator resistance estimation in per unit (pu), IQ24

Handle: The estimator (EST) handle

Return: The Id magnitude value, pu

EST_getRsOnLineId_pu ()

extern _iq EST_getRsOnLineId_pu(EST_Handle handle);

Gets the Id value used for online stator resistance estimation in per unit (pu), IQ24

Handle: The estimator (EST) handle

Return: The Id value, pu

EST_getSpeed_krpm ()

_iq EST_getSpeed_krpm(EST_Handle handle)

Gets the speed value in kilo-rpm

Handle: The estimator (EST) handle

Return: The speed value, kilo-rpm

EST_getSignOfDirection ()

extern int_least8_t EST_getSignOfDirection(EST_Handle handle);

Gets the sign of the direction value in 8 bit signed integer (int_least8_t)

Handle: The estimator (EST) handle

Return: The sign of the direction value (-1 for negative, 1 for positive)

EST_getSpeed_krpm ()

_iq EST_getSpeed_krpm(EST_Handle handle)

Gets the speed value in per unit (pu), IQ24

Handle: The estimator (EST) handle

Return: The speed value, kilo-rpm

Copyright © 2013–2019, Texas Instruments Incorporated

EST_getState ()

EST_State_e EST_getState(EST_Handle handle)

Gets the state of the estimator

Handle: The estimator (EST) handle

Return: The estimator state

EST_getTorque_lbin ()

_iq EST_getTorque_lbin(EST_Handle handle)

Gets the torque value in per unit (pu), IQ24

Handle: The estimator (EST) handle

Return: The torque value, lb*in

EST_getTorque_Nm ()

extern _iq EST_getTorque_Nm(EST_Handle handle);

Gets the torque value in per unit (pu), IQ24

Handle: The estimator (EST) handle

Return: The torque value, N*m

EST_getDir_qFmt ()

extern uint_least8_t EST_getDir_qFmt(EST_Handle handle);

Gets the direction Q format from the estimator

Handle: The estimator (EST) handle

Return: The direction Q format

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.2.4 EST Run and Compute Functions

EST_computeLr_H ()

extern int32_t EST_computeLr_H(EST_Handle handle,const _iq current);

Computes the rotor inductance in Henries (H)

Handle: The estimator (EST) handle

Current: The current in the rotor

Return: The rotor inductance, H

EST_doCurrentCtrl ()

extern bool EST_doCurrentCtrl(EST_Handle handle);

Determines if current control should be performed during motor identification

Handle: The estimator (EST) handle

Return: A Boolean value denoting whether (true) or not (false) to perform current control

EST_genOutputLimits_Pid_Id ()

extern void EST_genOutputLimits_Pid_Id(EST_Handle handle, const _iq maxDutyCycle, _iq *outMin,_iq *outMax);

Generated the PID Id controller output limits

Handle: The estimator (EST) handle

maxDutyCycle: The maximum duty cycle, pu

outMin: The pointer to the minimum output value

outMax: The pointer to the maximum output value

EST_genOutputLimits_Pid_Iq ()

extern void EST_genOutputLimits_Pid_Iq(EST_Handle handle, const _iq maxDutyCycle, const _iq out_Id,_iq *outMin,_iq *outMax);

Generated the PID Iq controller output limits

Handle: The estimator (EST) handle

maxDutyCycle: The maximum duty cycle, pu

out_Id: The Id output value

outMin: The pointer to the minimum output value

outMax: The pointer to the maximum output value

EST_run ()

extern void EST_run(EST_Handle handle, const MATH_vec2 *plab_pu, const MATH_vec2 *pVab_pu, const _iq dcBus_pu, const _iq speed_ref_pu);

Runs the estimator

Handle: The estimator (EST) handle

Copyright © 2013–2019, Texas Instruments Incorporated

plab_pu: The pointer to the phase currents in the alpha/beta coordinate system, pu

IQ24 pVab_pu: The pointer to the phase voltages in the alpha/beta coordinate system, pu

IQ24 dcBus_pu: The DC bus voltage, pu

IQ24 speed_ref_pu: The speed reference value to the controller, pu IQ24

EST_computeDirection_qFmt ()

extern uint_least8_t EST_computeDirection_qFmt(EST_Handle handle,const int32_t flux_max);

Computes the direction Q format for the estimator

Handle: The estimator (EST) handle

flux_max: The maximum flux value

Return: The direction Q format

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.2.5 EST Counter Functions

EST_resetCounter_ctrl ()

extern void EST_resetCounter_ctrl(EST_Handle handle);

Resets the control counter

Handle: The estimator (EST) handle

EST_resetCounter_state ()

extern void EST_resetCounter_state(EST_Handle handle);

Resets the state counter

Handle: The estimator (EST) handle

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.2.6 EST State Control and Error Handling Functions

EST_isError () ***extern bool EST_isError(EST_Handle handle);***

Determines if there is an estimator error

Handle: The estimator (EST) handle

Return: A Boolean value denoting if there is an estimator error (true) or not (false)

EST_isIdle () ***extern bool EST_isIdle(EST_Handle handle);***

Determines if the estimator is idle

Handle: The estimator (EST) handle

Return: A Boolean value denoting if the estimator is idle (true) or not (false)

EST_isLockRotor () ***extern bool EST_isLockRotor(EST_Handle handle);***

Determines if the estimator is waiting for the rotor to be locked

Handle: The estimator (EST) handle

Return: A Boolean value denoting if the estimator is waiting for the rotor to be locked (true) or not (false)

EST_isMotorIdentified ()

EST_State_e EST_isMotorIdentified (EST_Handle handle)

Determines if the motor has been identified

Handle: The estimator (EST) handle

Return: The estimator state

EST_isOnLine () ***extern bool EST_isOnLine(EST_Handle handle);***

Determines if the estimator is ready for online control

Handle: The estimator (EST) handle

Return: A Boolean value denoting if the estimator is ready for online control (true) or not (false)

EST_updateState () ***extern bool EST_updateState(EST_Handle handle,const _iq Id_target_pu);***

Updates the estimator state

Handle: The estimator (EST) handle

Id_target_pu: The target Id current during each estimator state, pu IQ24

Return: A Boolean value denoting if the state has changed (true) or not (false)

EST_useZeroIq_ref ()

extern bool EST_useZeroIq_ref(EST_Handle handle);

Determines if a zero Iq current reference should be used in the controller

Copyright © 2013–2019, Texas Instruments Incorporated

Handle: The estimator (EST) handle

Return: A Boolean value denoting if a zero Iq current reference should be used (true) or not (false)

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.3 Hardware Abstraction Layer (HAL) API Functions – hal.c, hal.h, hal_obj.h

The HAL_Obj is a structure that contains the handles to the peripherals on the device. HAL_init() allocates memory for HAL object, and once the memory is allocated, HAL_setParams() configures the each peripheral according to the user settings in the object USER_Params.

3.3.3.1 HAL Enumerations and Structures

HAL_AdcData_t

Defines the ADC data This data structure contains the voltage and current values that are used when performing a HAL_AdcRead and then this structure is passed to the CTRL controller and the FAST estimator.

```
typedef struct _HAL_AdcData_t_
{
  MATH_vec3 I;          //!< the current values

  MATH_vec3 V;          //!< the voltage values

  _iq       dcBus;      //!< the dcBus value

} HAL_AdcData_t;
```

HAL_DacData_t

Defines the DAC data This data structure contains the pwm values that are used for the DAC output on a lot of the hardware kits for debugging.

```
typedef struct _HAL_DacData_t_
{
  _iq  value[4];      //!< the DAC data

} HAL_DacData_t;
```

HAL_PwmData_t

Defines the PWM data This structure contains the pwm voltage values for the three phases. A HAL_PwmData_t variable is filled with values from, for example, a space vector modulator and then sent to functions like HAL_writePwmData() to write to the PWM peripheral.

```
typedef struct _HAL_PwmData_t_
{
  MATH_vec3  Tabc;      //!< the PWM time-durations for each motor phase

} HAL_PwmData_t;
```

HAL_LedNumber_e

Enumeration to define the LEDs on ControlCARD

```
typedef enum
{
  HAL_Gpio_LED2=GPIO_Number_31,  //!< GPIO pin number for ControlCARD LED 2
  HAL_Gpio_LED3=GPIO_Number_34   //!< GPIO pin number for ControlCARD LED 3
} HAL_LedNumber_e;
```

GPIO_Number_e

Enumeration to define the general purpose I/O (GPIO) numbers

```
typedef enum
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
{
  GPIO_Number_0=0,    //!< Denotes GPIO number 0
  GPIO_Number_1,      //!< Denotes GPIO number 1
  GPIO_Number_2,      //!< Denotes GPIO number 2
...
  GPIO_Number_57,     //!< Denotes GPIO number 57
  GPIO_Number_58,     //!< Denotes GPIO number 58
  GPIO_numGpios
} GPIO_Number_e;
```

HAL_SensorType_e

```
typedef enum
{
  HAL_SensorType_Current=0,      //!< Enumeration for current sensor
  HAL_SensorType_Voltage         //!< Enumeration for voltage sensor
} HAL_SensorType_e;
```

HAL_Obj

The HAL object contains all handles to peripherals. When accessing a peripheral on a processor, use a HAL function along with the HAL handle for that processor to access its peripherals.

```
typedef struct _HAL_Obj_
{
  ADC_Handle    adcHandle;        //!< the ADC handle

  CLK_Handle    clkHandle;        //!< the clock handle

  CPU_Handle    cpuHandle;        //!< the CPU handle

  FLASH_Handle  flashHandle;      //!< the flash handle

  GPIO_Handle   gpioHandle;       //!< the GPIO handle

  OFFSET_Handle offsetHandle_I[3];  //!< the handles for the current offset
estimators
  OFFSET_Obj    offset_I[3];        //!< the current offset objects

  OFFSET_Handle offsetHandle_V[3];  //!< the handles for the voltage offset
estimators
  OFFSET_Obj    offset_V[3];        //!< the voltage offset objects

  OSC_Handle    oscHandle;        //!< the oscillator handle

  PIE_Handle    pieHandle;        //<! the PIE handle

  PLL_Handle    pllHandle;        //!< the PLL handle

  PWM_Handle    pwmHandle[3];     //<! the PWM handles

  PWMDAC_Handle pwmDacHandle[3];  //<! the PWMDAC handles

  PWR_Handle    pwrHandle;        //<! the power handle

  TIMER_Handle  timerHandle[3];   //<! the timer handles

  WDOG_Handle   wdogHandle;       //!< the watchdog handle

  HAL_AdcData_t adcBias;          //!< the ADC bias

  _iq           current_sf;       //!< the current scale factor, amps_pu/cnt

  _iq           voltage_sf;       //!< the voltage scale factor, volts_pu/cnt
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
  uint_least8_t numCurrentSensors; //!< the number of current sensors
  uint_least8_t numVoltageSensors; //!< the number of voltage sensors

  AFE_Handle    afeHandle;        //!< the AFE handle

#ifdef QEP
  QEP_Handle    qepHandle[1];      //!< the QEP handle
#endif

} HAL_Obj;
```


Copyright © 2013–2019, Texas Instruments Incorporated

3.3.3.2 HAL – ADC and AFE

HAL_setupAdcs () ***void HAL_setupAdcs(HAL_Handle handle)***

Sets up the ADCs (Analog to Digital Converters)

Handle: The driver (HAL) handle

HAL_setupAfe () ***void HAL_setupAfe(HAL_Handle halHandle)***

Sets up the AFE (Analog Front End)

Handle: The driver (HAL) handle

HAL_acqAdcInt () ***void HAL_acqAdcInt(HAL_Handle handle,const ADC_IntNumber_e intNumber)***

Acknowledges an interrupt from the ADC

Handle: The driver (HAL) handle

intNumber: The interrupt number

HAL_readAdcData() ***void HAL_readAdcData(HAL_Handle handle,HAL_AdcData_t *pAdcData)***

Reads the ADC data into the values pointed to by pAdcData

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.

Handle: The driver (HAL) handle

pAdcData: A pointer to the ADC data buffer

HAL_updateAdcBias ()

static inline void HAL_updateAdcBias(HAL_Handle handle)

Updates the ADC bias values This function is called before the motor is started. It sets the voltage and current measurement offsets.

Handle: The driver (HAL) handle

HAL_setBias () ***void HAL_setBias (HAL_Handle handle,const HAL_SensorType_e sensorType,uint_least8_t sensorNumber,const _iq bias)***

Sets the ADC bias value

Handle: The driver (HAL) handle

sensorType: The sensor type

sensorNumber: The sensor number

bias: The ADC bias value

HAL_getBias () ***void HAL_getBias (HAL_Handle handle,const HAL_SensorType_e sensorType,uint_least8_t sensorNumber)***

Gets the ADC bias value

Copyright © 2013–2019, Texas Instruments Incorporated

The ADC bias contains the feedback circuit's offset and bias. Bias is the mathematical offset used when a bi-polar signal is read into a uni-polar ADC.

Handle: The driver (HAL) handle

sensorType: The sensor type

sensorNumber: The sensor number

Return: The ADC bias value

HAL_cal ()

extern void HAL_cal(HAL_Handle handle);

Executes calibration routines

Values for offset and gain are programmed into OTP memory by TI factory. This calls and internal function that programs these offsets and gains into the ADC registers.

Handle: The driver (HAL) handle

HAL_AdcCalConversion ()

uint16_t HAL_AdcCalConversion(HAL_Handle handle);

Reads the converted value from the selected calibration channel

Handle: The driver (HAL) handle

Return: The converted value

HAL_AdcOffsetSelfCal ()

void HAL_AdcOffsetSelfCal(HAL_Handle handle);

Executes the offset calibration of the ADC

Handle: The driver (HAL) handle

HAL_getAdcSocSampleDelay ()

static inline ADC_SocSampleDelay_e HAL_getAdcSocSampleDelay(HAL_Handle handle, const ADC_SocNumber_e socNumber)

Gets the ADC delay value

Handle: The driver (HAL) handle

socNumber: The ADC SOC number

Return: The ADC delay value

HAL_setAdcSocSampleDelay ()

static inline ADC_SocSampleDelay_e HAL_setAdcSocSampleDelay(HAL_Handle handle, const ADC_SocNumber_e socNumber)

Sets the ADC delay value

Handle: The driver (HAL) handle

socNumber: The ADC SOC number

sampleDelay: The ADC delay value

Copyright © 2013–2019, Texas Instruments Incorporated

HAL_getCurrentScaleFactor ()

static inline _iq HAL_getCurrentScaleFactor(HAL_Handle handle)

Gets the current scale factor

The current scale factor is defined as USER_ADC_FULL_SCALE_CURRENT_A/USER_IQ_FULL_SCALE_CURRENT_A. This scale factor is not used when converting between PU amps and real amps.

Handle: The driver (HAL) handle

Return: The current scale factor

HAL_setCurrentScaleFactor () *static inline _iq HAL_setCurrentScaleFactor(HAL_Handle handle)*

Sets the current scale factor

The current scale factor is defined as USER_ADC_FULL_SCALE_CURRENT_A/USER_IQ_FULL_SCALE_CURRENT_A. This scale factor is not used when converting between PU amps and real amps.

Handle: The driver (HAL) handle

current_sf: The current scale factor

HAL_getVoltageScaleFactor ()

static inline _iq HAL_getVoltageScaleFactor(HAL_Handle handle)

Gets the voltage scale factor

The voltage scale factor is defined as USER_ADC_FULL_SCALE_VOLTAGE_V/USER_IQ_FULL_SCALE_VOLTAGE_V. This scale factor is not used when converting between PU volts and real volts.

Handle: The driver (HAL) handle

Return: The voltage scale factor

HAL_setVoltageScaleFactor ()

static inline _iq HAL_setVoltageScaleFactor(HAL_Handle handle)

Sets the voltage scale factor

Handle: The driver (HAL) handle

voltage_sf: The voltage scale factor

HAL_getNumCurrentSensors ()

static inline uint_least8_t HAL_getNumCurrentSensors(HAL_Handle handle)

Gets the number of current sensors

Handle: The driver (HAL) handle

Return: The number of current sensors

HAL_setNumCurrentSensors ()

static inline uint_least8_t HAL_setNumCurrentSensors(HAL_Handle handle)

Copyright © 2013–2019, Texas Instruments Incorporated

Sets the number of current sensors

Handle: The driver (HAL) handle

Return: The number of current sensors

HAL_getNumVoltageSensors ()

static inline uint_least8_t HAL_getNumVoltageSensors(HAL_Handle handle)

Gets the number of voltage sensors

Handle: The driver (HAL) handle

Return: The number of voltage sensors

HAL_setNumVoltageSensors ()

static inline uint_least8_t HAL_setNumVoltageSensors(HAL_Handle handle)

Sets the number of voltage sensors

Handle: The driver (HAL) handle

numVoltageSensors: The number of voltage sensors

Handle: The driver (HAL) handle

Return:

HAL_getOffsetBeta_lp_pu ()

static inline _iq HAL_getOffsetBeta_lp_pu(HAL_Handle handle, const HAL_SensorType_e sensorType, const uint_least8_t sensorNumber)

Gets the value used to set the low pass filter pole for offset estimation

An IIR single pole low pass filter is used to find the feedback circuit's offsets. This function returns the value of that pole.

Handle: The driver (HAL) handle

sensorType: The sensor type

sensorNumber: The sensor number

Return: The value used to set the low pass filter pole, pu

HAL_setOffsetBeta_lp_pu ()

static inline _iq HAL_setOffsetBeta_lp_pu(HAL_Handle handle, const HAL_SensorType_e sensorType, const uint_least8_t sensorNumber)

Sets the value used to set the low pass filter pole for offset estimation

An IIR single pole low pass filter is used to find the feedback circuit's offsets. This function returns the value of that pole.

Handle: The driver (HAL) handle

sensorType: The sensor type

Copyright © 2013–2019, Texas Instruments Incorporated

sensorNumber: The sensor number

beta_lp_pu: The value used to set the low pass filter pole, pu

HAL_setOffsetInitCond ()

static inline void HAL_setOffsetInitCond(HAL_Handle handle, const HAL_SensorType_e sensorType, const uint_least8_t sensorNumber, const _iq initCond)

Sets the offset initial condition value for offset estimation

Handle: The driver (HAL) handle

sensorType: The sensor type

sensorNumber: The sensor number

initCond: The initial condition value

HAL_getOffsetValue ()

static inline _iq HAL_getOffsetValue(HAL_Handle handle, const HAL_SensorType_e sensorType, const uint_least8_t sensorNumber)

Gets the offset value

The offsets that are calculated during the feedback circuits calibrations are returned from the IIR filter object.

Handle: The driver (HAL) handle

sensorType: The sensor type

sensorNumber: The sensor number

Return: The offset value

HAL_setOffsetValue ()

static inline _iq HAL_setOffsetValue(HAL_Handle handle, const HAL_SensorType_e sensorType, const uint_least8_t sensorNumber)

Sets the offset value

Handle: The driver (HAL) handle

sensorType: The sensor type

sensorNumber: The sensor number

Value: The initial offset value

HAL_runOffsetEst () *inline void HAL_runOffsetEst(HAL_Handle handle,const HAL_AdcData_t *pAdcData)*

Runs offset estimation

Handle: The driver (HAL) handle

pAdcData: The pointer to the ADC data

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.3.3 HAL – PWM and PWM-DAC

HAL_setupPwms () ***extern void HAL_setupPwms(HAL_Handle handle, const uint_least16_t systemFreq_MHz, const float_t pwmPeriod_usec, const uint_least16_t numPwmTicksPerIsrTick);***

Sets up the PWMs (Pulse Width Modulators)

Handle: The driver (HAL) handle

systemFreq_MHz: The system frequency, MHz

pwmPeriod_usec: The PWM period, usec

numPwmTicksPerIsrTick: The number of PWM clock ticks per ISR clock tick

HAL_setupPwmDacs ()

void HAL_setupPwmDacs(HAL_Handle handle)

Sets up the PWM DACs

Handle: The driver (HAL) handle

HAL_readTimerCnt ()

static inline uint32_t HAL_readTimerCnt(HAL_Handle handle,const uint_least8_t timerNumber)

Turns off the outputs of the EPWM peripherals which will put the power switches into a high impedance state

Handle: The driver (HAL) handle

HAL_reloadTimer () ***static inline void HAL_reloadTimer(HAL_Handle handle,const uint_least8_t timerNumber)***

Turns off the outputs of the EPWM peripherals which will put the power switches into a high impedance state

Handle: The driver (HAL) handle

HAL_readPwmPeriod ()

static inline uint16_t HAL_readPwmPeriod(HAL_Handle handle,const PWM_Number_e pwmNumber)

Reads PWM period register

Handle: The driver (HAL) handle

pwmNumber: The PWM number

Return: The PWM period value

HAL_disablePwm () ***void HAL_disablePwm (HAL_Handle handle);***

Turns off the outputs of the EPWM peripherals which will put the power switches into a high-impedance state

Handle: The driver (HAL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

HAL_enablePwm () *void HAL_enablePwm (HAL_Handle handle);*

Turns on the outputs of the EPWM peripheral which will allow the power switches to be controlled

Handle: The driver (HAL) handle

HAL_writeDacData ()

static inline void HAL_writeDacData(HAL_Handle handle,HAL_DacData_t *pDacData)

Writes DAC data to the PWM comparators for DAC (digital-to-analog conversion) output

Handle: The driver (HAL) handle

pDacData: The pointer to the DAC data

HAL_writePwmData()

void HAL_writePwmData (HAL_Handle handle, HAL_PwmData_t *pPwmData)

Writes PWM data to the PWM comparators for motor control

Handle: The driver (HAL) handle

pPwmData: The pointer to the PWM data

HAL_readPwmCmpA ()

static inline uint16_t HAL_readPwmCmpA(HAL_Handle handle,const PWM_Number_e pwmNumber)

Reads PWM compare register A

Handle: The driver (HAL) handle

pwmNumber: The PWM number

Return: The PWM compare value

HAL_readPwmCmpB ()

static inline uint16_t HAL_readPwmCmpB(HAL_Handle handle,const PWM_Number_e pwmNumber)

Reads PWM compare register B

Handle: The driver (HAL) handle

pwmNumber: The PWM number

Return: The PWM compare value

HAL_setTrigger () *static inline void HAL_setTrigger(HAL_Handle handle,const int16_t minwidth)*

Handle: The driver (HAL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

minwidth:

HAL_acqPwmInt ()

static inline void HAL_acqPwmInt(HAL_Handle handle,const PWM_Number_e pwmNumber)

Acknowledges an interrupt from the PWM

Handle: The driver (HAL) handle

pwmNumber: The PWM number

HAL_enablePwmInt ()

extern void HAL_enablePwmInt(HAL_Handle handle);

Enables the PWM interrupt

Handle: The driver (HAL) handle

HAL_hvProtection ()

void HAL_hvProtection(HAL_Handle handle)

Runs high voltage protection logic. Sets up the trip registers.

Handle: The driver (HAL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.3.4 HAL – CPU Timers

HAL_setupClks () ***void HAL_setupClks (HAL_Handle halHandle)***

Sets up the clocks

Handle: The driver (HAL) handle

HAL_setupTimers () ***void HAL_setupTimers(HAL_Handle handle,const uint_least16_t systemFreq_MHz);***

Setup CPU timers 0 and 1

Handle: The driver (HAL) handle

systemFreq_MHz: The system frequency, MHz

HAL_startTimer () ***static inline void HAL_startTimer(HAL_Handle handle,const uint_least8_t timerNumber)***

Starts the CPU timer

Handle: The driver (HAL) handle

timerNumber: The CPU timer number; 0, 1 or 2

HAL_stopTimer () ***static inline void HAL_stopTimer(HAL_Handle handle,const uint_least8_t timerNumber)***

Stops the CPU timer

Handle: The driver (HAL) handle

timerNumber: The timer number; 0, 1 or 2

HAL_setTimerPeriod ()

static inline void HAL_setTimerPeriod(HAL_Handle handle,const uint_least8_t timerNumber, const uint32_t period)

Sets the CPU timer period

Handle: The driver (HAL) handle

timerNumber: The timer number; 0, 1 or 2

period: The timer period

HAL_getTimerPeriod ()

static inline void HAL_getTimerPeriod(HAL_Handle handle,const uint_least8_t timerNumber, const uint32_t period)

Gets the CPU timer period

Handle: The driver (HAL) handle

timerNumber: The timer number; 0, 1 or 2

period: The timer period

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.3.5 HAL – GPIO and LED

HAL_setupGpios () ***void HAL_setupGpios(HAL_Handle handle)***

Sets up the GPIO

Handle: The driver (HAL) handle

HAL_toggleGpio () ***void HAL_toggleGpio (HAL_Handle handle ,const GPIO_Number_e gpioNumber);***

Takes in the enumeration GPIO_Number_e and toggles that GPIO pin.

Handle: The driver (HAL) handle

gpioNumber: The GPIO number

HAL_setGpioHigh () ***static inline void HAL_setGpioHigh(HAL_Handle handle,const GPIO_Number_e gpioNumber)***

Sets the GPIO pin high. Takes in the enumeration GPIO_Number_e and sets that GPIO pin high.

Handle: The driver (HAL) handle

gpioNumber: The GPIO number

HAL_setGpioLow () ***static inline void HAL_setGpioLowHAL_Handle handle,const GPIO_Number_e gpioNumber)***

Sets the GPIO pin low. Takes in the enumeration GPIO_Number_e and sets that GPIO pin low.

Handle: The driver (HAL) handle

gpioNumber: The GPIO number

HAL_toggleLed

Defines the function to turn LEDs on

```
#define HAL_toggleLed            HAL_toggleGpio
```


Copyright © 2013–2019, Texas Instruments Incorporated

3.3.3.6 HAL – Miscellaneous

HAL_init() ***HAL_Handle HAL_init(void *pMemory,const size_t numBytes)***

Initializes the driver (HAL) object and returns a handle to the HAL object

pMemory: A pointer to the memory for the driver object

numBytes: The number of bytes allocated for the driver object, bytes

Return: The driver (HAL) object handle

HAL_initIntVectorTable ()

void HAL_initIntVectorTable(HAL_Handle handle)

Initializes the interrupt vector table

Handle: The driver (HAL) handle

HAL_setParams () ***void HAL_setParams(HAL_Handle handle,const USER_Params *pUserParams)***

Sets the hardware abstraction layer parameters

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.

Handle: The driver (HAL) handle

pUserParams: The pointer to the user parameters

HAL_setupFlash () ***void HAL_setupFlash(HAL_Handle handle)***

Sets up the FLASH.

Handle: The driver (HAL) handle

HAL_setupPie () ***void HAL_setupPie(HAL_Handle handle)***

Sets up the Peripheral Interrupt Expansion (PIE).

Handle: The driver (HAL) handle

HAL_setupPll () ***void HAL_setupPll(HAL_Handle handle,const PLL_ClkFreq_e clkFreq)***

Sets up the Phase-Lock Loop (PLL).

Handle: The driver (HAL) handle

clkFreq: The clock frequency

HAL_setupPeripheralClks ()

void HAL_setupPeripheralClks (HAL_Handle handle)

Sets up the peripheral clocks.

Handle: The driver (HAL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

HAL_getOscTrimValue ()

uint16_t HAL_getOscTrimValue(int16_t coarse, int16_t fine);

Converts coarse and fine oscillator trim values into a single 16-bit word value.

Handle: The driver (HAL) handle

coarse: The coarse trim portion of the oscillator trim

fine: The fine trim portion of the oscillator trim

Return: The combined trim value

HAL_OscTempComp () *void HAL_OscTempComp(HAL_Handle handle);*

Executes the oscillator 1 and 2 calibration functions.

Handle: The driver (HAL) handle

HAL_osc1Comp () *void HAL_osc1Comp(HAL_Handle handle, const int16_t sensorSample);*

Executes the oscillator 1 calibration based on input sample.

Handle: The driver (HAL) handle

HAL_osc2Comp ()

void HAL_osc2Comp(HAL_Handle handle, const int16_t sensorSample);

Executes the oscillator 2 calibration based on input sample.

Handle: The driver (HAL) handle

HAL_setupFaults () *extern void HAL_setupFaults(HAL_Handle handle);*

Configures the fault protection logic

Sets up the trip zone inputs so that when a comparator signal from outside the micro-controller trips a fault, the EPWM peripheral blocks will force the power switches into a high-impedance state.

Handle: The driver (HAL) handle

HAL_setParams() *void HAL_setParams(HAL_Handle handle,const USER_Params *pUserParams)*

Sets the driver parameters

Handle: The driver (HAL) handle

pUserParams: The pointer to the user parameters

HAL_enableDebugInt ()

void HAL_enableDebugInt (HAL_Handle handle);

Enables real-time debug global interrupt

Handle: The driver (HAL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

HAL_enableGlobalInts ()

void HAL_enableGlobalInts(HAL_Handle handle);

Enables global interrupts

Handle: The driver (HAL) handle

HAL_disableGlobalInts ()

void HAL_disableGlobalInts(HAL_Handle handle);

Disables global interrupts

Handle: The driver (HAL) handle

HAL_disableWdog () *xtern void HAL_disableWdog(HAL_Handle handle);*

Disables the watchdog

Handle: The driver (HAL) handle

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.4 User Settings – user.c, user.h, userParams.h

3.3.4.1 USER Enumerations and Structures

Struct_USER_Params_

Structure for user parameters.

```
typedef struct _USER_Params_
{
  float_t       iqFullScaleCurrent_A;         //!< Defines the full scale current
for the IQ variables, A
  float_t       iqFullScaleVoltage_V;         //!< Defines the full scale voltage
for the IQ variable, V
  float_t       iqFullScaleFreq_Hz;           //!< Defines the full scale
frequency for IQ variable, Hz

  uint_least16_t  numIsrTicksPerCtrlTick;       //!< Defines the number of
Interrupt Service Routine (ISR) clock ticks per controller clock tick
  uint_least16_t  numCtrlTicksPerCurrentTick;   //!< Defines the number of
controller clock ticks per current controller clock tick
  uint_least16_t  numCtrlTicksPerEstTick;       //!< Defines the number of
controller clock ticks per estimator clock tick
  uint_least16_t  numCtrlTicksPerSpeedTick;     //!< Defines the number of
controller clock ticks per speed controller clock tick
  uint_least16_t  numCtrlTicksPerTrajTick;      //!< Defines the number of
controller clock ticks per trajectory clock tick
  uint_least8_t   numCurrentSensors;            //!< Defines the number of
current sensors
  uint_least8_t   numVoltageSensors;            //!< Defines the number of
voltage sensors

  float_t       offsetPole_rps;               //!< Defines the pole location for
the voltage and current offset estimation, rad/s
  float_t       fluxPole_rps;                 //!< Defines the pole location for
the flux estimation, rad/s
  float_t       zeroSpeedLimit;               //!< Defines the low speed limit
for the flux integrator, pu
  float_t       forceAngleFreq_Hz;            //!< Defines the force angle
frequency, Hz
  float_t       maxAccel_Hzps;                //!< Defines the maximum
acceleration for the speed profiles, Hz/s
  float_t       maxAccel_est_Hzps;            //!< Defines the maximum
acceleration for the estimation speed profiles, Hz/s
  float_t       directionPole_rps;            //!< Defines the pole location for
the direction filter, rad/s
  float_t       speedPole_rps;                //!< Defines the pole location for
the speed control filter, rad/s
  float_t       dcBusPole_rps;                //!< Defines the pole location for
the DC bus filter, rad/s
  float_t       fluxFraction;                 //!< Defines the flux fraction for
Id rated current estimation
  float_t       indEst_speedMaxFraction;      //!< Defines the fraction of
SpeedMax to use during inductance estimation
  float_t       powerWarpGain;                //!< Defines the PowerWarp gain for
computing Id reference

  uint_least16_t  systemFreq_MHz;               //!< Defines the system clock
frequency, MHz

  float_t       pwmPeriod_usec;               //!< Defines the Pulse Width
Modulation (PWM) period, usec
  float_t       voltage_sf;                   //!< Defines the voltage scale
factor for the system
  float_t       current_sf;                   //!< Defines the current scale
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
factor for the system
  float_t       voltageFilterPole_rps;        //!< Defines the analog voltage
filter pole location, rad/s
  float_t       maxVsMag_pu;                  //!< Defines the maximum voltage
magnitude, pu
  float_t       estKappa;                     //!< Defines the convergence factor
for the estimator

  MOTOR_Type_e    motor_type;                   //!< Defines the motor type
  uint_least16_t  motor_numPolePairs;           //!< Defines the number of pole
pairs for the motor

  float_t       motor_ratedFlux;              //!< Defines the rated flux of the
motor, V/Hz
  float_t       motor_Rr;                     //!< Defines the rotor resistance,
ohm
  float_t       motor_Rs;                     //!< Defines the stator resistance,
ohm
  float_t       motor_Ls_d;                   //!< Defines the direct stator
inductance, H
  float_t       motor_Ls_q;                   //!< Defines the quadrature stator
inductance, H
  float_t       maxCurrent;                   //!< Defines the maximum current
value, A
  float_t       maxCurrent_resEst;            //!< Defines the maximum current
value for resistance estimation, A
  float_t       maxCurrent_indEst;            //!< Defines the maximum current
value for inductance estimation, A
  float_t       maxCurrentSlope;              //!< Defines the maximum current
slope for Id current trajectory
  float_t       maxCurrentSlope_powerWarp;    //!< Defines the maximum current
slope for Id current trajectory during PowerWarp
  float_t       IdRated;                      //!< Defines the Id rated current
value, A
  float_t       IdRatedFraction_indEst;       //!< Defines the fraction of Id
rated current to use during inductance estimation
  float_t       IdRatedFraction_ratedFlux;    //!< Defines the fraction of Id
rated current to use during rated flux estimation
  float_t       IdRated_delta;                //!< Defines the Id rated delta
current value, A
  float_t       fluxEstFreq_Hz;               //!< Defines the flux estimation
frequency, Hz

  uint_least32_t  ctrlWaitTime[CTRL_numStates]; //!< Defines the wait times for
each controller state, estimator ticks
  uint_least32_t  estWaitTime[EST_numStates];   //!< Defines the wait times for
each estimator state, estimator ticks
  uint_least32_t  FluxWaitTime[EST_Flux_numStates]; //!< Defines the wait times
for each Ls estimator, estimator ticks
  uint_least32_t  LsWaitTime[EST_Ls_numStates]; //!< Defines the wait times for
each Ls estimator, estimator ticks
  uint_least32_t  RsWaitTime[EST_Rs_numStates]; //!< Defines the wait times for
each Rs estimator, estimator ticks
  uint_least32_t  ctrlFreq_Hz;                  //!< Defines the controller
frequency, Hz
  uint_least32_t  estFreq_Hz;                   //!< Defines the estimator
frequency, Hz
  uint_least32_t  RoverL_estFreq_Hz;            //!< Defines the R/L estimation
frequency, Hz
  uint_least32_t  trajFreq_Hz;                  //!< Defines the trajectory
frequency, Hz

  float_t       ctrlPeriod_sec;               //!< Defines the controller
execution period, sec

  float_t       maxNegativeIdCurrent_a;       //!< Defines the maximum negative
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
current that the Id PID is allowed to go to, A

  USER_ErrorCode_e  errorCode;
} USER_Params;
```

USER_ErrorCode_e

Structure for user error codes.

```
typedef enum
{
  USER_ErrorCode_NoError=0,                          //!< no error error code
  USER_ErrorCode_iqFullScaleCurrent_A_High=1,        //!< iqFullScaleCurrent_A
too high error code
  USER_ErrorCode_iqFullScaleCurrent_A_Low=2,         //!< iqFullScaleCurrent_A
too low error code
  USER_ErrorCode_iqFullScaleVoltage_V_High=3,        //!< iqFullScaleVoltage_V
too high error code
  USER_ErrorCode_iqFullScaleVoltage_V_Low=4,         //!< iqFullScaleVoltage_V
too low error code
  USER_ErrorCode_iqFullScaleFreq_Hz_High=5,          //!< iqFullScaleFreq_Hz too
high error code
  USER_ErrorCode_iqFullScaleFreq_Hz_Low=6,           //!< iqFullScaleFreq_Hz too
low error code
  USER_ErrorCode_numPwmTicksPerIsrTick_High=7,       //!< numPwmTicksPerIsrTick
too high error code
  USER_ErrorCode_numPwmTicksPerIsrTick_Low=8,        //!< numPwmTicksPerIsrTick
too low error code
  USER_ErrorCode_numIsrTicksPerCtrlTick_High=9,      //!< numIsrTicksPerCtrlTick
too high error code
  USER_ErrorCode_numIsrTicksPerCtrlTick_Low=10,      //!< numIsrTicksPerCtrlTick
too low error code
  USER_ErrorCode_numCtrlTicksPerCurrentTick_High=11, //!<
numCtrlTicksPerCurrentTick too high error code
  USER_ErrorCode_numCtrlTicksPerCurrentTick_Low=12,  //!<
numCtrlTicksPerCurrentTick too low error code
  USER_ErrorCode_numCtrlTicksPerEstTick_High=13,     //!< numCtrlTicksPerEstTick
too high error code
  USER_ErrorCode_numCtrlTicksPerEstTick_Low=14,      //!< numCtrlTicksPerEstTick
too low error code
  USER_ErrorCode_numCtrlTicksPerSpeedTick_High=15,   //!<
numCtrlTicksPerSpeedTick too high error code
  USER_ErrorCode_numCtrlTicksPerSpeedTick_Low=16,    //!<
numCtrlTicksPerSpeedTick too low error code
  USER_ErrorCode_numCtrlTicksPerTrajTick_High=17,    //!<
numCtrlTicksPerTrajTick too high error code
  USER_ErrorCode_numCtrlTicksPerTrajTick_Low=18,     //!<
numCtrlTicksPerTrajTick too low error code
  USER_ErrorCode_numCurrentSensors_High=19,          //!< numCurrentSensors too
high error code
  USER_ErrorCode_numCurrentSensors_Low=20,           //!< numCurrentSensors too
low error code
  USER_ErrorCode_numVoltageSensors_High=21,          //!< numVoltageSensors too
high error code
  USER_ErrorCode_numVoltageSensors_Low=22,           //!< numVoltageSensors too
low error code
  USER_ErrorCode_offsetPole_rps_High=23,             //!< offsetPole_rps too
high error code
  USER_ErrorCode_offsetPole_rps_Low=24,              //!< offsetPole_rps too low
error code
  USER_ErrorCode_fluxPole_rps_High=25,               //!< fluxPole_rps too high
error code
  USER_ErrorCode_fluxPole_rps_Low=26,                //!< fluxPole_rps too low
error code
  USER_ErrorCode_zeroSpeedLimit_High=27,             //!< zeroSpeedLimit too
high error code
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
  USER_ErrorCode_zeroSpeedLimit_Low=28,              //!< zeroSpeedLimit too low
error code
  USER_ErrorCode_forceAngleFreq_Hz_High=29,          //!< forceAngleFreq_Hz too
high error code
  USER_ErrorCode_forceAngleFreq_Hz_Low=30,           //!< forceAngleFreq_Hz too
low error code
  USER_ErrorCode_maxAccel_Hzps_High=31,              //!< maxAccel_Hzps too high
error code
  USER_ErrorCode_maxAccel_Hzps_Low=32,               //!< maxAccel_Hzps too low
error code
  USER_ErrorCode_maxAccel_est_Hzps_High=33,          //!< maxAccel_est_Hzps too
high error code
  USER_ErrorCode_maxAccel_est_Hzps_Low=34,           //!< maxAccel_est_Hzps too
low error code
  USER_ErrorCode_directionPole_rps_High=35,          //!< directionPole_rps too
high error code
  USER_ErrorCode_directionPole_rps_Low=36,           //!< directionPole_rps too
low error code
  USER_ErrorCode_speedPole_rps_High=37,              //!< speedPole_rps too high
error code
  USER_ErrorCode_speedPole_rps_Low=38,               //!< speedPole_rps too low
error code
  USER_ErrorCode_dcBusPole_rps_High=39,              //!< dcBusPole_rps too high
error code
  USER_ErrorCode_dcBusPole_rps_Low=40,               //!< dcBusPole_rps too low
error code
  USER_ErrorCode_fluxFraction_High=41,               //!< fluxFraction too high
error code
  USER_ErrorCode_fluxFraction_Low=42,                //!< fluxFraction too low
error code
  USER_ErrorCode_indEst_speedMaxFraction_High=43,    //!<
indEst_speedMaxFraction too high error code
  USER_ErrorCode_indEst_speedMaxFraction_Low=44,     //!<
indEst_speedMaxFraction too low error code
  USER_ErrorCode_powerWarpGain_High=45,              //!< powerWarpGain too high
error code
  USER_ErrorCode_powerWarpGain_Low=46,               //!< powerWarpGain too low
error code
  USER_ErrorCode_systemFreq_MHz_High=47,             //!< systemFreq_MHz too
high error code
  USER_ErrorCode_systemFreq_MHz_Low=48,              //!< systemFreq_MHz too low
error code
  USER_ErrorCode_pwmFreq_kHz_High=49,                //!< pwmFreq_kHz too high
error code
  USER_ErrorCode_pwmFreq_kHz_Low=50,                 //!< pwmFreq_kHz too low
error code
  USER_ErrorCode_voltage_sf_High=51,                 //!< voltage_sf too high
error code
  USER_ErrorCode_voltage_sf_Low=52,                  //!< voltage_sf too low
error code
  USER_ErrorCode_current_sf_High=53,                 //!< current_sf too high
error code
  USER_ErrorCode_current_sf_Low=54,                  //!< current_sf too low
error code
  USER_ErrorCode_voltageFilterPole_Hz_High=55,       //!< voltageFilterPole_Hz
too high error code
  USER_ErrorCode_voltageFilterPole_Hz_Low=56,        //!< voltageFilterPole_Hz
too low error code
  USER_ErrorCode_maxVsMag_pu_High=57,                //!< maxVsMag_pu too high
error code
  USER_ErrorCode_maxVsMag_pu_Low=58,                 //!< maxVsMag_pu too low
error code
  USER_ErrorCode_estKappa_High=59,                   //!< estKappa too high
error code
  USER_ErrorCode_estKappa_Low=60,                    //!< estKappa too low error
code
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
  USER_ErrorCode_motor_type_Unknown=61,              //!< motor type unknown
error code
  USER_ErrorCode_motor_numPolePairs_High=62,         //!< motor_numPolePairs too
high error code
  USER_ErrorCode_motor_numPolePairs_Low=63,          //!< motor_numPolePairs too
low error code
  USER_ErrorCode_motor_ratedFlux_High=64,            //!< motor_ratedFlux too
high error code
  USER_ErrorCode_motor_ratedFlux_Low=65,             //!< motor_ratedFlux too
low error code
  USER_ErrorCode_motor_Rr_High=66,                   //!< motor_Rr too high
error code
  USER_ErrorCode_motor_Rr_Low=67,                    //!< motor_Rr too low error
code
  USER_ErrorCode_motor_Rs_High=68,                   //!< motor_Rs too high
error code
  USER_ErrorCode_motor_Rs_Low=69,                    //!< motor_Rs too low error
code
  USER_ErrorCode_motor_Ls_d_High=70,                 //!< motor_Ls_d too high
error code
  USER_ErrorCode_motor_Ls_d_Low=71,                  //!< motor_Ls_d too low
error code
  USER_ErrorCode_motor_Ls_q_High=72,                 //!< motor_Ls_q too high
error code
  USER_ErrorCode_motor_Ls_q_Low=73,                  //!< motor_Ls_q too low
error code
  USER_ErrorCode_maxCurrent_High=74,                 //!< maxCurrent too high
error code
  USER_ErrorCode_maxCurrent_Low=75,                  //!< maxCurrent too low
error code
  USER_ErrorCode_maxCurrent_resEst_High=76,          //!< maxCurrent_resEst too
high error code
  USER_ErrorCode_maxCurrent_resEst_Low=77,           //!< maxCurrent_resEst too
low error code
  USER_ErrorCode_maxCurrent_indEst_High=78,          //!< maxCurrent_indEst too
high error code
  USER_ErrorCode_maxCurrent_indEst_Low=79,           //!< maxCurrent_indEst too
low error code
  USER_ErrorCode_maxCurrentSlope_High=80,            //!< maxCurrentSlope too
high error code
  USER_ErrorCode_maxCurrentSlope_Low=81,             //!< maxCurrentSlope too
low error code
  USER_ErrorCode_maxCurrentSlope_powerWarp_High=82,  //!<
maxCurrentSlope_powerWarp too high error code
  USER_ErrorCode_maxCurrentSlope_powerWarp_Low=83,   //!<
maxCurrentSlope_powerWarp too low error code
  USER_ErrorCode_IdRated_High=84,                    //!< IdRated too high error
code
  USER_ErrorCode_IdRated_Low=85,                     //!< IdRated too low error
code
  USER_ErrorCode_IdRatedFraction_indEst_High=86,     //!< IdRatedFraction_indEst
too high error code
  USER_ErrorCode_IdRatedFraction_indEst_Low=87,      //!< IdRatedFraction_indEst
too low error code
  USER_ErrorCode_IdRatedFraction_ratedFlux_High=88,  //!<
IdRatedFraction_ratedFlux too high error code
  USER_ErrorCode_IdRatedFraction_ratedFlux_Low=89,   //!<
IdRatedFraction_ratedFlux too low error code
  USER_ErrorCode_IdRated_delta_High=90,              //!< IdRated_delta too high
error code
  USER_ErrorCode_IdRated_delta_Low=91,               //!< IdRated_delta too low
error code
  USER_ErrorCode_fluxEstFreq_Hz_High=92,             //!< fluxEstFreq_Hz too
high error code
  USER_ErrorCode_fluxEstFreq_Hz_Low=93,              //!< fluxEstFreq_Hz too low
error code
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
  USER_ErrorCode_ctrlFreq_Hz_High=94,                //!< ctrlFreq_Hz too high
error code
  USER_ErrorCode_ctrlFreq_Hz_Low=95,                 //!< ctrlFreq_Hz too low
error code
  USER_ErrorCode_estFreq_Hz_High=96,                 //!< estFreq_Hz too high
error code
  USER_ErrorCode_estFreq_Hz_Low=97,                  //!< estFreq_Hz too low
error code
  USER_ErrorCode_RoverL_estFreq_Hz_High=98,          //!< RoverL_estFreq_Hz too
high error code
  USER_ErrorCode_RoverL_estFreq_Hz_Low=99,           //!< RoverL_estFreq_Hz too
low error code
  USER_ErrorCode_trajFreq_Hz_High=100,               //!< trajFreq_Hz too high
error code
  USER_ErrorCode_trajFreq_Hz_Low=101,                //!< trajFreq_Hz too low
error code
  USER_ErrorCode_ctrlPeriod_sec_High=102,            //!< ctrlPeriod_sec too
high error code
  USER_ErrorCode_ctrlPeriod_sec_Low=103,             //!< ctrlPeriod_sec too low
error code
  USER_ErrorCode_maxNegativeIdCurrent_a_High=104,    //!< maxNegativeIdCurrent_a
too high error code
  USER_ErrorCode_maxNegativeIdCurrent_a_Low=105,     //!< maxNegativeIdCurrent_a
too low error code
  USER_numErrorCodes=106                             //!< the number of user
error codes
} USER_ErrorCode_e;
```


Copyright © 2013–2019, Texas Instruments Incorporated

3.3.4.2 USER Set and Compute Functions

USER_setParams() ***void USER_setParams(USER_Params *pUserParams)***

Sets the user parameter values

pUserParams: A pointer to the user param structure

USER_calcPIgains () ***void USER_calcPIgains(CTRL_Handle handle)***

Updates Id and Iq PI gains

Handle: The controller (CTRL) handle

USER_computeTorque_Ls_Id_Iq_pu_to_Nm_sf ()
_iq USER_computeTorque_Ls_Id_Iq_pu_to_Nm_sf(void);

Computes the scale factor needed to convert from torque created by Ld, Lq, Id and Iq, from per unit to Nm

Return: The scale factor to convert torque from (Ld - Lq) * Id * Iq from per unit to Nm, in IQ24 format

USER_computeTorque_Flux_Iq_pu_to_Nm_sf ()
_iq USER_computeTorque_Flux_Iq_pu_to_Nm_sf(void);

Computes the scale factor needed to convert from torque created by flux and Iq, from per unit to Nm

Return: The scale factor to convert torque from Flux * Iq from per unit to Nm, in IQ24 format

USER_computeFlux_pu_to_Wb_sf ()
_iq USER_computeFlux_pu_to_Wb_sf (void);

Computes the scale factor needed to convert from per unit to Wb

Return: The scale factor to convert from flux per unit to flux in Wb, in IQ24 format

USER_computeFlux_pu_to_VpHz_sf () ***_iq USER_computeFlux_pu_to_VpHz_sf (void);***

Computes the scale factor needed to convert from per unit to V/Hz

Return: The scale factor to convert from flux per unit to flux in V/Hz, in IQ24 format

USER_computeFlux () ***_iq USER_computeFlux(CTRL_Handle handle, const _iq sf);***

Computes Flux in Wb or V/Hz depending on the scale factor sent as parameter

Handle: The controller (CTRL) handle

sf: The scale factor to convert flux from per unit to Wb or V/Hz

Return: The flux in Wb or V/Hz depending on the scale factor sent as parameter, in IQ24 format

Copyright © 2013–2019, Texas Instruments Incorporated

USER_computeTorque_Nm ()

_iq USER_computeTorque_Nm(CTRL_Handle handle, const _iq torque_Flux_sf, const _iq torque_Ls_sf);

Computes Flux in Wb or V/Hz depending on the scale factor sent as parameter

Handle: The controller (CTRL) handle

torque_Flux_sf: The scale factor to convert torque from (Ld - Lq) * Id * Iq from per unit to Nm

torque_Ls_sf: The scale factor to convert torque from Flux * Iq from per unit to Nm

Return: The torque in Nm, in IQ24 format

USER_computeTorque_lbin ()

_iq USER_computeTorque_lbin(CTRL_Handle handle, const _iq torque_Flux_sf, const _iq torque_Ls_sf);

Computes Torque in lbin

Handle: The controller (CTRL) handle

torque_Flux_sf: The scale factor to convert torque from (Ld - Lq) * Id * Iq from per unit to Nm

torque_Ls_sf: The scale factor to convert torque from Flux * Iq from per unit to Nm

Return: The torque in Nm, in IQ24 format

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.4.3 USER Error Handling Functions

USER_checkForErrors ()

void USER_checkForErrors(USER_Params *pUserParams);

Checks for errors in the user parameter values

pUserParams: A pointer to the user param structure

USER_getErrorCode ()

USER_ErrorCode_e USER_getErrorCode(USER_Params *pUserParams);

Gets the error code in the user parameters

pUserParams: A pointer to the user param structure

Return: The error code

USER_setErrorCode ()

void USER_setErrorCode(USER_Params *pUserParams,const USER_ErrorCode_e errorCode);

Sets the error code in the user parameters

pUserParams: A pointer to the user param structure

Return: The error code

Copyright © 2013–2019, Texas Instruments Incorporated

3.3.5 Miscellaneous Functions

softwareUpdate1p6 ()

void softwareUpdate1p6 (CTRL_Handle handle)

Recalculates inductances with the correct Q Format. A bug fix for InstaSPIN-FOC v1.6, the software version in ROM on the F2806x devices. This function only applies to F2806x devices, since F2802x and F2805x have v1.7 which does not need this fix.

Handle: The controller (CTRL) handle

3.4 InstaSPIN-MOTION and the SpinTAC API

InstaSPIN-MOTION combines InstaSPIN-FOC with the SpinTAC Motion Control Suite developed by LineStream Technologies. These components provide a low maintenance, high-performance, and easy to use solution for simple motion systems. SpinTAC offers two solutions: one for velocity control applications and one for position control applications.

The components of the velocity control solution include a motion sequence generator (SpinTAC Velocity Plan), motion profile generator (SpinTAC Velocity Move), closed-loop disturbance rejecting speed controller (SpinTAC Velocity Control), and system inertia identification (SpinTAC Velocity Identify).

The SpinTAC motion control library allows you to use multiple instances of each component. It allows for two instances of SpinTAC Plan and SpinTAC Move. It allows for controlling two motion axes using SpinTAC Control and one instance of SpinTAC Identify.

Similarly, the components of the position control solution include a signal converter (SpinTAC Position Convert), position sequence generator (SpinTAC Position Plan), motion profile generator (SpinTAC Position Move), and closed-loop disturbance rejecting cascaded position and speed controller (SpinTAC Position Control).

These components are packaged together as the SpinTAC motion control library which can be found in sw/modules/spintac. The SpinTAC motion control library is designed to be modular. This allows developers to include only selected SpinTAC components in projects in order to minimize code size. Any component can be used in conjunction with other SpinTAC components or third party software.

The SpinTAC motion control library allows you to use multiple instances of each component. It allows for two instances of SpinTAC Plan and SpinTAC Move. It allows for controlling two motion axes using SpinTAC Control and one instance of SpinTAC Identify.

Figure 3-2 an overview of how the components of the velocity solution of the SpinTAC Motion Suite connect together and how they interface with InstaSPIN-FOC.

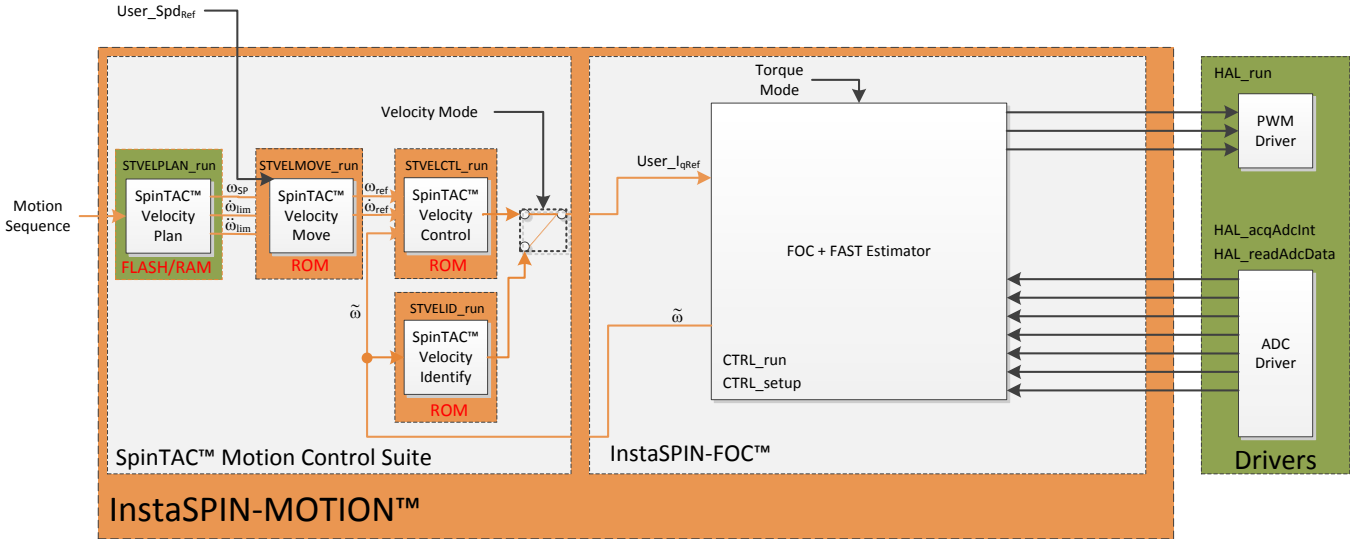


Figure 3-2. InstaSPIN-MOTION Velocity Control

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 3-3 provides an overview of how the components of the position solution of the SpinTAC Motion Suite connect together and how they interface with InstaSPIN-FOC.

Copyright © 2013–2019, Texas Instruments Incorporated

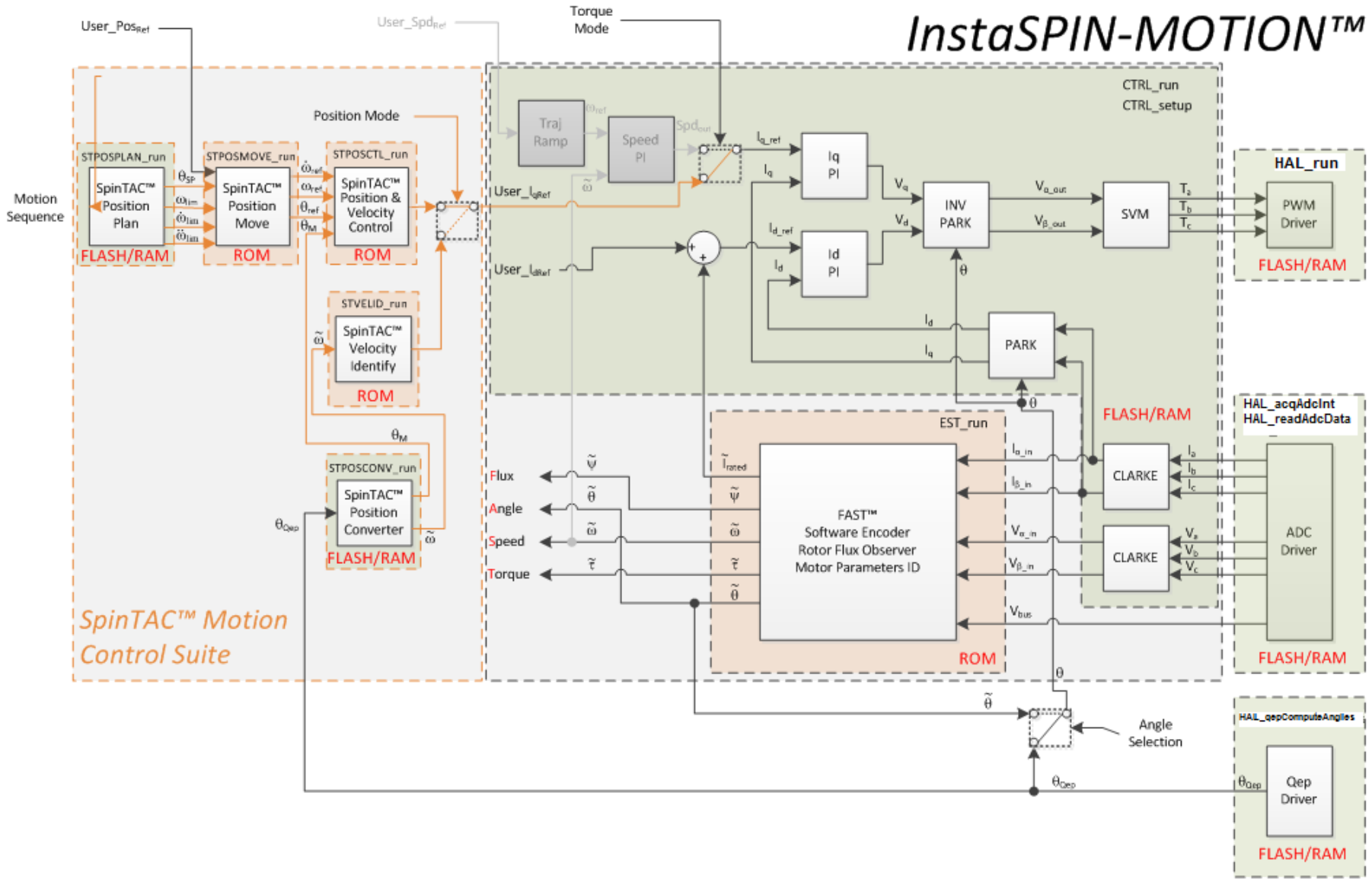


Figure 3-3. InstaSPIN-MOTION Position Control

Copyright © 2013–2019, Texas Instruments Incorporated

The API for the SpinTAC Motion Control Suite can be broken down into the following components, where each has a specific prefix for their API functions:

Component	Prefix
1. SpinTAC Velocity Control	STVELCTL
2. SpinTAC Velocity Move	STVELMOVE
3. SpinTAC Velocity Plan	STVELPLAN
4. SpinTAC Velocity Identify	STVELID
5. SpinTAC Position Convert	STPOSCONV
6. SpinTAC Position Control	STPOSCTL
7. SpinTAC Position Move	STPOSMOVE
8. SpinTAC Position Plan	STPOSPLAN

Each component of the SpinTAC Suite contains an initialize function and a run function. The initialize function is designed to establish the handle that will be used to interface to the SpinTAC component. The run function is the main calculation function for that component. All variables in these components can be accessed via get and set functions. The commonly used functions for SpinTAC components are detailed in Section 3.5.9.

Copyright © 2013–2019, Texas Instruments Incorporated

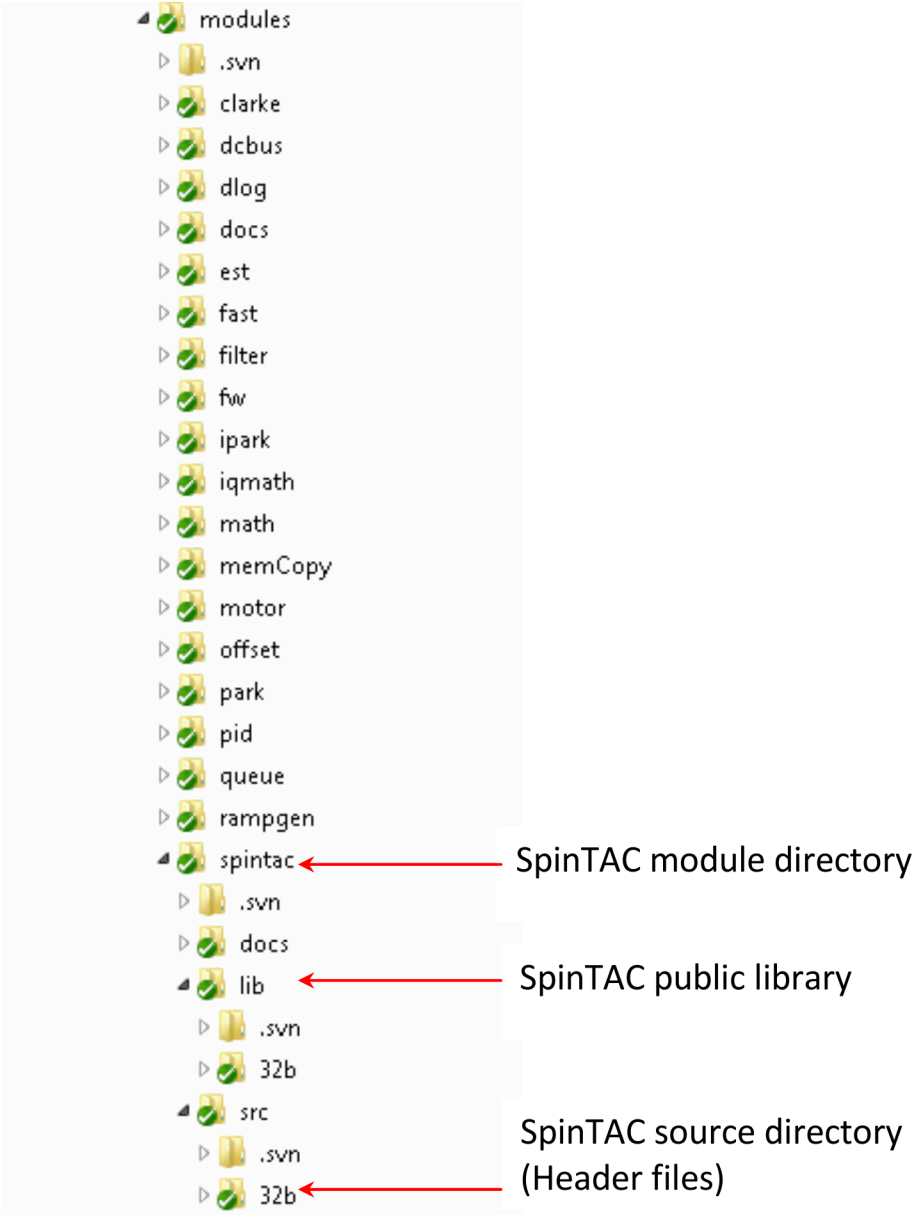


Figure 3-4. SpinTAC Module Directory Structure

3.4.1 Header Files, Public Library, and ROM Library

The SpinTAC suite is comprised of a public library, and a ROM library. The public library is also known as the SpinTAC library file. "SpinTAC.lib" is located at /sw/modules/spintac/lib/32b/ and the user must include this file in the project. If the project requires fpu32 support, use the library file "SpinTAC_fpu32.lib." Note that this library still uses the fixed point IQMath library. The header files that need to be included in user code are listed in Table 3-1. Developers can include the header files associated with the desired components. The source code for this library is not available.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-1. User Code Header Files

SpinTAC Component	Header File
SpinTAC Velocity Control	spintac_vel_ctl.h
SpinTAC Velocity Move	spintac_vel_move.h
SpinTAC Velocity Plan	spintac_vel_plan.h
SpinTAC Velocity Identify	spintac_vel_id.h
SpinTAC Position Convert	spintac_pos_conv.h
SpinTAC Position Control	spintac_pos_ctl.h
SpinTAC Position Move	spintac_pos_move.h
SpinTAC Position Plan	spintac_pos_plan.h
SpinTAC Version	spintac_version.h

The SpinTAC ROM library is a C-callable library embedded in on-chip execute-only ROM on the TMS320F2805xM and TMS320F2806xM devices. The source code for this library is not available. The ROM library implements the core SpinTAC functions that are called by the Public library.

3.4.2 Version Information

For detailed version information about the SpinTAC library, the ST_Ver_t object contains this information. The structure is detailed in Table 3-2.

Table 3-2. SpinTAC Version Structure

Member Name	Data Type	Description	V2.2.7 Example
Major	uint16_t	Major version of library	2
Minor	uint16_t	Minor version of library	2
Revision	uint16_t	Revision version of library	7
MathType	ST_MathType_e	Math implementation the library was compiled for.	FIXED_POINT32b
SecureROM	uint32_t	SecureROM version	20010008
Date	int32_t	Date the library was compiled	20140530
Label	uint_least8_t [10]	Other information about library	TI_C2000

3.4.2.1 Code Example for Returning SpinTAC Version Information

There are four steps to returning the SpinTAC version information. These steps are done in each example lab project. They are included below to show how simple it is to include the SpinTAC version information in your project.

3.4.2.1.1 Include the Header File

This should be done with the rest of the module header file includes. In the InstaSPIN-MOTION lab example project, this file is included in all of the SpinTAC module header files. For your project, this step can be completed by including spintac_velocity.h or spintac_position.h.

```
// SpinTAC
#include "spintac_velocity.h"

//OR
#include "spintac_position.h"
```


Copyright © 2013–2019, Texas Instruments Incorporated

3.4.2.1.2 Declare the Global Structure

This should be done with the global variable declarations in the main source file. In the example lab projects, this structure is included in the ST_Obj structure that is declared as part of the spintac_velocity.h or spintac_position.h header files.

```
ST_Obj    st_obj;
ST_Handle stHandle;
```

3.4.2.1.3 Initialize the Configuration Variables

This should be done in the main function of the project ahead of the forever loop. This will load all of the default values into SpinTAC Version structure. This step can be completed by running the function ST_init that is declared in the spintac_velocity.h or spintac_position.h header files.

```
// initialize the SpinTAC Components
stHandle = ST_init(&st_obj, sizeof(st_obj));
```

3.4.2.1.4 Return the Version Information

Now that the version handle has been defined it can be used to return the version information of the SpinTAC library.

```
uint16_t      major, minor, revision;      // Variables to return the version numbers
ST_getVersionNumbers(stVersionHandle, &major, &minor, &revision);
```

3.4.3 SpinTAC Structure Names

All structure data type names follow the pattern:

ST_[*Object*] [*Functionality*] [*Subfunctionality*(*optional*)]_**t**

For example, the Velocity Control structure type is named ST_VelCtl_t.

The Configuration substructure type of Velocity Control is named ST_VelCtlCfg_t, and is contained inside the structure ST_VelCtl_t

All the structures and sub structure names of SpinTAC are listed in Table 3-3.

Table 3-3. SpinTAC Structure Names

Component	Structure Name	Substructure Name	Description
Velocity Control	ST_VelCtl_t		Main structure
		ST_VelCtlCfg_t	Configuration substructure
Velocity Move	ST_VelMove_t		Main structure
		ST_VelMoveCfg_t	Configuration substructure
		ST_VelMoveMsg_t	Information substructure
Velocity Plan	ST_VelPlan_t		Main structure
Velocity Identify	ST_VelId_t		Main structure
		ST_VelIdCfg_t	Configuration substructure
Position Convert	ST_PosConv_t		Main structure
		ST_PosConvCfg_t	Configuration substructure
Position Control	ST_PosCtl_t		Main structure
		ST_PosCtlCfg_t	Configuration substructure

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-3. SpinTAC Structure Names (continued)

Component	Structure Name	Substructure Name	Description
Position Move	ST_PosMove_t		Main structure
		ST_PosMoveCfg_t	Configuration substructure
		ST_PosMoveMsg_t	Information substructure
Position Plan	ST_PosPlan_t		Main structure
Version	ST_Ver_t		Main Structure

3.4.4 SpinTAC Variables

Variables are broadly classified as shown in Table 3-4.

Table 3-4. SpinTAC Variables

Variable Categories	Subclasses
Configuration Variables: Variables used to configure SpinTAC components.	**System Variables:** These are known values that are based on system parameters. Some examples: 1. Sample time of the interrupt 2. Scaling factor between a mechanical revolution and user unit. 3. The system gain is a special case in that the inertia in the velocity loop or position-velocity loop is normally obtained by running an inertia estimation function for a constant inertia system. A constant inertia can be estimated by Inertia Identify in open loop. However, if the system inertia is gradually changing, it needs to be calculated or estimated outside Velocity Control and sent to Velocity Control in real time.
	Protection Variables: Used to indicate the upper limit or lower limit of other variables. For this reason, protection variables all have the suffix "Max" or "Min". Some examples: 1. OutMax and OutMin are the protection bounds for the variable Out. These bounds come from the fixed-point IQMath calculation limit, from the physical limit, or the safe limit obtained through experience. 2. VelMax is determined by the maximum allowable velocity for a specific motor, which is a physical limit. 3. BwMax, the Bandwidth upper limit can be set high in the beginning and then decreased after a reasonable upper bound is determined through testing.
	Tuning Variables: Parameters used to give users flexibility to adjust SpinTAC components' behavior. Some examples: 1. Control bandwidth of Velocity Control. 2. Velocity, acceleration/deceleration, and jerk limits of Velocity Move. 3. Low pass filter time constant for Velocity Identify.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-4. SpinTAC Variables (continued)

Variable Categories	Subclasses
Input/Output Variables: Variables that are used to pass values into and out of the SpinTAC components	**Input variables:** Interfaces for external signals to be input into a SpinTAC component. They receive the external signals before the component function is called. Examples include: 1. Reference and feedback signals of Velocity Control 2. Feedback signal of Inertia Identify. **Control Variables:** Signals used to control each component. Examples are: 1. ENB and RES are control variables for each SpinTAC component. Generally, when the value of RES is false, ENB enables a component on the rising edge and disables a component when it is set to false. When the value of RES is true, the component is disabled (in RESET) and the value of ENB is held at false. 2. TST is a control variable used to test the Velocity Move component. If the value of TST is true, Velocity Move only gives the profile information without producing the profile to the controller; otherwise, it will produce the profile. This control bit can be used by the designers to verify the calculated profile limits and profile time before applying the profile to the controller.
	Output variables: Contain the outputs of a SpinTAC component. These variables need to be set to the appropriate external variables after the component function is called. Examples: 1. Control output of Velocity Control and Inertia Identify 2. Profile references of Velocity Move. **Information Variables:** Read-only variables, which provide useful information about a SpinTAC component. Examples include: 1. State and the error code of any SpinTAC component 2. Profile time and the actual limits of Velocity Move.
Internal Variables: The internal variables in the SpinTAC components should not be accessed by customers. It is hazardous to modify the internal variables.The internal variables are stored in locations declared as bulk memory and are not listed in this document.	

3.5 SpinTAC API

This section describes the Application Program Interface (API) of each of the SpinTAC components including each component's internal state machine, primary functions and data structures for control and configuration. The components of the velocity solution are presented first, then the components of the position solution.

3.5.1 SpinTAC Velocity Control

The SpinTAC Velocity Controller is different from error-based control designs. The following example is used to illustrate the difference.

A velocity system can be described with Equation 1:

$$J\dot{v}(t) = f\left(v(t), d(t)\right) + u(t) \tag{1}$$

In Equation 1, v(t), u(t), and d(t) are system output (velocity), system input (torque), and external disturbance respectively; f(·) is an unknown nonlinear function, and J is the system inertia. In traditional control design, a PI controller would be used to control these dynamics with the proportional gain and integral gains determined experimentally. Model-based controllers, on the other hand, respond to the dynamics based on a predefined linear or nonlinear model.

The SpinTAC Velocity Controller is unique in that it treats the nonlinear term f(·) as a disturbance that can be estimated and rejected. The tuning process is also simplified via a parameterization method that enables high-performance control of dynamical systems using a single tuning parameter: the control bandwidth.

3.5.1.1 SpinTAC Velocity Control Interface

The SpinTAC Velocity Control interfaces and functions are shown in Figure 3-5.

Copyright © 2013–2019, Texas Instruments Incorporated

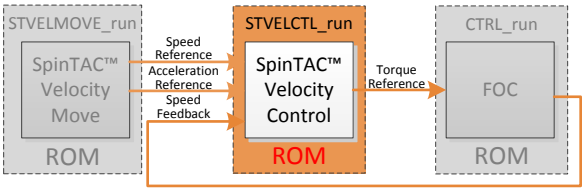


Figure 3-5. SpinTAC Velocity Control Interfaces

NOTE: In SpinTAC Velocity Control, the value for inertia can be obtained by the SpinTAC Velocity Identify component. The switch cfg.FiltEn can be enabled if the feedback is very noisy. The only controller tuning parameter is the bandwidth which is set via BwScale.

Table 3-5 lists the interface parameters for SpinTAC Velocity Control.

Table 3-5. SpinTAC Velocity Control Interface Parameters

Signal Type	Structure Member Name	Data Type	Description	Value Range	Units
Config	cfg.Axis	ST_Axis_e	SpinTAC Control Axis ID	{ST_AXIS0, ST_AXIS1}	
	cfg.T_sec	_iq24	Sampling time	(0 , 1]	s
	cfg.InertiaMax	_iq24	Maximum system inertia for velocity loop control	(0, 100]	PU · s^2 / pu
	cfg.InertiaMin	_iq24	Minimum system inertia for velocity loop control	(0, cfg.InertiaMax]	PU · s^2 / pu
	cfg.OutMax	_iq24	Maximum control signal	[-1 , 1]	PU
	cfg.OutMin	_iq24	Minimum control signal	[-1, cfg.OutMax]	PU
	cfg.VelMax	_iq24	Maximum reference signal (maximum velocity of the system)	[-1 , 1]	pu / s
	cfg.VelMin	_iq24	Minimum reference signal (minimum velocity of the system)	[-1 , cfg.VelMax]	pu / s
	cfg.BwScaleMax	_iq24	Bandwidth scale upper limit	[0.01, min(100, 0.01/ cfg.T_sec)]	
	cfg.BwScaleMin	_iq24	Bandwidth scale lower limit	[0.01, cfg.BwScaleMax]	
	cfg.FiltEn	bool	Enable feedback low pass filter	false: disabled; true: enabled	
Inputs	VelRef	_iq24	Reference signal (velocity reference)	[cfg.VelMin, cfg.VelMax]	pu / s
	AccRef	_iq24	Feedforward signal (acceleration reference)		pu / s^2
	VelFdb	_iq24	Feedback signal (velocity feedback)		pu / s
	Inertia	_iq24	System Inertia	[cfg.InertiaMin, cfg.InertiaMax]	PU · s^2 / pu
	Friction	_iq24	Friction coefficient	[0, 5]	PU · s / pu
Tuning	BwScale	_iq24	Bandwidth scale	[cfg.BwScaleMin, cfg.BwScaleMax]	
Control	ENB	bool	Enable bit	false: disabled; true: enabled	
	RES	bool	Reset bit	false: not reset; true: reset ERR_ID, and hold Out as 0	
Output	Out	_iq24	Control output	[cfg.OutMin, cfg.OutMax]	PU

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-5. SpinTAC Velocity Control Interface Parameters (continued)

Signal Type	Structure Member Name	Data Type	Description	Value Range	Units
Info	Bw_radps	_iq20	Controller bandwidth		rad/s
	STATUS	ST_CtlStatus_e	Status information	{ST_CTL_IDLE, ST_CTL_INIT, ST_CTL_CONF, ST_CTL_BUSY}	
	ERR_ID	uint16_t	Error code	See Table 12-2	

3.5.1.2 SpinTAC Velocity Control Run Function

The primary function is STVELCTL_run(ST_VELCTL_Handle handle), where handle is a pointer to a specific ST_VelCtl_t object. This function runs SpinTAC Velocity Control. It is recommended to run this controller at one-fifth or one-tenth the rate of the current controller.

Copyright © 2013–2019, Texas Instruments Incorporated

void **STVELCTL_run**(ST_VELCTL_Handle handle)

Parameters:

No.	Type	Parameters	Description
1	ST_VELCTL_Handle	Handle	The pointer to a ST_VelCtl_t object

The SpinTAC Velocity Control state transition map is shown in Figure 3-6.

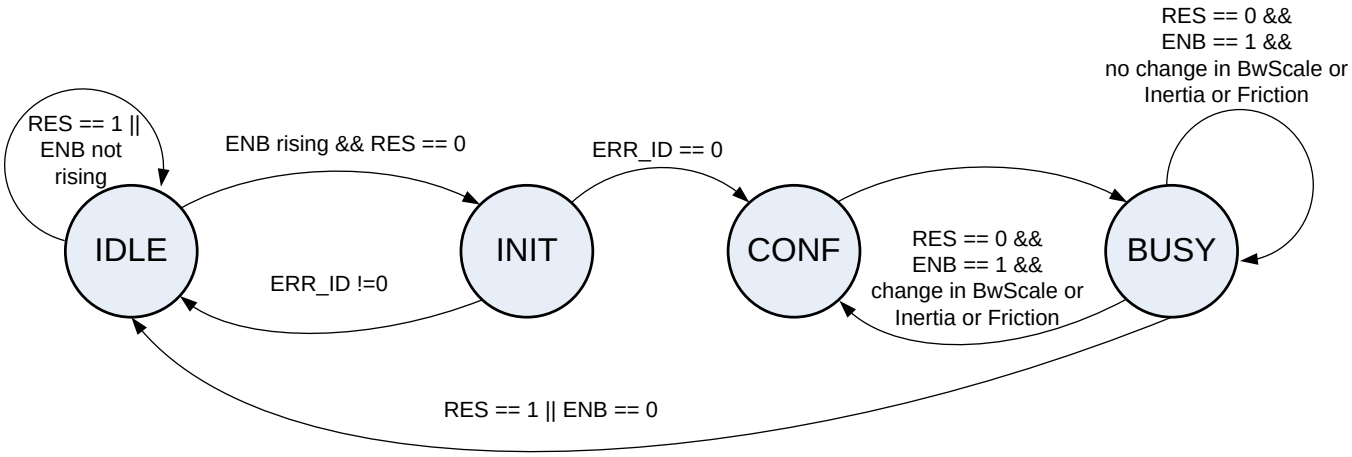


Figure 3-6. SpinTAC Velocity Control State Transition Map

Note in Figure 3-6, the transitions from state IDLE to INIT, INIT to CONF, and CONF to BUSY, happen within one sample time. Therefore, the controller works directly at the sample time when it is enabled.

The SpinTAC Velocity Control states transitions are described in Table 3-6.

Table 3-6. SpinTAC Velocity Control State Transition

From State	To State	Transition Condition	Action
IDLE			1. Set ENB = false; 2. Hold Out as zero Parameter validation 1. Validate the configuration parameters, including cfg.InertiaMax, cfg.InertiaMin, cfg.BwScaleMax, cfg.BwScaleMin, cfg.Axis, cfg.FiltEn, and cfg.T_sec. If any of the checked variables are invalid, ERR_ID will be nonzero.
	INIT	RES == false AND ENB is on rising edge AND ERR_ID == 0	
INIT			Parameter validation 1. Validate the parameters, including cfg.VelMax, cfg.VelMin, cfg.OutMax, cfg.OutMin. If any of the checked variables are invalid, ERR_ID will be nonzero.
	IDLE	ERR_ID != 0	Set ENB = false
	CONF	ERR_ID == 0	
CONF	BUSY		Saturate Inertia, BwScale, and Friction by the specified bounds. If saturation occurs, ERR_ID will be 1012, or 1013, 1014, or 1016.
BUSY			Generate control signal
	IDLE	RES == true OR ENB == false	Set ENB = false
	CONF	RES == false AND ENB == true AND changes in BwScale, Inertia, or Friction	

Copyright © 2013–2019, Texas Instruments Incorporated

3.5.2 SpinTAC Velocity Move

SpinTAC Velocity Move generates motion profiles satisfying the specified maximum jerk, acceleration, and velocity values. The relationships among the generated reference signals are: the velocity reference is the derivative of position reference; the acceleration reference is the derivative of the velocity reference; and the jerk reference is the derivative of the acceleration.

3.5.2.1 SpinTAC Velocity Move Interface

The interfaces and functions of SpinTAC Velocity Move are shown in Figure 3-7.

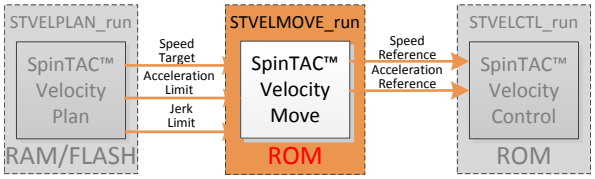


Figure 3-7. SpinTAC Velocity Move Interfaces

Table 3-7 lists the interface parameters for SpinTAC Velocity Move.

Table 3-7. SpinTAC Velocity Move Interfaces

Signal Type	Structure Member Name	Data Type	Description	Value Range	Units
Config	cfg.Axis	ST_Axis_e	SpinTAC Move Axis ID	{ST_AXIS0, ST_AXIS1}	
	cfg.CurveType	ST_MoveCurve Type_e	Curve type	{ST_MOVE_CUR_TRAP, ST_MOVE_CUR_SCRV, ST_MOVE_CUR_STCRV}	
	cfg.T_sec	_iq24	Sampling time	(0 , 0.01]	s
	cfg.VelMax	_iq24	Maximum velocity of the system	(0 , 1]	pu / s
	cfg.AccMax	_iq24	Maximum acceleration of the system	[0.001 , 120]	pu / s^2
	cfg.JrkMax	_iq20	Maximum jerk of the system	[0.0005, 2000]	pu / s^3
	cfg.VelStart	_iq24	Velocity start value	[-cfg.VelMax , cfg.VelMax]	pu / s
	cfg.IgnoreLimitErrors	bool	If a profile bound is set outside the valid value range, this will saturate the profile limit to within the valid value range	false: provide an error code & do not generate a profile; true: saturate profile limit & generate a profile	
Message	msg.ProTime_tick	uint_32	Profile time in sample time counts		Sample Counts
	msg.Acc	_iq24	Maximum acceleration of the profile		pu / s^2
	msg.Jrk	_iq20	Maximum jerk of the profile		pu / s^3
Inputs	VelEnd	_iq24	Velocity end value	[-cfg.VelMax , cfg.VelMax]	pu / s
	AccLim	_iq24	Acceleration limit	[0.001 , cfg.AccMax]	pu / s^2
	JrkLim	_iq20	Jerk limit	[0.0005, cfg.JrkMax]	pu / s^3
Control	ENB	bool	Enable bit	false: profile done or disabled; true: enable and run	
	RES	bool	Reset bit	false: not reset; true: reset ERR_ID, and hold profile outputs as previous values	
	TST	bool	Profile configuration test bit	false: not test; true: test profile configuration	

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-7. SpinTAC Velocity Move Interfaces (continued)

Signal Type	Structure Member Name	Data Type	Description	Value Range	Units
Outputs	VelRef	_iq24	Velocity reference		pu / s
	AccRef	_iq24	Acceleration reference		pu / s^2
	JrkRef	_iq20	Jerk reference		pu / s^3
Info	STATUS	ST_MoveStatus_e	Status information	{ST_MOVE_IDLE, ST_MOVE_INIT, ST_MOVE_CONF, ST_MOVE_BUSY, ST_MOVE_HALT}	
	DON	bool	Profile done indicator	false: running; true: profile done or idle	
	ERR_ID	uint16_t	Error code	See Table 13-1	

For simplicity in SpinTAC Velocity Move, all velocity profiles use the configured acceleration for all moves. For instance, you could have a profile that would technically be decelerating, but SpinTAC Velocity Move would use the acceleration limit for that profile.

3.5.2.2 SpinTAC Velocity Move Run Function

The main function is STVELMOVE_run(ST_VELMOVE_Handle handle), where handle is a pointer to a specific ST_VelMove_t object. This function runs SpinTAC velocity move. This function must be run at the same rate as SpinTAC Velocity Control.

void **STVELMOVE_run**(ST_VELMOVE_Handle handle)

Parameters:

No.	Type	Parameters	Description
1	ST_VELMOVE_Handle	Handle	The pointer to a ST_VelMove_t object

The SpinTAC Velocity Move state transition map is shown in Figure 3-8.

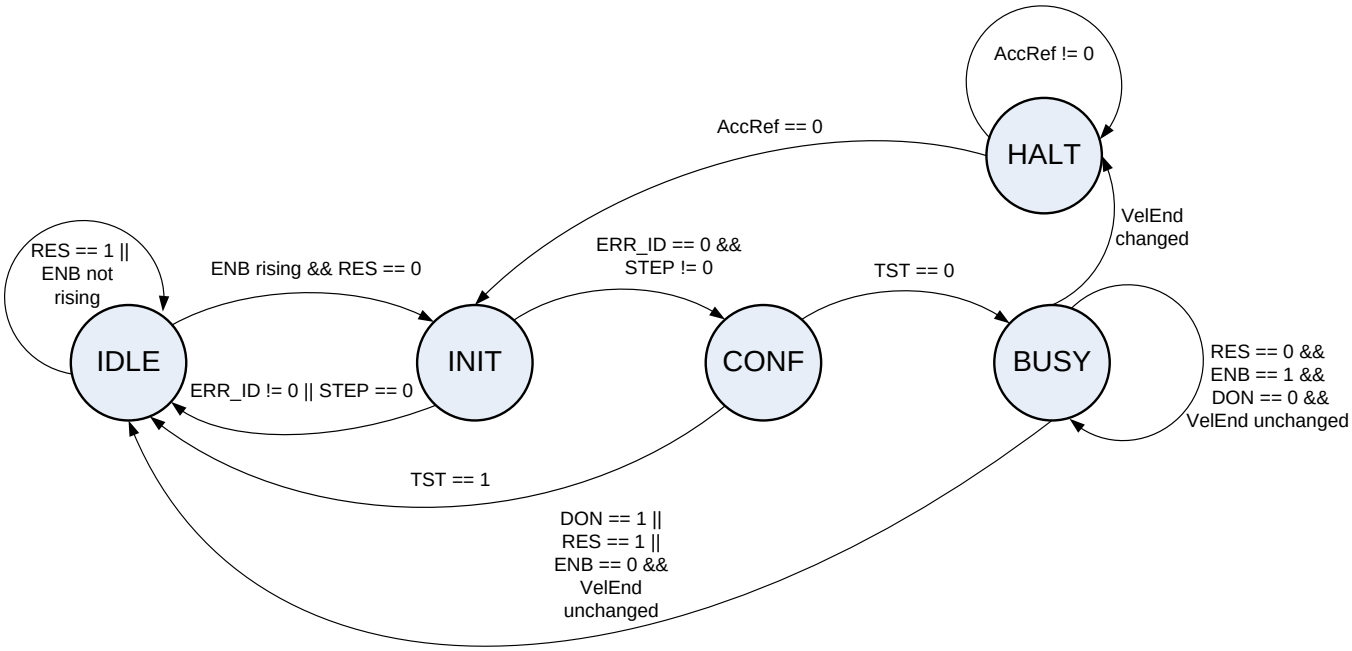


Figure 3-8. SpinTAC Velocity Move State Transition Map

Copyright © 2013–2019, Texas Instruments Incorporated

Note in Figure 3-8, the transitions from state IDLE to INIT, and then to CONF, happen within one sample time. Thus, the profile is generated at the sample time when the profile is enabled.

The states of the SpinTAC Velocity Move are described in Table 3-8.

Table 3-8. SpinTAC Velocity Move State Transition

From State	To State	Transition Condition[(1)(2)(3)(4)]	Action
IDLE			Keep IDLE status 1. Set ENB = false; 2. Set DON = true; 3. Hold the values for VelRef, AccRef, and JrkRef (The value of cfg.VelStart and VelEnd can only be set in the IDLE state)
	INIT	RES == false AND ENB is on rising edge	
INIT			Parameter validation 1. Validate the configuration parameters, including cfg.VelMax, cfg.VelStart, VelEnd, cfg.JrkMax, cfg.AccMax, JrkLim, AccLim, cfg.CUR_MOD, and cfg.T_sec. If any of the checked variables is invalid, ERR_ID will be nonzero; 2. Calculate velocity step, STEP
	IDLE	ERR_ID != 0 OR STEP == 0	Set ENB = false
	CONF	ERR_ID == 0 AND STEP != 0	
CONF			Determine the profile with the configured parameters
	IDLE	TST == true	Set the value of VelEnd back to the value of cfg.VelStart
	BUSY	TST == false	
BUSY			Produce the profile 1. Update references VelRef, AccRef, JrkRef at each sample time; 2. If the profile is finished, set DON as true
	IDLE	RES == true OR ENB == false OR DON == true	Set ENB = false
	HALT	VelEnd changed during profile	
HALT			Reduce AccRef to 0. Update AccRef, JrkRef at each sample time. Prepare to smoothly move to the new VelEnd.
	INIT	AccRef == 0	

(1) The RES signal provides the ability to place SpinTAC Velocity Move into reset.

(2) If RES is set to true, ENB will be set to false. Any current errors will be discarded. The current profile is discarded, AccRef is set to 0, VelRef is held as the value from previous sample time, and VelEnd and cfg.VelStart are set with the value of VelRef. Therefore, when RES is set back to false, a new profile can be started with the held over velocity reference.

(3) The ENB signal provides the start signal to SpinTAC Velocity Move. The ENB signal only functions when RES is set to false.

(4) The purpose of TST bit is to provide the profile information without generating trajectories. The information includes the profile time and actual maximums for acceleration and jerk. The TST signal is received by the function at the rising edge of ENB in the INIT state. If TST is true, it is in the test mode. In test mode, the profile output VelRef will keep the value of cfg.VelStart; AccRef will be 0, not influenced by VelEnd. After the test, the profile information (msg.ProTtime_tick, msg.Acc, and msg.Jrk) is output, VelEnd is set back to the value of cfg.VelStart, DON will be set to true, and ENB will be set to false.

3.5.3 SpinTAC Velocity Plan

SpinTAC Velocity Plan components provide the functionality to setup and run motion sequences determined by the user application.

3.5.3.1 SpinTAC Velocity Plan Interface

The interfaces and functions of SpinTAC Velocity Plan are shown in Figure 3-9 and described in Table 3-9.

Copyright © 2013–2019, Texas Instruments Incorporated

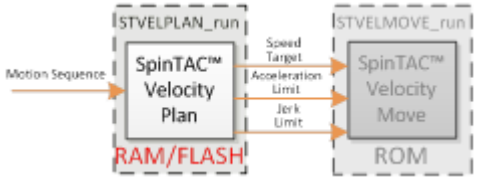


Figure 3-9. SpinTAC Velocity Plan Interfaces

Table 3-9. SpinTAC Velocity Plan Interfaces

Signal Type	Structure Member Name	Data Type	Description	Value Range	Units
Control	ENB	bool	Enable bit	false: disabled; true: enabled	
	RES	bool	Reset bit	false: not reset; true: reset	
Outputs	VelEnd	_iq24	Current velocity setpoint	[-VelMax , VelMax]	pu / s
	AccLim	_iq24	Current acceleration limit	[0.002 , AccMax]	pu / s^2
	JrkLim	_iq20	Current jerk limit	[0.001, JrkMax]	pu / s^3
Info	Timer_tick	int32_t	Time remaining in the current state		
	STATUS	ST_PlanStatus_e	Status information	{ST_PLAN_IDLE, ST_PLAN_INIT, ST_PLAN_CONF, ST_PLAN_BUSY}	
	CurState	uint16_t	Current state index	[0, StateNum)	
	CurTran	uint16_t	Current transition index	[0, TranNum)	
	FsmState	ST_PlanFsmState_e	Status to indicate if it is in a transition, or in a state, or waiting for a transition	{ ST_FSM_STATE_STAY, ST_FSM, _STATE_COND, ST_FSM _STATE_TRAN }	
	DON	bool	Plan done indicator	false: not done; true: done	
	ERR_ID	uint16_t	Plan function that caused the error	See Table 13-4	
	CfgError.ERR_idx	uint16_t	Component index where the error occurred		
	CfgError.ERR_code	uint16_t	Condition that caused error	See Table 13-4	

3.5.3.2 SpinTAC Velocity Plan Primary Functions

The primary function is STVELPLAN_run(ST_VELPLAN_Handle handle), where handle is a pointer to a specific ST_VelPlan_t object. This function runs SpinTAC Velocity Plan. This function can be run in the main loop of the program.

void **STVELPLAN_run**(ST_VELPLAN_Handle handle)

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The pointer to a ST_VelPlan_t object

The function for SpinTAC Velocity Plan that handles decrementing the state timer is STVELPLAN_runTick(ST_VELPLAN_Handle handle), where handle is a pointer to a specific ST_VelPlan_t object. This function decrements the state times for SpinTAC Velocity Plan. This function must be run at the same rate as SpinTAC Velocity Control.

Copyright © 2013–2019, Texas Instruments Incorporated

void **STVELPLAN_runTick**(ST_VELPLAN_Handle handle)

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The pointer to a ST_VelPlan_t object

The SpinTAC Velocity Plan state transition map is shown in Figure 3-10.

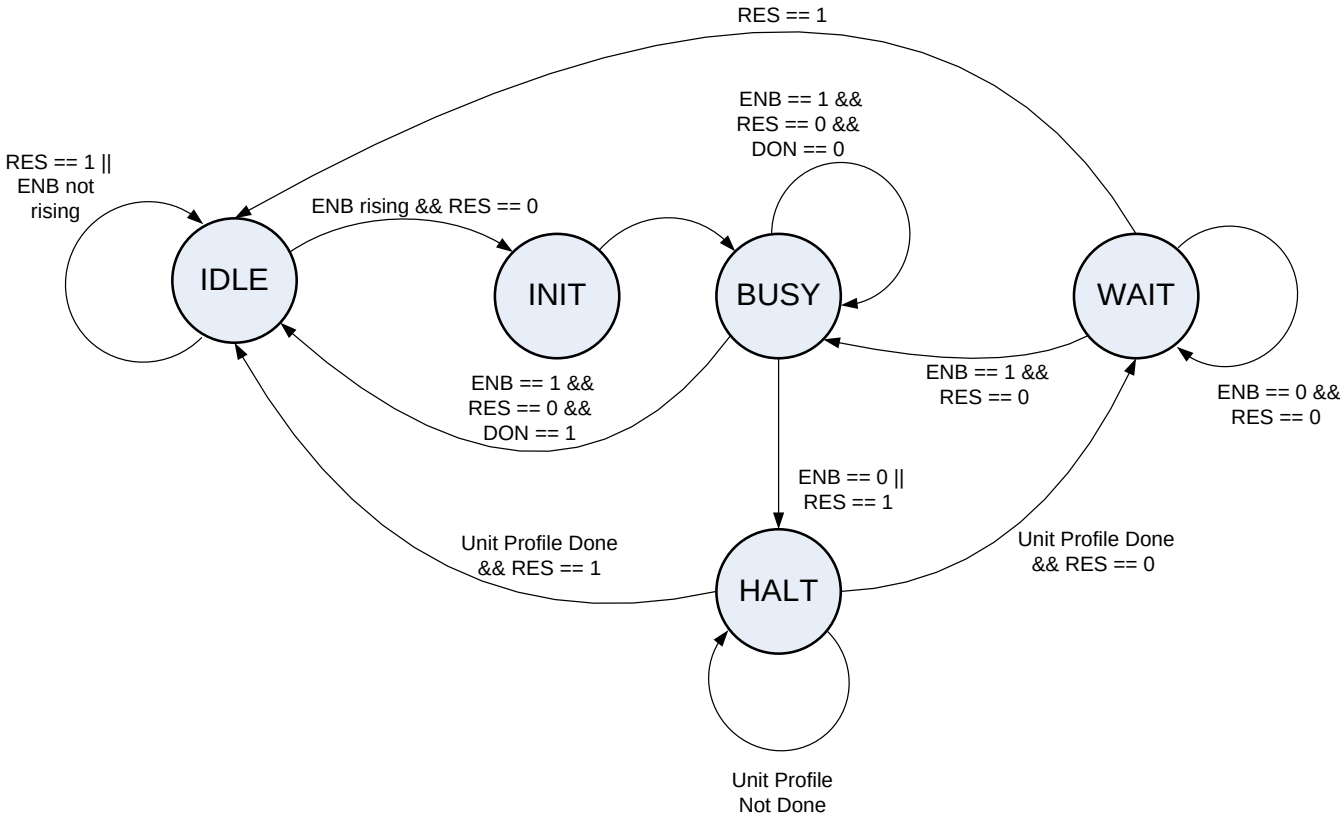


Figure 3-10. SpinTAC Velocity Plan State Transition Map

Note in Figure 3-10, the transitions from state IDLE to INIT, and then to BUSY, happen within one sample time.

The states of SpinTAC Velocity Plan are described in Table 3-10.

Table 3-10. SpinTAC Velocity Plan State Transition

From State	To State	Transition Condition[(1)(2)(3)(4)(5)]	Action
IDLE			Keep IDLE status 1. Set ENB = false; 2. Hold the values for VelEnd, AccLim, and JrkLim
	INIT	RES == false AND ENB is on rising edge	
INIT	BUSY		Parameter validation 1. Reset internal status; 2. Enter state 0 and execute the actions defined for entering state 0.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-10. SpinTAC Velocity Plan State Transition (continued)

From State	To State	Transition Condition[1][2][3][4][5]	Action
BUSY			Operate the plan
	IDLE	RES == false AND ENB == true AND DON == true	Set ENB = false
	HALT	RES == true OR ENB == false	Load the halt state profile configurations
HALT	IDLE	HALT state Timer up AND RES == true	Load the configurations of state 0
	WAIT	HALT state Timer up AND RES == false	
WAIT	IDLE	RES == true	Load the configurations of state 0
	HALT	ENB == true	Load the configurations of the last state

(1) The RES signal provides the ability to place SpinTAC Velocity Plan into reset.

(2) The ENB signal controls the operation of SpinTAC Velocity Plan when RES is false.

(3) If ENB is set to false when SpinTAC Velocity Plan is running, SpinTAC Velocity Plan will then send out the velocity setpoint and limits of the HALT state. When the unit profile is done, SpinTAC Velocity Plan will enter WAIT state, and can only continue the plan when ENB is set to true.

(4) If RES is set to true, ENB will be set to false. SpinTAC Velocity Plan will then send out the velocity setpoint and limits of state 0, and SpinTAC Velocity Plan enters IDLE state.

(5) Effectively, ENB functions as a pause/start button, while RES functions as stop.

Table 3-11 lists the functions that can be used to perform operations such as set, get, add, and delete configuration and runtime parameters of SpinTAC Velocity Plan. These functions are described in more detail in Section 3.5.9.

Table 3-11. SpinTAC Velocity Plan Additional Functions

Function Group	Function Name	Description
Initialization	STVELPLAN_init	Initialize SpinTAC Velocity Plan
Configuration	STVELPLAN_setCfg	Set the system and protection parameters
	STVELPLAN_setCfgArray	Setup configuration array
	STVELPLAN_setCfgHaltState	Set the parameters for the HALT state
	STVELPLAN_addCfgState	Add a new State
	STVELPLAN_addCfgVar	Add a new Variable
	STVELPLAN_addCfgCond	Add a new Condition that compares a Variable against static values
	STVELPLAN_addCfgVarCond	Add a new Condition that compares two Variables
	STVELPLAN_addCfgTran	Add a new Transition
	STVELPLAN_addCfgAct	Add a new Action
Runtime	STVELPLAN_run	Run SpinTAC Velocity Plan. Can run from main loop.
	STVELPLAN_runTick	Run SpinTAC Velocity Plan Timer function. Must run from ISR.
	STVELPLAN_setVar	Set the value of a Variable during runtime
	STVELPLAN_getVar	Get the value of a Variable during runtime
	STVELPLAN_reset	Reset SpinTAC Velocity Plan and configuration
	STVELPLAN_setUnitProfDone	Sets if the currently running profile is done

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-11. SpinTAC Velocity Plan Additional Functions (continued)

Function Group	Function Name	Description
Plan modification and debugging functions (Provide ability to modify SpinTAC Velocity Plan at runtime)	STVELPLAN_getCfgStateNum	Get the number of configured States
	STVELPLAN_getCfgVarNum	Get the number of configured Variables
	STVELPLAN_getCfgCondNum	Get the number of configured Conditions
	STVELPLAN_getCfgTranNum	Get the number of configured Transitions
	STVELPLAN_getCfgActNum	Get the number of configured Actions
	STVELPLAN_getCfg	Get the system & protection parameters
	STVELPLAN_getCfgHaltState	Get the parameters for the HALT state
	Each function with a suffix -Add has three other functions: -Del, -Set, and -Get to delete the item, to set the item, and to get the item respectively. These functions allow runtime modification of SpinTAC Velocity Plan.	

3.5.4 SpinTAC Velocity Identify

SpinTAC Velocity Identify estimates system inertia according to the applied torque profile and the measured velocity feedback.

Neglecting disturbances and non-linearities, simple motion systems can be described with Equation 2.

$$J\dot{v}(t) = -Bv(t) + u(t) \quad (2)$$

In Equation 2, v(t) and u(t) are the velocity and control input, respectively; J and B are the inertia ratio and friction coefficient, respectively.

SpinTAC Velocity Identify applies a continuous torque profile and estimates system inertia ratio with respect to the speed feedback. The estimated inertia and friction should be provided to SpinTAC controllers. SpinTAC Velocity Identify should be used to estimate the inertia for both velocity and position solutions.

3.5.4.1 SpinTAC Velocity Identify Interface

The interfaces of SpinTAC Velocity Identify are shown in Section 8.5.2.1.1 and described in Table 3-12.

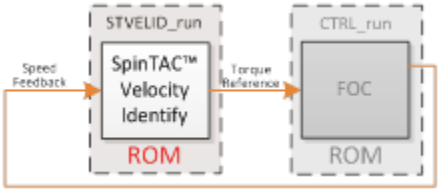


Figure 3-11. SpinTAC Velocity Identify Interfaces

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-12. SpinTAC Velocity Identify Interfaces and Parameters

Signal Type	Variable Name	Data Type	Description	Value Range	Unit
Config	cfg.T_sec	_iq24	Sampling time	(0 , 0.01]	s
	cfg.VelMax	_iq24	Maximum velocity of the system	(0 , 1]	pu / s
	cfg.OutMax	_iq24	Maximum velocity loop control signal	(0 , 1]	PU
	cfg.OutMin	_iq24	Minimum velocity loop control signal	[-1, 0)	PU
	cfg.VelPos	_iq24	Velocity positive value	(0 , cfg.VelMax]	pu / s
	cfg.OutPos	_iq24	Control signal positive value	(0 , cfg.OutMax]	PU
	cfg.OutNeg	_iq24	Control signal negative value	[cfg.OutMin, 0)	PU
	cfg.LpfTime_tick	int16_t	Feedback signal low pass filter time constant	[1 , 100]	Sample Counts
	cfg.TimeOut_sec	_iq24	Maximum time allowed for inertia identification	[1, 10]	s
	cfg.RampTime_sec	_iq24	Time allowed for control signal to reach 1.0 PU during inertia estimation process	[cfg.T, sec 25]	s
Input	VelFdb	_iq24	Velocity feedback		pu / s
Control	ENB	bool	Enable bit	false: disabled; true: enabled	
	RES	bool	Reset bit	false: not reset; true: reset ERR_ID, and hold Out as 0	
Output	Out	_iq24	Torque signal		PU
Result	InertiaEst	_iq24	Estimated inertia		PU· s^2 / pu
	FrictionEst	_iq24	Estimated friction coefficient		PU· s / pu
Info	STATUS	ST_VelIdStatus_e	Status	{ST_VEL_ID_IDLE, ST_VEL_ID_INIT, ST_VEL_ID_BUSY }	
	DON	bool	Identification completed indicator	false: running or disabled; true: Identification complete	
	ERR_ID	uint16_t	Error ID	See Table 7-1	

3.5.4.2 SpinTAC Velocity Identify Run Function

The primary function is STVELID_run(ST_VELID_Handle handle), where handle is the pointer to a specific ST_VelId_t object. This function needs to be called at the configured sample rate.

void **STVELID_run**(ST_VELID_Handle handle)

Parameters:

No.	Type	Parameters	Description
1	ST_VELID_Handle	handle	The pointer to a ST_VelId_t object

Copyright © 2013–2019, Texas Instruments Incorporated

The state transition map is shown in Figure 3-12.

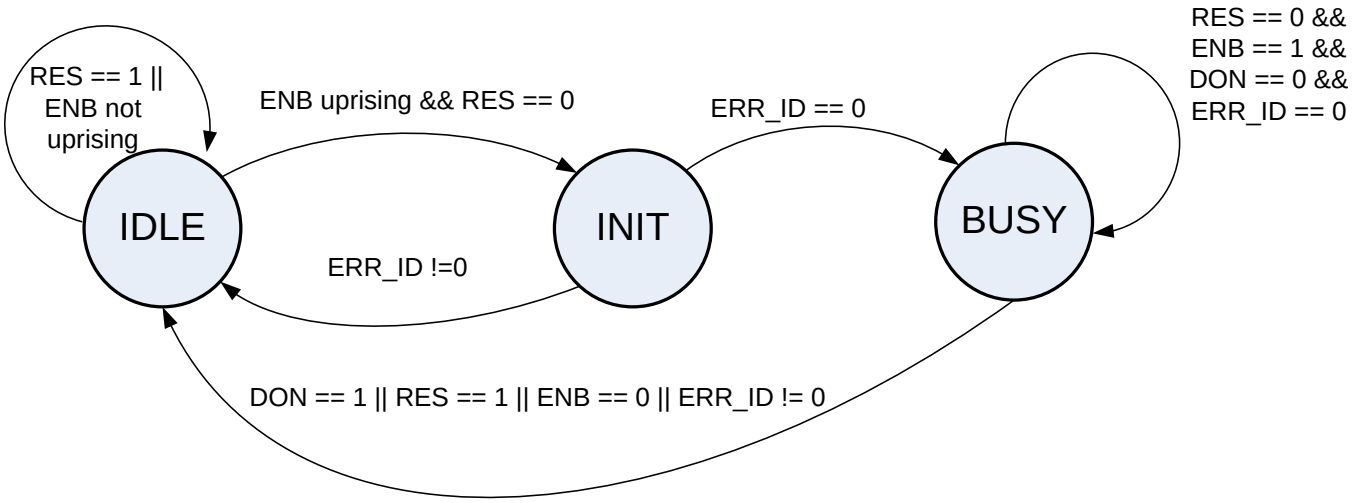


Figure 3-12. SpinTAC Velocity Identify State Transition Map

The state transitions of Inertia Identify are described in Table 3-13.

Table 3-13. SpinTAC Velocity Identify State Transition

From State	To State	Transition Condition	Action
IDLE			1. Set ENB = false; 2. Hold Out as 0 Parameter validation 1. Validate the configuration parameters, including cfg.VelMax, cfg.OutMax, cfg.OutMin, cfg.LpfTime_tick, cfg.TimeOut_sec, and cfg.T_sec, if any of the checked variables is invalid, ERR_ID will be nonzero;
	INIT	RES == false AND ENB is on rising edge AND ERR_ID == 0	
INIT			Parameter validation 1. Validate the parameters, including cfg.VelPos, cfg.OutPos, cfg.OutNeg, and cfg.RampTime_sec, if any of the checked variables is invalid, ERR_ID will be nonzero;
	IDLE	ERR_ID != 0	Set ENB = false
	BUSY	ERR_ID == 0	
BUSY			Produce torque profile and monitor velocity feedback 1. Produce torque profile and monitor velocity feedback at each sample time; 2. If the profile is finished, set DON as true; if the profile times out, ERR_ID is set to 2004; 3. If the estimated inertia is not positive, ERR_ID is set to 2003; 4. If the motor stops during the test, ERR_ID is set to 2006;
	IDLE	RES == true OR ENB == false OR DON == true OR ERR_ID != 0	1. If RES == true OR ENB == FLASE, Out will steadily approach zero, and then wait a few seconds to settle. ERR_ID will be set to 2005. Set ENB = false

Copyright © 2013–2019, Texas Instruments Incorporated

3.5.5 SpinTAC Position Convert

The SpinTAC Position Convert is required for systems that use an encoder for feedback. It calculates position and velocity signals from the position encoder feedback. This component is used to provide the velocity feedback signal [pu/s] for SpinTAC Velocity Identify and SpinTAC Velocity Control when using an encoder to provide electrical angle to the FOC. It is also used for all position solutions to convert the encoder electrical angle into a mechanical angle. This component is not required for systems that use the FAST sensorless estimator. SpinTAC Position Convert will also provide an estimation of the Slip Speed for AC induction motors.

3.5.5.1 SpinTAC Position Convert Interfaces

The interfaces and functions of SpinTAC Position Convert are shown in Figure 3-13 and described in Table 3-14.

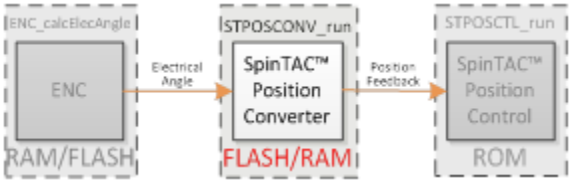


Figure 3-13. SpinTAC Position Convert Interfaces

Table 3-14. SpinTAC Position Convert Interfaces and Parameters

Signal Type	Structure Member Name	Data Type	Description	Value Range	Unit
Config	cfg.T_sec	_iq24	Sampling time	(0 , 0.01]	s
	cfg.ROMax_erev	_iq24	Maximum bound for electrical revolution [ERev]	[0 , 16]	ERev
	cfg.ROMin_erev	_iq24	Minimum bound for electrical revolution [ERev]	[-16, 0]	ERev
	cfg.erev_TO_pu_ps	_iq24	Conversion ratio from electrical revolution [ERev] to velocity user unit [pu/s]	(0 , 0.01]	pu / s / ERev
	cfg.PolePairs	_iq24	Conversion ratio from mechanical revolution to electrical revolution [Pole Pairs]	[1, 32]	
	cfg.ROMax_mrev	_iq24	Position Rollover Bound	[2, 100]	MRev
	cfg.LpfTime_tick	int16	Low-pass filter time constant [ISR ticks]	[1 , 100]	Sample counts
	cfg.SampleTimeOverTimeConst	_iq24	Scalar value used in the Slip Compensator (ACIM Only)	[0, 128.0)	
	cfg.OneOverFreqTimeConst	_iq24	Scalar value used in the Slip Compensator (ACIM Only)	[0, 128.0)	
Input	Pos_erev	_iq24	Saw-tooth electrical angle signal	[cfg.ROMin_erev, cfg.ROMax_erev)	ERev
	Id	_iq24	Id Current Feedback (ACIM Only)	[-1.0, 1.0]	PU
	Iq	_iq24	Iq Current Feedback (ACIM Only)	[-1.0, 1.0]	PU
Control	ENB	bool	Enable bit	false: disabled; true: enabled	
	RES	bool	Reset bit	false: not reset; true: reset	

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-14. SpinTAC Position Convert Interfaces and Parameters (continued)

Signal Type	Structure Member Name	Data Type	Description	Value Range	Unit
Outputs	Vel	_iq24	Calculated velocity in user unit (unfiltered)	[-1, 1]	pu / s
	VelLpf	_iq24	Filtered velocity in user unit	[-1, 1]	pu / s
	Pos_mrev	_iq24	Saw-tooth mechanical angle signal	[-cfg.ROMax_mrev, cfg.ROMax_mrev]	MRev
	PosROCounts	int32_t	Position roll over counts		
	SlipVel	_iq24	Speed of magnetic slip (ACIM Only)	[-1.0, 1.0]	ERev / s
Info	STATUS	ST_PosConvStatus_e	Status information	{ST_POS_CONV_IDLE, ST_POS_CONV_INIT, ST_POS_CONV_BUSY}	
	ERR_ID	uint16_t	Error code	See Table 18-2	

3.5.5.2 SpinTAC Position Convert Run Function

The primary function is STPOSCONV_run(ST_POSCONV_Handle handle), where handle is the pointer to a specific ST_PosConv_t object, this handle needs to be established by the initialize function STPOSCONV_init.

void **STPOSONV_run**(ST_POSCONV_Handle handle)

Parameters:

No.	Type	Parameters	Description
1	ST_POSCONV_Handle	handle	The pointer to a ST_PosConv_t object

The state transition map is shown in Figure 3-14.

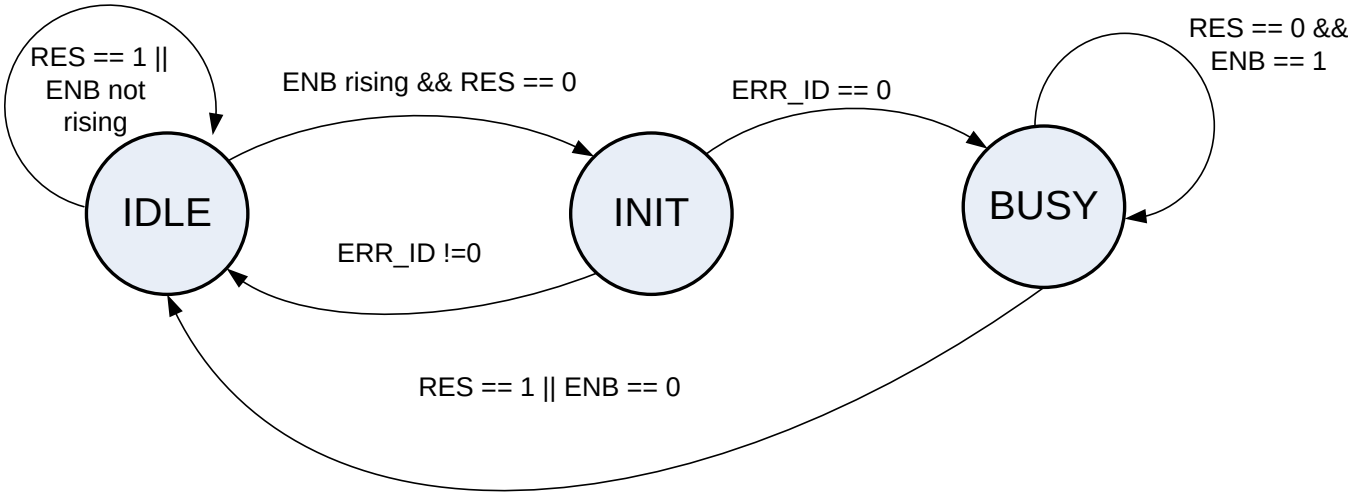


Figure 3-14. SpinTAC Position Convert State Transition Map

The state transitions of the SpinTAC Position Convert are described in Table 3-15.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-15. SpinTAC Position Convert State Transition

From State	To State	Transition Condition	Action
IDLE			1. Set ENB = false; 2. Hold Vel, VelLpf, Pos_mrev, and PosROCounts as 0
	INIT	RES == false AND ENB is on rising edge	
INIT			Parameter validation 1. Validate the configuration parameters, including cfg.ROMax_erev, cfg.ROMin_erev, cfg.erev_TO_pu_ps, cfg.PolePairs, cfg.ROMax_mrev, cfg.LpfTime_ticks, and cfg.T_sec, if any of the checked variables is invalid, ERR_ID will be nonzero;
	IDLE	ERR_ID != 0	Set ENB = false
	BUSY	ERR_ID == 0	
BUSY			Calculate sawtooth position signal in [MRev] and velocity signal in [pu/s]
	IDLE	RES == true OR ENB == false	Set ENB = false

3.5.6 SpinTAC Position Control

The SpinTAC Position Control is a cascaded controller that controls the position and velocity loops for a motion system. It allows developers to tune both loops with a single tuning parameter. Similarly to SpinTAC Velocity Control, SpinTAC Position Control rejects external disturbances, which is represented by an unknown, nonlinear term in the control equations. The tuning process is also simplified, like in SpinTAC Velocity Control, via a parameterization method that enables high-performance control of dynamical systems using a single tuning parameter: bandwidth. This single parameter is used to tune both the velocity and position loops of SpinTAC Position Control.

3.5.6.1 SpinTAC Position Control Interface

The interfaces and functions of SpinTAC Position Control are shown in Figure 3-15.

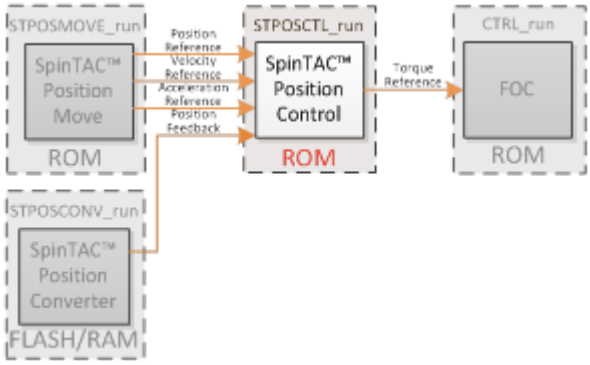


Figure 3-15. SpinTAC Position Control Interfaces

Note: in SpinTAC Position Control, the inertia, cfg.Inertia, can be obtained by SpinTAC Velocity Identify. The switch cfg.RampDist can be enabled if the dominant disturbance is ramp type. The only tuning parameter of the controller is the bandwidth BwScale.

Table 3-16 lists the interface parameters for SpinTAC Position Control.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-16. SpinTAC Position Control Interface Parameters

Signal Type	Structure Member Name	Data Type	Description	Value Range	Unit
Config	cfg.Axis	ST_Axis_e	SpinTAC Control Axis ID	{ST_AXIS0, ST_AXIS1}	
	cfg.T_sec	_iq24	Sampling time	(0 , 0.01]	s
	cfg.InertiaMax	_iq24	Maximum system inertia	(0, 100]	PU · s2 / pu
	cfg.InertiaMin	_iq24	Minimum system inertia	(0, cfg.InertiaMax]	PU · s2 / pu
	cfg.OutMax	_iq24	Maximum control signal	(0 , 1]	PU
	cfg.OutMin	_iq24	Minimum control signal	[-1, 0)	PU
	cfg.VelMax	_iq24	Maximum velocity reference signal	(0 , 1]	pu / s
	cfg.ROMax_mrev	_iq24	Position rollover bound	[2, 100]	MRev
	cfg.mrev_TO_pu	_iq24	Conversion ratio from mechanical revolution [MRev] to [pu]	[0.002, 1]	pu / MRev
	cfg.PosErrMax_mrev	_iq24	Maximum allowable position error	(0, cfg.ROMax_mrev /2]	MRev
	cfg.RampDist	bool	Reject ramp disturbance	false: disabled; true: enabled	
	cfg.BwScaleMax	_iq24	Bandwidth upper limit	[0.01, min(50, 0.005/ cfg.T_sec)]	
	cfg.BwScaleMin	_iq24	Bandwidth lower limit	[0, cfg.BwScaleMax]	
	cfg.FiltEn	bool	Enable feedback low pass filter	false: disabled; true: enabled	
Inputs	PosRef_mrev	_iq24	Position reference signal	[-cfg.ROMax_mrev, cfg.ROMax_mrev]	MRev
	VelRef	_iq24	Velocity reference signal	[-cfg.VelMax, cfg.VelMax]	pu / s
	AccRef	_iq24	Acceleration reference signal	[cfg.OutMin, cfg.OutMax]	pu / s2
	PosFdb_mrev	_iq24	Position feedback signal	[-cfg.ROMax_mrev, cfg.ROMax_mrev)	MRev
	Inertia	_iq24	System inertia	[cfg.InertiaMin, cfg.InertiaMax]	PU · s2 / pu
	Friction	_iq24	System friction	[0, 5]	PU · s2 / pu
Tuning	BwScale	_iq24	Controller bandwidth scale	[cfg.BwScaleMin, cfg.BwScaleMax]	
Control	ENB	bool	Enable bit	false: disabled; true: enabled	
	RES	bool	Reset bit	false: not reset; true: reset ERR_ID, and hold Out as 0	
Output	Out	_iq24	Control output	[cfg.OutMin, cfg.OutMax]	PU
Info	Bw_radps	_iq20	Controller bandwidth		rad/s
	STATUS	ST_CtlStatus_e	Status information	{ST_CTL_IDLE, ST_CTL_INIT, ST_CTL_CONF, ST_CTL_BUSY}	
	ERR_ID	uint16_t	Error code	See Table 12-3	
	PosErr_mrev	_iq24	Position error		MRev

3.5.6.2 SpinTAC Position Control Run Function

The primary function is STPOSCTL_run(ST_POSCTL_Handle handle), where handle is the pointer to a specific ST_PosCtl_t object, this handle needs to be established by the initialize function STPOSCTL_init.

void **STPOSCTL_run**(ST_POSCTL_Handle handle)

Parameters:

Copyright © 2013–2019, Texas Instruments Incorporated

No.	Type	Parameters	Description
1	ST_POSCTL_Handle	Handle	The pointer to a ST_PosCtl_t object

The SpinTAC Position Control state transition map is shown in Figure 3-16.

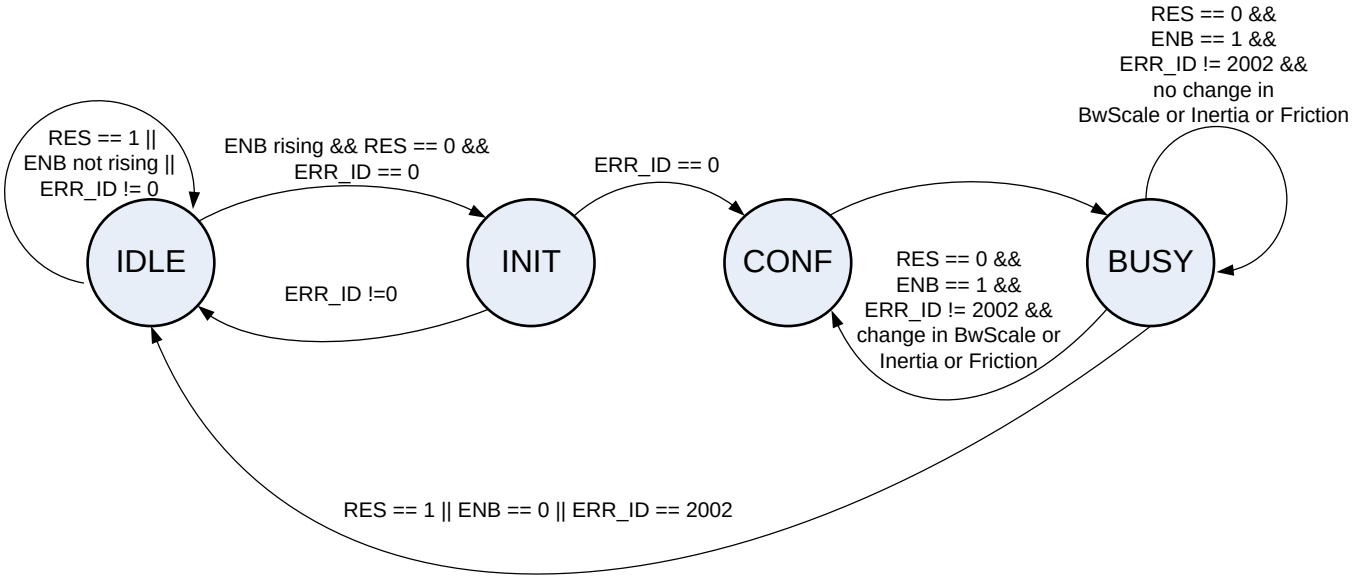


Figure 3-16. SpinTAC Position Control State Transition Map

Note in Figure 3-16, the transitions from state IDLE to INIT, INIT to CONF, and CONF to BUSY, happen within one sample time. Therefore, the controller works directly at the sample time when it is enabled.

The state transitions of SpinTAC Position Control are described in Table 3-17.

Table 3-17. SpinTAC Position Control State Transition

From State	To State	Transition Condition	Action
IDLE			1. Set ENB = false; 2. Hold Out as zero
	INIT	RES == false AND ENB is on rising edge AND ERR_ID == 0	
INIT			Parameter validation 1. Validate the configuration parameters, including cfg.VelMax, cfg.OutMax, cfg.OutMin, cfg.ROMax_mrev, cfg.mrev_TO_pu, cfg.PosErrMax_mrev, cfg.InertiaMax, cfg.InertiaMin, cfg.BwScaleMax, cfg.BwScaleMin, cfg.DistType, and cfg.T_sec. If any of the checked variables is invalid, ERR_ID will be nonzero
	IDLE	ERR_ID != 0	Set ENB = false
	CONF	ERR_ID == 0	
CONF	BUSY		Saturate Inertia and BwScale by the specified bounds. If saturation occurs, ERR_ID will be 1012, or 1013, or 1014, or 1015.
BUSY			1. Generate control signal. 2. If PosErr_mrev exceeds cfg.PosErrMax_mrev, ERR_ID =2002.
	IDLE	RES == true OR ENB == false OR ERR_ID == 2002	Set ENB = false
	CONF	RES == false AND ENB == true AND changes in BwScale or Inertia AND ERR_ID != 2002	

Copyright © 2013–2019, Texas Instruments Incorporated

3.5.7 SpinTAC Position Move

SpinTAC Position Move provides two modes: velocity-controlled position profile and position-controlled position profile. The former mode is suited for ramp signals and profiles switching among different velocities. The second mode is best suited for point-to-point position moves. cfg.ProfileType is used to switch between the two modes, which can only be adjusted at the rising edge of the ENB signal. Mode switching requires cfg.VelStart set to zero and STATUS in the IDLE state.

3.5.7.1 Position Move Interface

The interfaces and functions of SpinTAC Position Move are shown in Figure 3-17.

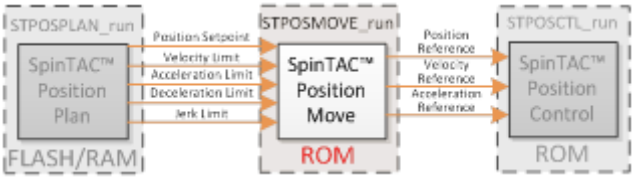


Figure 3-17. SpinTAC Position Move Interfaces

Table 3-18 lists the interface parameters for SpinTAC Position Move.

Table 3-18. SpinTAC Position Move Interfaces

Signal Type	Structure Member Name	Data Type	Description	Value Range	Unit
Config	cfg.Axis	ST_Axis_e	SpinTAC Move Axis ID	{ST_AXIS0, ST_AXIS1}	
	cfg.ProfileType	ST_PosMoveProfiletype_e	Profile mode (velocity-controlled or position-controlled)	{ST_POS_MOVE_VEL_TYPE, ST_POS_MOVE_POS_TYPE}	
	cfg.CurveType	ST_MoveCurveType_e	Curve type	{ST_MOVE_CUR_TRAP, ST_MOVE_CUR_SCRV, ST_MOVE_CUR_STCRV}	
	cfg.T_sec	_iq24	Sampling time	(0 , 0.01]	s
	cfg.ROMax_mrev	_iq24	Position rollover bound	[2, 100]	MRev
	cfg.mrev_TO_pu	_iq24	Conversion ratio from mechanical revolution to pu	[0.002, 1]	Pu / MRev
	cfg.VelMax	_iq24	Maximum velocity of the system	(0 , 1]	pu / s
	cfg.AccMax	_iq24	Maximum acceleration of the system	[0.001 , 120]	pu / s2
	cfg.DecMax	_iq24	Maximum deceleration of the system	[0.001 , 120]	pu / s2
	cfg.JrkMax	_iq20	Maximum jerk of the system	[0.0005, 2000]	pu / s3
	cfg.VelStart	_iq24	Velocity start value	[-cfg.VelMax , cfg.VelMax]	pu / s
	cfg.PosStart_mrev	_iq24	Position start value	[-cfg.ROMax, cfg.ROMax]	MRev
	cfg.IgnoreLimitErrors	bool	If a profile bound is set outside the valid value range, this will saturate the profile limit to within the valid value range	false: provide an error code & do not generate a profile; true: saturate profile limit & generate a profile	

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-18. SpinTAC Position Move Interfaces (continued)

Signal Type	Structure Member Name	Data Type	Description	Value Range	Unit
Message	msg.ProTime_tick	uint32_t	Profile time within 1 million counts		Sample Counts
	msg.ProTime_mtick	uint32_t	Profile time million counts		Million Sample Counts
	msg.Vel	_iq24	Maximum velocity of the profile		pu / s
	msg.Acc	_iq24	Maximum acceleration of the profile		pu / s2
	msg.Dec	_iq24	Maximum deceleration of the profile		pu / s2
	msg.Jrk	_iq20	Maximum jerk of the profile		pu / s3
	msg.PosStepMax_mrev	_iq24	Maximum position step		MRev
Inputs	PosStepInt_mrev	int32_t	Position step integer value	[-2147483647, 2147483647]	MRev
	PosStepFrac_mrev	_iq24	Position step fraction value	(-1, 1)	MRev
	VelLim	_iq24	Velocity limit	(0, cfg.VelMax]	pu / s
	AccLim	_iq24	Acceleration limit	[0.001 , cfg.AccMax]	pu / s2
	DecLim	_iq24	Deceleration limit	[0.001 , cfg.DecMax]	pu / s2
	JrkLim	_iq20	Jerk limit	[0.0005, cfg.JrkMax]	pu / s3
	VelEnd	_iq24	Velocity end value	[-cfg.VelMax , cfg.VelMax]	pu / s
Control	ENB	bool	Enable bit	false: profile done or disabled; true: enable and run	
	RES	bool	Reset bit	false: not reset; true: reset ERR_ID, and hold profile outputs as previous values	
	TST	bool	Profile configuration test bit	false: not test; true: test profile configuration	
Info	STATUS	ST_MoveStatus_e	Status information	{ST_MOVE_IDLE, ST_MOVE_INIT, ST_MOVE_CONF, ST_MOVE_BUSY, ST_MOVE_HALT}	
	DON	bool	Profile done indicator	false: running; true: profile done or idle	
	ERR_ID	uint16_t	Error code	See Table 13-2	
Outputs	PosRollOver	int32_t	Position rollover counts		
	PosRef_mrev	_iq24	Position reference		MRev
	VelRef	_iq24	Velocity reference		pu / s
	AccRef	_iq24	Acceleration reference		pu / s2
	JrkRef	_iq20	Jerk reference		pu / s3

3.5.7.2 SpinTAC Position Move Run Function

The SpinTAC Position Move function is STPOSMOVE_run(ST_POSMOVE_Handle handle), where handle is a pointer to a specific ST_PosMove_t object, this handle needs to be established by the initialize function STPOSMOVE_init.

void **STPOSMOVE_run**(ST_POSMOVE_Handle handle)

Parameters:

Copyright © 2013–2019, Texas Instruments Incorporated

No.	Type	Parameters	Description
1	ST_POSMOVE_Handle	Handle	The pointer to a ST_PosMove_t object

The SpinTAC Position Move_t state transition map is shown in Figure 3-18.

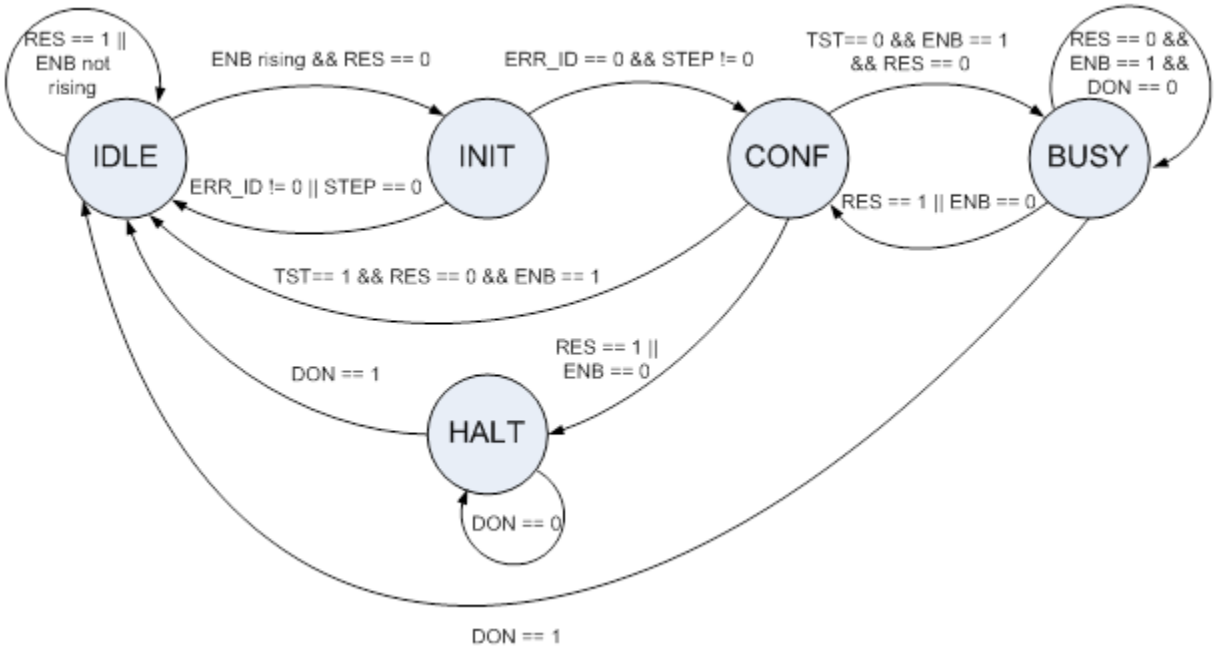


Figure 3-18. SpinTAC Position Move State Transition Map

Note in Figure 3-18, the transitions from state IDLE to INIT, and then to CONF, happen within one sample time. Thus, the profile is generated at the same sample time that the profile was enabled.

The states of SpinTAC Position Move are described in
Changed Section 13.1, InstaSPIN-MOTION Profile Generation.

Table 3-19. Position Move State Transition

From State	To State	Transition Condition[(1)(2)(3)(4)]	Action
IDLE			Keep IDLE status 1. Set ENB = false; 2. Set DON = false; 3. Hold the values for PosRef, VelRef, AccRef, and JrkRef (The value of cfg.PosStart_mrev can only be set in the IDLE state.)
	INIT	RES == false AND ENB is on rising edge	
INIT			Parameter validation 1. Validate the configuration parameters, including cfg.PosStart_mrev, PosStepInt_mrev, PosStepFrac_mrev, cfg.VelStart, VelEnd, cfg.VelMax, cfg.AccMax, cfg.DecMax, cfg.JrkMax, AccLim, DecLim, JrkLim, cfg.CurveType, cfg.ProfileType, and cfg.T_sec. If any of the checked variables is invalid, ERR_ID will be nonzero; 2. Calculate profile step STEP (velocity step if cfg.ProfileType == ST_POS_MOVE_VEL_TYPE, or position step if cfg.ProfileType == ST_POS_MOVE_POS_TYPE).
	IDLE	ERR_ID != 0 OR STEP == 0	Set ENB = false
	CONF	ERR_ID == 0 AND STEP != 0	

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-19. Position Move State Transition (continued)

From State	To State	Transition Condition[(1)(2)(3)(4)]	Action
CONF			Determine the profile with the configured parameters
	IDLE	TST == true AND ENB == true AND RES == false	In test mode, no profile is produced 1. If cfg.ProfileType == ST_POS_MOVE_POS_TYPE, set PosStepInt_mrev and PosStepFrac_mrev as 0; if cfg.ProfileType == ST_POS_MOVE_VEL_TYPE, set VelEnd back to the value of cfg.VelStart
	BUSY	TST == false AND ENB == true AND RES == false	
	HALT	RES == true OR ENB == false	
BUSY			Produce the profile 1. Update references PosRef_mrev, VelRef, AccRef, JrkRef at each sample time; 2. If the profile is finished, DON = true
	IDLE	RES == false AND ENB == true AND DON == true	Set ENB = false
	CONF	RES == true OR ENB == false	Configure a deceleration profile 1. Set cfg.ProfileType = ST_POS_MOVE_VEL_TYPE, VelEnd = 0, AccLim = cfg.AccMax, cfg.DecMax, JrkLim = cfg.JrkMax
HALT			Generate a deceleration profile 1. Update references PosRef, VelRef, AccRef, JrkRef at each sample time; 2. If the deceleration profile is finished, DON = True
	IDLE	DON == true	Set ENB = false
	HALT	DON == false	

(1) The RES signal provides the ability to place SpinTAC Position Move into reset.

(2) If RES is set to true, ENB will be set to false. Any current errors will be discarded. SpinTAC Position Move will then generate a deceleration profile to stop all motion of the axis.

(3) The ENB signal provides the start signal to SpinTAC Position Move. The ENB signal only functions when RES is false.

(4) The purpose of the TST bit is to provide the profile information without actually generating trajectories. The information includes the profile time and actual maximums for velocity, acceleration and jerk.
TST signal is received by the function at the rising edge of ENB in the INIT state. If TST is true, it operates in test mode. In test mode, the profile output PosRef_mrev will keep the value of cfg.PosStart_mrev; AccRef will be 0, not influenced by PosStepInt_mrev and PosStepFrac_mrev. After the test, the profile information (msg.ProTime_tick, msg.ProTime_mtick, msg.Vel, msg.Acc, and msg.Jrk) is output, DON will be set to true, and ENB will be set to false.

3.5.8 SpinTAC Position Plan

SpinTAC Position Plan provides the functionality to setup and run position sequences determined by the user application.

3.5.8.1 Position Plan Interface

The interfaces and functions of SpinTAC Position Plan are shown in Figure 3-19.

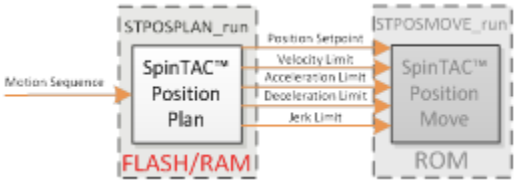


Figure 3-19. SpinTAC Position Plan Interfaces

Table 3-20 lists the interface parameters for SpinTAC Position Plan.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-20. SpinTAC Position Plan Interfaces

Signal Type	Structure Member Name	Data Type	Description	Value Range	Unit
Control	ENB	bool	Enable bit	false: disabled; true: enabled	
	RES	bool	Reset bit	false: not reset; true: reset	
Outputs	PosStepInt_mrev	uint32_t	Current Position Step command integer part	[-2147483647, 2147483647]	MRev
	PosStepFrac_mrev	_iq24	Current Position Step command fraction part	(-1, 1)	MRev
	VelLim	_iq24	Current velocity limit	(0 , VelMax]	pu / s
	AccLim	_iq24	Current acceleration limit	[0.001 , AccMax]	pu / s2
	DecLim	_iq24	Current deceleration limit	[0.001 , DecMax]	pu / s2
	JrkLim	_iq20	Current jerk limit	[0.0005, JrkMax]	pu / s3
	Timer_tick	uint32_t	Time remaining in the current state		Sample Counts
Info	STATUS	ST_PlanStatus_e	Status information	{ST_PLAN_IDLE, ST_PLAN_INIT, ST_PLAN_BUSY, ST_PLAN_HALT, ST_PLAN_WAIT}	
	CurState	unit16_t	Current state index	[0, StateNum)	
	CurTran	unit16_t	Current transition index	[0, TranNum)	
	FsmState	ST_PlanFsmState_e	Status to indicate if it is in a transition, or in a state, or waiting for a transition	{ST_FSM _STATE_STAY, ST_FSM _STATE_COND, ST_FSM_STATE_TRAN }	
	Timer_tick	uint32_t	Time remaining in the current state		Sample Counts
	ERR_ID	uint16_t	Error code	See Table 13-6	
	DON	bool	Plan done indicator	false: not done; true: done	
	CfgError.ERR_idx	uint16_t	Index where the error occurred		
	CfgError.ERR_code	uint16_t	Condition that caused the error	See Table 13-6	

3.5.8.2 SpinTAC Position Plan Primary Functions

The primary function is STPOSPLAN_run(ST_POSPLAN_Handle handle), where handle is a pointer to a specific ST_PosPlan_t object, this handle needs to be established by the initialize function ST_POSPLAN_init. This function can be called from the main-loop of the project.

void **STPOSPLAN_run**(ST_POSPLAN_Handle handle)

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The pointer to a ST_PosPlan_t object

The ISR function is STPOSPLAN_runTick(ST_POSPLAN_Handle handle), where handle is a pointer to a specific ST_PosPlan_t object. This function handles the time-critical code of ST_PosPlan. This function must be called in the main ISR of the project.

void **STPOSPLAN_runTick**(ST_POSPLAN_Handle handle)

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The pointer to a ST_PosPlan_t object

Copyright © 2013–2019, Texas Instruments Incorporated

The ST_PosPlan state transition map is the same as ST_VelPlan, as shown in Figure 3-10.

The states of ST_PosPlan are described in Table 3-21.

Table 3-21. SpinTAC Position Plan State Transition

From State	To State	Transition Condition[(1)(2)(3)(4)]	Action
IDLE			Keep IDLE status 1. Set ENB = false; 2. Keep PosStepInt = 0, PosStepFrac=0, and hold the values for VelLim, AccLim, and JrkLim
	INIT	RES == false AND ENB is on rising edge	
INIT	BUSY		Parameter validation 1. Reset internal status; 2. Enter state 0 and execute the actions defined for entering state 0.
BUSY			Operate the plan
	IDLE	RES == false AND ENB == True AND DON == true	Set ENB = false
	HALT	RES == true OR ENB == false	Load the halt state profile configurations
HALT	IDLE	HALT state Timer times up AND RES == true	Load the configurations of state 0
	WAIT	HALT state Timer times up AND RES == false	
WAIT	IDLE	RES == true	Load the configurations of state 0
	HALT	ENB == true	Load the configurations of the last state

(1) The RES signal provides the ability to place SpinTAC Position Plan into reset.

(2) The ENB signal controls the operation of SpinTAC Position Plan when RES is false.

(3) If ENB is set to false when SpinTAC Position Plan is running, SpinTAC Position Plan will then send out the position step and limits of the HALT state. When the unit profile is done, SpinTAC Position Plan will enter WAIT state, and can only continue the plan when ENB is set to true.
If RES is set to true, ENB will be set to false. SpinTAC Position Plan will then send out the position step and limits of state 0, and SpinTAC Position Plan enters IDLE state.

(4) Effectively, ENB functions as a pause/start button, while RES functions as stop.

Table 3-22 lists the functions that can be used to do operations like set, get, add, and delete configuration and runtime parameters of SpinTAC Position Plan. These functions are described in more detail in Section 3.5.9.

Table 3-22. SpinTAC Position Plan Additional Functions

Function Group	Function Name	Description
Initialization	STPOSPLAN_init	Initialize SpinTAC Position Plan
Configuration	STPOSPLAN_setCfgArray	Set the array that SpinTAC Position Plan will use to store the configuration
	STPOSPLAN_setCfg	Set the system parameters, protection parameters
	STPOSPLAN_setCfgHaltState	Set the parameters for the HALT state
	STPOSPLAN_addCfgState	Add a new State
	STPOSPLAN_addCfgVar	Add a new Variable
	STPOSPLAN_addCfgCond	Add a new Condition that compares a Variable against static values
	STPOSPLAN_addCfgVarCond	Add a new Condition that compares two Variables
	STPOSPLAN_addCfgTran	Add a new Transition
	STPOSPLAN_addCfgAct	Add a new Action

Copyright © 2013–2019, Texas Instruments Incorporated

Table 3-22. SpinTAC Position Plan Additional Functions (continued)

Function Group	Function Name	Description
Runtime	STPOSPLAN_run	Run SpinTAC Position Plan. Can run from main loop.
	STPOSPLAN_runTick	Run SpinTAC Position Plan Timer. Must run from main ISR
	STPOSPLAN_setVar	Set the value of a Variable during runtime
	STPOSPLAN_getVar	Get the value of a Variable during runtime
	STPOSPLAN_reset	Reset SpinTAC Position Plan and configuration
	STPOSPLAN_setUnitProfDone	Sets if the currently running profile is done
Plan modification and debugging functions (Provide runtime modification ability of Plan)	STPOSPLAN_getCfgStateNum	Get the number of configured States
	STPOSPLAN_getCfgVarNum	Get the number of configured Variables
	STPOSPLAN_getCfgCondNum	Get the number of configured Conditions
	STPOSPLAN_getCfgTranNum	Get the number of configured Transitions
	STPOSPLAN_getCfgActNum	Get the number of configured Actions
	STPOSPLAN_getCfg	Get the system & protection parameters
	STPOSPLAN_getCfgHaltState	Get the parameters for the HALT state
	Each function with a suffix -add has three other functions: -del, -set, and -get to delete the item, to set the item, and to get the item respectively. These functions allow online SpinTAC Position Plan modification.	

3.5.9 SpinTAC Functions

The following is a list of commonly used SpinTAC functions. This section is provided as a reference and prior to the implementation of a component the corresponding section of this document should be read.

void **STVELCTL_run**(ST_VELCTL_Handle)

Function: This function runs SpinTAC Velocity Control.

Parameters:

No.	Type	Parameters	Description
1	ST_VELCTL_Handle	handle	The handle to the ST_VelCtl_t datatype

void **STVELMOVE_run**(ST_VELMOVE_Handle)

Function: This function runs SpinTAC Velocity Move.

Parameters:

No.	Type	Parameters	Description
1	ST_VELMOVE_Handle	handle	The handle to the ST_VelMove_t datatype

void **STVELID_run**(ST_VELID_Handle)

Function: This function runs SpinTAC Velocity Identify.

Parameters:

No.	Type	Parameters	Description
1	ST_VELID_Handle	handle	The handle to the ST_VelId_t datatype

void **STVELPLAN_run**(ST_VELPLAN_Handle)

Function: This function runs SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype

Copyright © 2013–2019, Texas Instruments Incorporated

void **STVELPLAN_runTick**(ST_VELPLAN_Handle)

Function: This function runs SpinTAC Velocity Plan Timer. Must run from an ISR.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype

void **STVELPLAN_addCfgAct**(ST_VELPLAN_Handle, uint16_t, ST_PlanCond_e , uint16_t , uint16_t, uint16_t, ST_PlanActOptn_e, _iq24, ST_PlanActTrgr_e)

Function: This function adds an action to SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	uint16_t	StateIdx	The index of the state in which the action to be executed
3	ST_PlanCond_e	AndOr	ST_COND_NC: act without condition; ST_COND_FC: act when the first condition satisfied; ST_COND_AND: act when both conditions satisfied; ST_COND_OR: act when either condition satisfied;
4	uint16_t	CondIdx1	The index of the first condition
5	uint16_t	CondIdx2	The index of the second condition
6	uint16_t	VarIdx	The index of the variable to be operated
7	ST_PlanActOptn_e	Opt	ST_ACT_EQ: set the value to the variable ST_ACT_QDD: add the value to the variable
8	_iq24	Value	The value to be set to the variable or added to the variable
9	ST_PlanActTrgr_e	EnterExit	ST_ACT_ENTR: execute the action when entering the state ST_ACT_EXIT: execute the action when exiting the state

void **STVELPLAN_setCfgArray**(ST_VELPLAN_Handle, uint32_t *, const size_t, uint16_t, uint16_t, uint16_t, uint16_t, uint16_t)

Function: This function adds an action to SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	uint32_t *	cfgArray	Pointer to cfgArray used for Plan configuration.
3	const size_t	numBytes	The number of bytes allocated to the cfgArray array. Get number of bytes by calling sizeof(cfgArray)
4	uint16_t	MaxActNum	Number of Actions
5	uint16_t	MaxCondNum	Number of Conditions
6	uint16_t	MaxVarNum	Number if Variables
7	uint16_t	MaxTranNum	Number of Transitions
8	uint16_t	MaxStateNum	Number of States

Copyright © 2013–2019, Texas Instruments Incorporated

ST_VELPLAN_Handle **STVELPLAN_init** (void *, const size_t)

Function: This function initializes SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	void *	pMemory	The pointer to the memory for a ST_VelPlan_t object
2	const size_t	numBytes	The number of bytes allocated to the ST_VelPlan_t object

Return:

The ST_VelPlan_t object handle

void **STVELPLAN_addCfgCond**(ST_VELPLAN_Handle, uint16_t, ST_PlanComp_e, _iq24, _iq24)

Function: This function adds a condition to SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	uint16_t	VarIdx	The index of the variable to be compared
3	ST_PlanComp_e	Comp	ST _COMP_NA: Not defined; ST _COMP_EQ: equal to Value1; ST _COMP_NEQ: not equal to Value1; ST __COMP_GT: greater than Value1; ST_COMP_EGT: equal to or greater than Value1; ST_COMP_LW: lower than Value1; ST_COMP_ELW: equal to or lower than; ST_COMP_In: belong to the range of (Value1, Value2); ST_COMP_Ein: belong to the range of [Value1, Value2); ST_COMP_InE: belong to the range of (Value1, Value2]; ST_COMP_EinE: belong to the range of [Value1, Value2]; ST_COMP_Out: not belong to the range of [Value1, Value2]; ST_COMP_Eout: not belong to the range of (Value1, Value2]; ST_COMP_OutE: not belong to the range of [Value1, Value2); ST_COMP_EoutE: not belong to the range of (Value1, Value2)
4	_iq24	Value1	The first value
5	_iq24	Value2	The second value

void **STVELPLAN_addCfgVarCond** ST_VELPLAN_Handle, uint16_t, uint16_t, ST_PlanComp_e)

Function: This function adds a condition that comapres two variables to SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	uint16_t	VarIdx1	The index of the first variable to be compared
3	uint16_t	VarIdx2	The index of the second variable to be compared

Copyright © 2013–2019, Texas Instruments Incorporated

No.	Type	Parameters	Description
4	ST_PlanComp_e	Comp	ST _COMP_NA: Not defined; ST _COMP_EQ: equal to Value1; ST _COMP_NEQ: not equal to Value1; ST__COMP_GT: greater than Value1; ST_COMP_EGT: equal to or greater than Value1; ST_COMP_LW: lower than Value1; ST_COMP_ELW: equal to or lower than;

void **STVELPLAN_setCfgHaltState**(ST_VELPLAN_Handle, _iq24, _iq24, _iq20, int32_t)

Function: This function adds a state to SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	_iq24	VelEnd	Velocity set point for the HALT state
3	_iq24	AccLim	Acceleration limit for the HALT state
4	_iq20	JrkLim	Jerk limit for the HALT state
5	int32_t	Timer	Timer in [Count] to indicate how long to stay in the HALT state before further operation

void **STVELPLAN_setCfg**(ST_VELPLAN_Handle, _iq24, _iq24, _iq24, _iq20, bool)

Function: This function sets the SpinTAC Velocity Plan system configuration.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	_iq24	T_sec	Sample time in [s]
3	_iq24	VelMax	System maximum velocity limit
4	_iq24	AccMax	System maximum acceleration limit
5	_iq20	JrkMax	System maximum jerk limit
6	bool	LoopENB	false: plan operates only once and then goes back to IDLE state true: plan operate again each time entering IDLES state

void **STVELPLAN_addCfgState**(ST_VELPLAN_Handle, _iq24, int32_t)

Function: This function adds a state to SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	_iq24	VelEnd	Velocity set point for the state
3	int32_t	Timer_tick	Timer in [Count] to indicate how long to state in the state

void **STVELPLAN_addCfgTran**(ST_VELPLAN_Handle, uint16_t, uint16_t, ST_PlanCond_e, uint16_t, uint16_t, _iq24, _iq20)

Function: This function adds a transaction to SpinTAC Velocity Plan.

Parameters:

Copyright © 2013–2019, Texas Instruments Incorporated

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	uint16_t	FromState	The index of the state in which the transition starts
3	uint16_t	ToState	The index of the state in which the transition to ends
4	ST_PlanCond_e	AndOr	ST_COND_NC: transit without condition; ST_COND_FC: transit with the first condition satisfied; ST_COND_AND: transit with both conditions satisfied; ST_COND_OR: transit with either condition satisfied;
5	uint16_t	CondIdx1	The index of the first condition
6	uint16_t	CondIdx2	The index of the second condition
7	_iq24	AccLim	The acceleration limit used for this transition
8	_iq20	JrkLim	The jerk limit used for this transition

void **STVELPLAN_addCfgVar**(ST_VELPLAN_Handle, ST_PlanVar_e, _iq24)

Function: This function adds a variable to SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	ST_PlanVar_e	Type	ST_VAR_INOUT: timer type, can be used in condition ST_VAR_IN: sensor type, only receive value, can be used in condition ST_VAR_OUT: actuator type, only send value, cannot be used in condition
3	_iq24	Value	Initial value of the variable

void **STVELPLAN_setUnitProfDone**(ST_VELPLAN_Handle, bool)

Function: This function informs SpinTAC Velocity Plan if the currently running profile is done.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	bool	ProDON	false: not done; true: done

void **STVELPLAN_getVar**(ST_VELPLAN_Handle, uint16_t, _iq24 *)

Function: This function returns the value of a variable from SpinTAC Velocity Plan. It is typically used to send data from SpinTAC Velocity Plan to an external component

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	uint16_t	VarIdx	The index of the variable to receive the value
3	_iq24 *	Value	The pointer to the external variable to receive the value

Copyright © 2013–2019, Texas Instruments Incorporated

void **STVELPLAN_setVar**(ST_VELPLAN_Handle, uint16_t, _iq24)

Function: This function sets the value of a variable in SpinTAC Velocity Plan. It is typically used to pass sensor data into SpinTAC Velocity Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	uint16_t	VarIdx	The index of the variable to receive the value
3	_iq24	Value	The value to be set to the variable

void **STPOSCTL_run**(ST_POSCTL_Handle)

Function: This function runs SpinTAC Position Control.

Parameters:

No.	Type	Parameters	Description
1	ST_POSCTL_Handle	handle	The handle to the ST_PosCtl_t datatype

void **STPOSMOVE_run**(ST_POSMOVE_Handle)

Function: This function runs SpinTAC Position Move.

Parameters:

No.	Type	Parameters	Description
1	ST_POSMOVE_Handle	handle	The handle to the ST_PosMove_t datatype

void **STPOSCONV_run**(ST_POSCONV_Handle)

Function: This function runs SpinTAC Position Convert.

Parameters:

Parameters:

No.	Type	Parameters	Description
1	ST_POSCONV_Handle	handle	The handle to the ST_PosConv_t datatype

void **STPOSPLAN_run**(ST_POSPLAN_Handle)

Function: This function runs SpinTAC Position Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype

void **STPOSPLAN_runTick**(ST_POSPLAN_Handle)

Function: This function runs SpinTAC Position Plan Timer. Must run from an ISR.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype

Copyright © 2013–2019, Texas Instruments Incorporated

void **STPOSPLAN_addCfgAct**(ST_POSPLAN_Handle, uint16_t, ST_PlanCond_e , uint16_t , uint16_t, uint16_t, ST_PlanActOptn_e, _iq24, ST_PlanActTrgr_e)

Function: This function adds an action to SpinTAC Position Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	uint16_t	StateIdx	The index of the state in which the action to be executed
3	uint16_t	VarIdx	The index of the variable to be operated
4	ST_PlanCond_e	AndOr	ST_COND_NC: act without condition; ST_COND_FC: act when the first condition satisfied; ST_COND_AND: act when both conditions satisfied; ST_COND_OR: act when either condition satisfied;
5	uint16_t	CondIdx1	The index of the first condition
6	uint16_t	CondIdx2	The index of the second condition
7	ST_PlanActOptn_e	Opt	ST_ACT_EQ: set the value to the variable ST_ACT_QDD: add the value to the variable
8	_iq24	Value	The value to be set to the variable or added to the variable
9	ST_PlanActTrgr_e	EnterExit	ST_ACT_ENTR: execute the action when entering the state ST_ACT_EXIT: execute the action when exiting the state

void **STPOSPLAN_setCfgArray**(ST_POSPLAN_Handle, uint32_t *, const size_t, uint16_t, uint16_t, uint16_t, uint16_t, uint16_t)

Function: This function adds an action to SpinTAC Position Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	uint32_t *	cfgArray	Pointer to cfgArray used for Plan configuration.
3	const size_t	numBytes	The number of bytes allocated to the cfgArray array. Get number of bytes by calling sizeof(cfgArray)
4	uint16_t	MaxActNum	Number of Actions
5	uint16_t	MaxCondNum	Number of Conditions
6	uint16_t	MaxVarNum	Number if Variables
7	uint16_t	MaxTranNum	Number of Transitions
8	uint16_t	MaxStateNum	Number of States

ST_POSPLAN_Handle **STPOSPLAN_init**(void *, const size_t)

Function: This function initializes SpinTAC Position Plan.

Parameters:

No.	Type	Parameters	Description
1	void *	pMemory	The pointer to the memory for a ST_PosPlan_t object
	const size_t	numBytes	The number of bytes allocated to the ST_PosPlan_t object

Copyright © 2013–2019, Texas Instruments Incorporated

Return: The ST_PosPlan_t object handle

void **STPOSPLAN_addCfgCond**(ST_POSPLAN_Handle, uint16_t, ST_PlanComp_e, _iq24, _iq24)

Function: This function adds a condition to SpinTAC Position Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	uint16_t	VarIdx	The index of the variable to be compared
3	ST_PlanComp_e	Comp	ST _COMP_NA: Not defined; ST _COMP_EQ: equal to Value1; ST _COMP_NEQ: not equal to Value1; ST__COMP_GT: greater than Value1; ST_ COMP_EGT: equal to or greater than Value1; ST_ COMP_LW: lower than Value1; ST_ COMP_ELW: equal to or lower than; ST_ COMP_In: belong to the range of (Value1, Value2); ST_ COMP_Ein: belong to the range of [Value1, Value2); ST_ COMP_InE: belong to the range of (Value1, Value2]; ST_ COMP_EinE: belong to the range of [Value1, Value2]; ST_ COMP_Out: not belong to the range of [Value1, Value2]; ST_ COMP_Eout: not belong to the range of (Value1, Value2]; ST_ COMP_OutE: not belong to the range of [Value1, Value2); ST_ COMP_EoutE: not belong to the range of (Value1, Value2)
4	_iq24	Value1	The first value
5	_iq24	Value2	The second value

void **STPOSPLAN_addCfgVarCond**(ST_POSPLAN_Handle, uint16_t, uint16_t, ST_PlanComp_e)

Function: This function adds a condition that comapres two variables to SpinTAC Position Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	uint16_t	VarIdx1	The index of the first variable to be compared
3	uint16_t	VarIdx2	The index of the second variable to be compared
4	ST_PlanComp_e	Comp	ST _COMP_NA: Not defined; ST _COMP_EQ: equal to Value1; ST _COMP_NEQ: not equal to Value1; ST__COMP_GT: greater than Value1; ST_ COMP_EGT: equal to or greater than Value1; ST_ COMP_LW: lower than Value1; ST_ COMP_ELW: equal to or lower than;

void **STPOSPLAN_setCfgHaltState**(ST_POSPLAN_Handle, int32_t, _iq24,_iq24, _iq24, , int32_t)

Function: This function adds a state to SpinTAC Position Plan.

Parameters:

Copyright © 2013–2019, Texas Instruments Incorporated

No.	Type	Parameters	Description
1	ST_VELPLAN_Handle	handle	The handle to the ST_VelPlan_t datatype
2	int32_t	PosStepInt_mrev	Position step integer part for the HALT state
3	_iq24	PosStepFrac_mrev	Position step fraction part for the HALT state
4	_iq24	VelLim	Velocity limit for the HALT state
5	_iq24	AccLim	Acceleration limit for the HALT state
6	_iq20	JrkLim	Jerk limit for the HALT state
7	Int32_t	Timer	Timer in [Count] to indicate how long to stay in the HALT state before further operation

void **STPOSPLAN_setCfg**(ST_POSPLAN_Handle, _iq24, _iq24, _iq24, _iq20, bool)

Function: This function sets the SpinTAC Position Plan system configuration.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	_iq24	T_sec	Sample time in [s]
3	_iq24	VelMax	System maximum velocity limit
4	_iq24	AccMax	System maximum acceleration limit
5	_iq20	JrkMax	System maximum jerk limit
6	bool	LoopENB	false: plan operates only once and then goes back to IDLE state true: plan operate again each time entering IDLES state

void **STPOSPLAN_addCfgState**(ST_POSPLAN_Handle, int32_t, _iq24, int32_t)

Function: This function adds a state to SpinTAC Position Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	int32_t	PosStepInt_mrev	Position step command integer part value
3	_iq24	PosStepFrac_mrev	Position step command fraction part value
4	int32_t	Timer_tick	Timer in [Count] to indicate how long to state in the state

void **STPOSPLAN_addCfgTran**(ST_POSPLAN_Handle, uint16_t, uint16_t, ST_PlanCond_e, uint16_t, uint16_t, _iq24, _iq24, _iq20)

Function: This function adds a transaction to SpinTAC Position Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	uint16_t	FromState	The index of the state in which the transition starts
3	uint16_t	ToState	The index of the state in which the transition to ends
4	ST_PlanCond_e	AndOr	ST_COND_NC: transition without condition; ST_COND_FC: transition when the first condition satisfied; ST_COND_AND: transition when both conditions satisfied; ST_COND_OR: transition when either condition satisfied;

Copyright © 2013–2019, Texas Instruments Incorporated

No.	Type	Parameters	Description
5	uint16_t	CondIdx1	The index of the first condition
6	uint16_t	CondIdx2	The index of the second condition
7	_iq24	VelLim	The velocity limit used for this transition
8	_iq24	AccLim	The acceleration limit used for this transition
9	_iq24	DecLim	The deceleration limit used for this transition
10	_iq20	JrkLim	The jerk limit used for this transition

void **STPOSPLAN_addCfgVar**(ST_POSPLAN_Handle, ST_PlanVar_e, _iq24)

Function: This function adds a variable to SpinTAC Position Plan.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	ST_PlanVar_e	Type	ST_VAR_INOUT: timer type, can be used in condition ST_VAR_IN: sensor type, only receive value, can be used in condition ST_VAR_OUT: actuator type, only send value, cannot be used in condition
3	_iq24	Value	Initial value of the variable

void **STPOSPLAN_setUnitProfDone**(ST_POSPLAN_Handle, bool)

Function: This function informs SpinTAC Position Plan if the currently running profile is done.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	bool	ProDON	false: not done; true: done

void **STPOSPLAN_getVar**(ST_POSPLAN_Handle, uint16_t, _iq24 *)

Function: This function returns the value of a variable from SpinTAC Position Plan. It is typically used to send data from SpinTAC Position Plan to an external component.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	uint16_t	VarIdx	The index of the variable to receive the value
3	_iq24 *	Value	The pointer to the external variable to receive the value

void **STPOSPLAN_setVar**(ST_POSPLAN_Handle, uint16_t, _iq24)

Function: This function returns the value of a variable from SpinTAC Position Plan. It is typically used to send data from SpinTAC Position Plan to an external component.

Parameters:

No.	Type	Parameters	Description
1	ST_POSPLAN_Handle	handle	The handle to the ST_PosPlan_t datatype
2	uint16_t	VarIdx	The index of the variable to receive the value
3	_iq24	Value	The value to be set to the variable

Copyright © 2013–2019, Texas Instruments Incorporated

void **ST_getVersionNumber**(ST_VER_Handle, uint16_t *, uint16_t *, uint16_t *)

Function: This function returns the version number of the SpinTAC library.

Parameters:

No.	Type	Parameters	Description
1	ST_VER_Handle	handle	The hanlde to the ST_Ver_t datatype
2	uint16_t *	major	Major version of library
3	uint16_t *	minor	Minor version of library
4	uint16_t *	revision	Revision version of library

Copyright © 2013–2019, Texas Instruments Incorporated

User Parameters (user.h)

User.h is where all user parameters are stored. Some of these values can be manipulated through the GUI or CCStudio during run-time, but must be updated in user.h for permanent saving.

Topic **Page**

4.1 Currents and Voltages 205
4.2 Clocks and Timers 207
4.3 Decimation 208
4.4 Limits 209
4.5 Poles 211
4.6 User Motor and ID Settings 212
4.7 SpinTAC Parameters (spintac_velocity.h and spintac_position.h) 214
4.8 Setting ACIM Motor Parameters in user.h 218

Copyright © 2013–2019, Texas Instruments Incorporated

4.1 Currents and Voltages

User.h contains the public interface for user initialization data for the CTRL, HAL, and EST modules.

4.1.1 USER_IQ_FULL_SCALE_FREQ_Hz

```
#define USER_IQ_FULL_SCALE_FREQ_Hz   (800.0)
```

This module defines the full scale frequency for IQ variable, in Hz. All frequencies are converted into (pu) based on the ratio to this value. This value must be larger than the maximum speed that you are expecting from the motor.

4.1.2 USER_IQ_FULL_SCALE_VOLTAGE_V

```
#define USER_IQ_FULL_SCALE_VOLTAGE_V   (24.0)
```

This module defines the full-scale value for the IQ30 variable of voltage inside the system. All voltages are converted into pu based on the ratio to this value.

CAUTION

- This value MUST be larger than the maximum value of any voltage calculated inside the control system otherwise the value can saturate and roll over, causing an inaccurate value.
- This value is OFTEN greater than the maximum measured ADC value, especially with high Bemf motors operating at higher than rated speeds.
- If you know the value of your Bemf constant, and you know you are operating at higher than rated speed due to field weakening, be sure to set this value higher than the expected Bemf voltage
- This value can also be used to calculate the minimum flux that can be identified by calculating the following formula:

  USER_IQ_FULL_SCALE_VOLTAGE_V/USER_EST_FREQ_Hz/0.7

For high-flux motors (that is, washing machine motors), it is recommended to start with a value ~3x greater than the USER_ADC_FULL_SCALE_VOLTAGE_V and increase to 4-5x if scenarios where a Bemf calculation may exceed these limits.

For low-flux motors (that is, low-inductance high-speed hobby motors), it is recommended to have a value that allows flux identification as per the equation:

USER_IQ_FULL_SCALE_VOLTAGE_V/USER_EST_FREQ_Hz/0.7

4.1.3 USER_ADC_FULL_SCALE_VOLTAGE_V

```
#define USER_ADC_FULL_SCALE_VOLTAGE_V   (66.32)
```

This module defines the maximum voltage at the input to the AD converter. The value that will be represented by the maximum ADC input (3.3V) and conversion (0FFFh). Hardware dependent, this should be based on the voltage sensing and scaling to the ADC input.

4.1.4 USER_VOLTAGE_SF

```
#define USER_VOLTAGE_SF
((float_t)((USER_ADC_FULL_SCALE_VOLTAGE_V)/(USER_IQ_FULL_SCALE_VOLTAGE_V)))
```

This module defines the voltage scale factor for the system.

The compile time calculation for scale factor (ratio) is used throughout the system.

4.1.5 USER_IQ_FULL_SCALE_CURRENT_A

```
#define USER_IQ_FULL_SCALE_CURRENT_A   (10.0)
```

This module defines the full scale current for the IQ variables, in A.

Copyright © 2013–2019, Texas Instruments Incorporated

All currents are converted into (pu) based on the ratio to this value.

CAUTION

This value MUST be larger than the maximum current readings that you are expecting from the motor or the reading will roll over to 0, creating a control issue.

4.1.6 USER_ADC_FULL_SCALE_CURRENT_A

```
#define USER_ADC_FULL_SCALE_CURRENT_A   (17.30)
```

This module defines the maximum current at the AD converter.

This value will be represented by the maximum ADC input (3.3 V) and conversion (0FFFh).

The value is hardware dependent and should be based on the current sensing and scaling to the ADC input.

4.1.7 USER_CURRENT_SF

```
#define USER_CURRENT_SF
((float_t)((USER_ADC_FULL_SCALE_CURRENT_A)/(USER_IQ_FULL_SCALE_CURRENT_A)))
```

This module is the scale factor for the system.

The compile time calculation for scale factor (ratio) is used throughout the system.

4.1.8 USER_NUM_CURRENT_SENSORS

```
#define USER_NUM_CURRENT_SENSORS   (3)
```

This module defines the number of current sensors used.

The value is defined by the hardware capability present; may be (2) or (3).

4.1.9 USER_NUM_VOLTAGE_SENSORS

```
#define USER_NUM_VOLTAGE_SENSORS   (3)
```

This module defines the number of voltage (phase) sensors.

The value must be (3).

4.1.10 I_A_offset , I_B_offset , I_C_offset

```
#define I_A_offset (0. 8661925197)
#define I_B_offset (0.8679816127)
#define I_C_offset (0.8638074994)
```

This module defines the ADC current offsets for A, B, and C phases.

One-time hardware dependent, though the calibration can be done at run-time as well.

After initial board calibration these values should be updated for your specific hardware so they are available after compile in the binary to be loaded to the controller.

4.1.11 V_A_offset , V_B_offset , V_C_offset

```
#define V_A_offset   (0.1776982546)
#define V_B_offset   (0.1776063442)
#define V_C_offset   (0.1771019101)
```

This module defines the ADC voltage offsets for A, B, and C phases.

One-time hardware dependent, though the calibration can be done at run-time as well.

After initial board calibration these values should be updated for your specific hardware so they are available after compile in the binary to be loaded to the controller.

Copyright © 2013–2019, Texas Instruments Incorporated

4.2 Clocks and Timers

4.2.1 USER_SYSTEM_FREQ_MHz

```
#define USER_SYSTEM_FREQ_MHz   (90.0) // Maximum frequency for F2805xF/M and F2806xF/M devices
#define USER_SYSTEM_FREQ_MHz   (60.0) // Maximum frequency for F2802xF devices
```

This module defines the system clock frequency, in MHz.

4.2.2 USER_PWM_FREQ_kHz

```
#define USER_PWM_FREQ_kHz   (20.0)
```

This module defines the Pulse Width Modulation (PWM) frequency, in kHz.

PWM frequency can be set directly here up to 30 kHz safely (60 kHz MAX in some cases).

For higher PWM frequencies (60 kHz+ typical for low-inductance, high-current ripple motors), it is recommended to use the ePWM hardware and adjustable ADC SOC to decimate the ADC conversion done interrupt to the control system. This can be done by using the hardware decimation USER_NUM_PWM_TICKS_PER_ISR_TICK. If hardware decimation is not used in high PWM frequencies, there is a risk of missing interrupts and disrupting the timing of the control state machine.

4.2.3 USER_MAX_VS_MAG_PU

```
#define USER_MAX_VS_MAG_PU   (0.5)
```

Set to 1.0 if a current reconstruction technique is not used. For more information, see the svgen_current module in Lab10a-x.

Defines the maximum voltage vector (Vs) magnitude allowed. This value sets the maximum magnitude for the output of the Id and Iq PI current controllers. The Id and Iq current controller outputs are Vd and Vq.

The relationship between Vs, Vd, and Vq is:

Vs = sqrt(Vd^2 + Vq^2).

In this FOC controller, the Vd value is set equal to USER_MAX_VS_MAG*USER_VD_MAG_FACTOR.

Vq = sqrt(USER_MAX_VS_MAG^2 - Vd^2).

- Set USER_MAX_VS_MAG = 0.5 for a pure sinewave with a peak at SQRT(3)/2 = 86.6% duty cycle. No current reconstruction is needed for this scenario.
- Set USER_MAX_VS_MAG = 2/SQRT(3) = 0.5774 for a pure sinewave with a peak at 100% duty cycle. Current reconstruction will be needed for this scenario (Lab10a-x).
- Set USER_MAX_VS_MAG = 4/3 = 0.6666 to create a trapezoidal voltage waveform. Current reconstruction will be needed for this scenario (Lab10a-x).
- For space vector over-modulation, see lab 10 for details on system requirements that will allow the SVM generator to go all the way to trapezoidal.

4.2.4 USER_PWM_PERIOD_usec

```
#define USER_PWM_PERIOD_usec   (1000.0/USER_PWM_FREQ_kHz)
```

This module defines the Pulse Width Modulation (PWM) period, in µsec.

Compile time calculation.

4.2.5 USER_ISR_FREQ_Hz

```
#define USER_ISR_FREQ_Hz
((float_t)USER_PWM_FREQ_kHz * 1000.0 / (float_t)USER_NUM_PWM_TICKS_PER_ISR_TICK)
```

This module defines the Interrupt Service Routine (ISR) frequency, in Hz.

Compile time calculation.

Copyright © 2013–2019, Texas Instruments Incorporated

4.2.6 USER_ISR_PERIOD_usec

```
#define USER_ISR_PERIOD_usec
  (USER_PWM_PERIOD_usec * (float_t)USER_NUM_PWM_TICKS_PER_ISR_TICK)
```

This module defines the Interrupt Service Routine (ISR) period, in µsec.

4.3 Decimation

Decimation defines the number of ticks between module execution.

Controller clock tick (CTRL) is the main clock used for all timing in the software.

Typically the PWM Frequency triggers (can be decimated by the ePWM hardware for less overhead) an ADC SOC.

ADC SOC triggers an ADC Conversion Done.

ADC Conversion Done triggers ISR.

This relates the hardware ISR rate to the software controller rate.

Typcially want to consider some form of decimation (ePWM hardware, CURRENT or EST) over 16kHz ISR to insure interrupt completes and leaves time for background tasks.

4.3.1 USER_NUM_PWM_TICKS_PER_ISR_TICK

```
#define USER_NUM_PWM_TICKS_PER_ISR_TICK    (1)
```

This module defines the number of PWM periods per interrupt.

Relationship between PWM frequency and interrupt frequency.

4.3.2 USER_NUM_ISR_TICKS_PER_CTRL_TICK

```
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK   (1)
```

This module defines the number of controller clock ticks per current controller clock tick.

Relationship of controller clock rate to current controller (FOC) rate.

4.3.3 USER_NUM_CTRL_TICKS_PER_CURRENT_TICK

```
#define USER_NUM_CTRL_TICKS_PER_CURRENT_TICK   (1)
```

This module defines the number of controller clock ticks per estimator clock tick.

Relationship of controller clock rate to estimator (FAST) rate.

4.3.4 USER_NUM_CTRL_TICKS_PER_EST_TICK

```
#define USER_NUM_CTRL_TICKS_PER_EST_TICK   (1)
```

This module depends on needed dynamic performance, FAST provides very good results as low as 1 kHz while more dynamic or high speed applications may require up to 15 kHz.

4.3.5 USER_NUM_CTRL_TICKS_PER_SPEED_TICK

```
#define USER_NUM_CTRL_TICKS_PER_SPEED_TICK   (15)
```

This module defines the number of controller clock ticks per speed controller clock tick.

The value is the relationship of controller clock rate to speed loop rate.

4.3.6 USER_NUM_CTRL_TICKS_PER_TRAJ_TICK

```
#define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK   (15)
```

This module defines the number of controller clock ticks per trajectory clock tick.

The value is the relationship of controller clock rate to trajectory loop rate; typically, the same as the speed rate.

Copyright © 2013–2019, Texas Instruments Incorporated

4.3.7 USER_CTRL_FREQ_Hz

#define USER_CTRL_FREQ_Hz
(uint_least32_t)(USER_ISR_FREQ_Hz/USER_NUM_ISR_TICKS_PER_CTRL_TICK)

This module defines the controller frequency, in Hz.

Compile time calculation.

4.3.8 USER_EST_FREQ_Hz

```
#define USER_EST_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_EST_TICK)
```

This module defines the estimator frequency, in Hz.

Compile time calculation.

4.3.9 USER_TRAJ_FREQ_Hz

```
#define USER_TRAJ_FREQ_Hz
(uint_least32_t)(USER_CTRL_FREQ_Hz/USER_NUM_CTRL_TICKS_PER_TRAJ_TICK)
```

This module defines the trajectory frequency, in Hz.

Compile time calculation.

4.3.10 USER_CTRL_PERIOD_usec

```
#define USER_CTRL_PERIOD_usec   (USER_ISR_PERIOD_usec *
USER_NUM_ISR_TICKS_PER_CTRL_TICK)
```

This module defines the controller execution period, in µsec.

Compile time calculation.

4.3.11 USER_CTRL_PERIOD_sec

```
#define USER_CTRL_PERIOD_sec
((float_t)USER_CTRL_PERIOD_usec/(float_t)1000000.0)
```

This module defines the controller execution period, in sec.

Compile time calculation.

4.4 Limits

4.4.1 USER_MAX_NEGATIVE_ID_REF_CURRENT_A

```
#define USER_MAX_NEGATIVE_ID_REF_CURRENT_A   (-0.5 * USER_MOTOR_MAX_CURRENT)
```

Example, adjust to meet safety needs of your motor:

-0.5 * USER_MOTOR_MAX_CURRENT

This module defines the maximum negative current to be applied in Id reference.

Used in field weakening only, this is a safety setting (for example, to protect against demagnetization).

User must also be aware that overall current magnitude [$\sqrt{(Id^2 + Iq^2)}$] should be kept below any machine design specifications.

4.4.2 USER_ZEROSPEEDLIMIT

```
#define USER_ZEROSPEEDLIMIT   (1.0 / USER_IQ_FULL_SCALE_FREQ_Hz)
```

typical = 0.002 pu, 1-5 Hz

Hz = USER_ZEROSPEEDLIMIT * USER_IQ_FULL_SCALE_FREQ_Hz

This module defines the low speed limit for the flux integrator, in pu.

Copyright © 2013–2019, Texas Instruments Incorporated

This is the speed range (CW/CCW) at which the ForceAngle object is active, but only if enabled.

Outside of this speed, or if disabled, the ForceAngle will NEVER be active and the angle is provided by FAST only.

4.4.3 USER_FORCE_ANGLE_FREQ_Hz

```
#define USER_FORCE_ANGLE_FREQ_Hz   (USER_ZEROSPEEDLIMIT * USER_IQ_FULL_SCALE_FREQ_Hz)
```

Typical force angle start-up speed = 1.0

This module defines the force angle frequency, in Hz.

Frequency of stator vector rotation used by the ForceAngle object.

Can be positive or negative

4.4.4 USER_MAX_CURRENT_SLOPE_POWERWARP

```
#define USER_MAX_CURRENT_SLOPE_POWERWARP
 (0.3*USER_MOTOR_RES_EST_CURRENT/USER_IQ_FULL_SCALE_CURRENT_A/USER_TRAJ_FREQ_Hz)
```

This defines the maximum current slope for Id trajectory during PowerWarp mode.

For Induction motors only, controls how fast Id input can change under PowerWarp control.

4.4.5 USER_MAX_ACCEL_Hzps

```
#define USER_MAX_ACCEL_Hzps   (20.0)
```

This module defines the starting maximum acceleration ***and*** deceleration for the speed profiles, in Hz/sec.

Updated in run-time through user functions.

Inverter, motor, inertia, and load will limit actual acceleration capability.

4.4.6 USER_MAX_ACCEL_EST_Hzps

```
#define USER_MAX_ACCEL_EST_Hzps   (2.0)
```

This module defines maximum acceleration for the estimation speed profiles, in Hz/sec.

Only used during Motor ID (commission).

4.4.7 USER_MAX_CURRENT_SLOPE

```
#define USER_MAX_CURRENT_SLOPE
(USER_MOTOR_RES_EST_CURRENT/USER_IQ_FULL_SCALE_CURRENT_A/USER_TRAJ_FREQ_Hz)
```

This module defines the maximum current slope for Id trajectory during estimation.

4.4.8 USER_IDRATED_FRACTION_FOR_RATED_FLUX

```
#define USER_IDRATED_FRACTION_FOR_RATED_FLUX   (1.0)
```

This module defines the fraction of IdRated to use during rated flux estimation.

Default is 1.0; do not change.

4.4.9 USER_IDRATED_FRACTION_FOR_L_IDENT

```
#define USER_IDRATED_FRACTION_FOR_L_IDENT   (1.0)
```

This module defines the fraction of IdRated to use during inductance estimation.

Default is 1.0; do not change.

4.4.10 USER_IDRATED_DELTA

```
#define USER_IDRATED_DELTA   (0.00002)
```

This module defines the IdRated delta to use during estimation.

Copyright © 2013–2019, Texas Instruments Incorporated

4.4.11 USER_SPEEDMAX_FRACTION_FOR_L_IDENT

```
#define USER_SPEEDMAX_FRACTION_FOR_L_IDENT   (1.0)
```

This module defines the fraction of SpeedMax to use during inductance estimation.

4.4.12 USER_FLUX_FRACTION

```
#define USER_FLUX_FRACTION   (1.0)
```

This module defines flux fraction to use during inductance identification.

4.4.13 USER_POWERWARP_GAIN

```
#define USER_POWERWARP_GAIN   (1.0)
```

This module defines the PowerWarp gain for computing Id reference.

Induction motors only. Default is 1.0; do not change.

4.4.14 USER_R_OVER_L_EST_FREQ_Hz

```
#define USER_R_OVER_L_EST_FREQ_Hz   (300)
```

This module defines the R/L estimation frequency, in Hz.

4.5 Poles

4.5.1 USER_VOLTAGE_FILTER_POLE_Hz

```
#define USER_VOLTAGE_FILTER_POLE_Hz   (344.62)
```

This module defines the analog filter pole location, in Hz.

The value must match the hardware voltage feedback filter that is calculated based on Equation 8 in Section 5.2.4.

4.5.2 USER_VOLTAGE_FILTER_POLE_rps

#define USER_VOLTAGE_FILTER_POLE_rps (2.0 * MATH_PI * USER_VOLTAGE_FILTER_POLE_Hz)

This module defines the analog voltage filter pole location, in rad/s.

Compile time calcuation from Hz to rad/s.

4.5.3 USER_OFFSET_POLE_rps

```
#define USER_OFFSET_POLE_rps   (20.0)
```

This module defines the software pole location for the voltage and current offset estimation, in rad/s.

Should not be changed from default of (20.0).

4.5.4 USER_FLUX_POLE_rps

```
#define USER_FLUX_POLE_rps   (100.0)
```

This module defines the software pole location for the flux estimation, in rad/s.

This value should not be changed from default of 100.0 rps.

4.5.5 USER_DIRECTION_POLE_rps

```
#define USER_DIRECTION_POLE_rps   (6.0)
```

This module defines the software pole location for the direction filter, in rad/s.

Copyright © 2013–2019, Texas Instruments Incorporated

4.5.6 USER_SPEED_POLE_rps

```
#define USER_SPEED_POLE_rps   (100.0)
```

This module defines the software pole location for the frequency estimator filter, in rad/s. For most applications, 100.0 rps is sufficient. For high-speed motors, performance may be improved by increasing this value up to 500.0

4.5.7 USER_DCBUS_POLE_rps

```
#define USER_DCBUS_POLE_rps   (100.0)
```

This module defines the software pole location for the DC bus filter, in rad/s.

4.5.8 USER_EST_KAPPAQ

```
#defineUSER_EST_KAPPAQ   (1.5)
```

This module defines the convergence factor for the estimator.

Do not change from default for FAST.

4.6 User Motor and ID Settings

4.6.1 USER_MOTOR_TYPE

```
#define USER_MOTOR_TYPE   MOTOR_Type_Pm
```

Motor_Type_Pm (All Synchronous: BLDC, PMSM, SMPM, IPM) or Motor_Type_Induction (Asynchronous ACI).

4.6.2 USER_MOTOR_NUM_POLE_PAIRS

```
#define USER_MOTOR_NUM_POLE_PAIRS   (4)
```

PAIRS, not total poles. Used to calculate user RPM from rotor, in Hz only.

4.6.3 USER_MOTOR_Rr

```
#define USER_MOTOR_Rr   (NULL)
```

Induction motors only, else NULL.

4.6.4 USER_MOTOR_Rs

```
#define USER_MOTOR_Rs   (2.303403)
```

Identified phase to neutral resistance in a Y equivalent circuit (Ohms, float).

4.6.5 USER_MOTOR_Ls_d

```
#define USER_MOTOR_Ls_d   (0.008464367)
```

For PM, Identified average stator inductance (Henry, float).

4.6.6 USER_MOTOR_Ls_q

```
#define USER_MOTOR_Ls_q   (0.008464367)
```

For PM, Identified average stator inductance (Henry, float).

4.6.7 USER_MOTOR_RATED_FLUX

```
#define USER_MOTOR_RATED_FLUX   (0.38)
```

Identified TOTAL flux linkage between the rotor and the stator (Webers = Volts* Seconds).

Copyright © 2013–2019, Texas Instruments Incorporated

4.6.8 USER_MOTOR_MAGNETIZING_CURRENT

```
#define USER_MOTOR_MAGNETIZING_CURRENT   (NULL)
```

Induction motors only, else NULL.

4.6.9 USER_MOTOR_RES_EST_CURRENT

```
#define USER_MOTOR_RES_EST_CURRENT   (1.0)
```

During Motor ID, maximum current (Amperes, float) used for Rs estimation, 10-20%.

4.6.10 USER_MOTOR_IND_EST_CURRENT

```
#define USER_MOTOR_IND_EST_CURRENT   (-1.0)
```

During Motor ID, maximum current (negative Amperes, float) used for Ls estimation, use just enough to enable rotation.

4.6.11 USER_MOTOR_MAX_CURRENT

```
#define USER_MOTOR_MAX_CURRENT   (3.82)
```

CRITICAL: Used during ID and run-time, sets a limit on the maximum current command output of the provided Speed PI Controller to the Iq controller.

4.6.12 USER_MOTOR_FLUX_EST_FREQ_Hz

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

During Motor ID, maximum commanded speed (Hz, float), ~10% rated.

4.6.13 USER_MOTOR_ENCODER_LINES (InstaSPIN-MOTION Only)

```
#define USER_MOTOR_ENCODER_LINES      (2500.0)
```

Used to setup an encoder, this provides the place to specify the number of lines on the encoder wheel.

4.6.14 USER_MOTOR_MAX_SPEED_KRPM (InstaSPIN-MOTION Only)

```
#define USER_MOTOR_MAX_SPEED_KRPM      (3.0)
```

Used to set an upper bound on speed reference for position control applications.

4.6.15 USER_SYSTEM_INERTIA (InstaSPIN-MOTION Only)

```
#define USER_SYSTEM_INERTIA      (0.02)
```

Inertia describes the amount of mass that is rigidly coupled with the motor. This is used by the InstaSPIN-MOTION controllers as an input. It should be identified with InstaSPIN-MOTION Inertia Identification.

4.6.16 USER_SYSTEM_FRICTION (InstaSPIN-MOTION Only)

```
#define USER_SYSTEM_FRICTION      (0.02)
```

Friction describes the resistance to motion that is seen by the motor. This is used by the InstaSPIN-MOTION controllers as an input. It should be identified with InstaSPIN-MOTION Inertia Identification.

4.6.17 USER_SYSTEM_BANDWIDTH_SCALE (InstaSPIN-MOTION Only)

```
#define USER_SYSTEM_BANDWIDTH_SCALE      (1.0)
```

Bandwidth Scale sets the default bandwidth that is used by the InstaSPIN-MOTION controllers. This should be updated after completing a tuning process with the InstaSPIN-MOTION controller.

Copyright © 2013–2019, Texas Instruments Incorporated

4.7 SpinTAC Parameters (spintac_velocity.h and spintac_position.h)

The SpinTAC components that make up InstaSPIN-MOTION need to be configured for your specific application. This is a simple process. The specific configuration details of the SpinTAC components in InstaSPIN-MOTION will be covered in subsequent sections of this document. The spintac_velocity.h and spintac_position.h header files provide a singular interface to include the SpinTAC components in your project. These files contain the macro definitions, type definitions, and function definitions required to setup the SpinTAC components. Including these files in your project is the first step to using the SpinTAC components that make up InstaSPIN-MOTION. The file that should be included is dependent on the type of control. If the application is a velocity application, spintac_velocity.h should be included. If the application requires position control, spintac_position.h should be included.

4.7.1 Macro Definitions

The macro definitions (#define) provide a simple method for updating your configuration in a large number of places throughout the Motorware project. The #define directive specifies a macro identifier and a replacement. The replacement is substituted for every subsequent occurrence of that macro identifier at compile time. The macro definitions used by SpinTAC components in spintac_velocity.h and spintac_position.h are described herein.

4.7.1.1 ST_MREV_ROLLOVER (spintac_position.h only)

This defines the maximum and minimum value that will be used to represent mechanical revolutions inside SpinTAC Position Control. This is used in order to maintain precision. When the value for mechanical revolutions reaches the value defined in ST_MREV_ROLLOVER, a rollover counter is incremented, and the value for mechanical revolutions will be set to negative ST_MREV_ROLLOVER.

4.7.1.2 ST_EREV_MAXIMUM (spintac_position.h only)

This defines the maximum value for an electrical revolution. This is the maximum electrical angle value produced by the encoder or other electrical angle source.

4.7.1.3 ST_POS_ERROR_MAXIMUM_MREV (spintac_position.h only)

This defines the maximum position error allowable in the application. If a position error beyond this threshold is detected it will force the controller output to zero until the error has been reduced below this threshold.

4.7.1.4 ST_ISR_TICKS_PER_SPINTAC_TICK

This identifies the decimation factor for the SpinTAC components of InstaSPIN-MOTION. This value represents the number of ISRs completed in-between each execution of the SpinTAC components. This value is calculated from parameters defined in user.h. Decimation factors are further explained in Section 9.2.

4.7.1.5 ST_SPEED_SAMPLE_TIME

This identifies how often the SpinTAC components are executed. This value is calculated from parameters that are defined in user.h.

4.7.1.6 ST_SPEED_PU_PER_KRPM

This identifies the scaling between kilo-rpm and pu/s speed. This is used to convert from user variables which are typically in kilo-rpm into scaled speed variables. This ensures that all calculations in InstaSPIN-MOTION will not cause an overflow. This value is calculated from parameters that are defined in user.h

4.7.1.7 ST_SPEED_KRPM_PER_PU

This identifies the scaling between pu/s and kilo-rpm speed. This is used to convert from scaled speed variables into user unit variables. This is done to ensure that all calculations in InstSPIN-MOTION will not cause an overflow. This value is calculated from parameters that are defined in user.h

Copyright © 2013–2019, Texas Instruments Incorporated

4.7.1.8 ST_MOTOR_INERTIA_PU

This identifies the inertia of the system in scaled units. This value is calculated from ST_MOTOR_INERTIA_A_PER_KRPM and will be provided to SpinTAC Velocity Control or SpinTAC Position Control.

4.7.1.9 ST_MOTOR_FRICTION_PU

This identifies the inertia of the system in scaled units. This value is calculated from ST_MOTOR_FRICTION_A_PER_KRPM and will be provided to SpinTAC Velocity Control or SpinTAC Position Control.

4.7.1.10 ST_MIN_ID_SPEED_RPM

This identifies the minimum speed of the motor prior to running SpinTAC Velocity Identify. This is done to ensure that the inertia identification process does not begin if the motor is spinning too quickly. This value is specified in rpm. If there is difficulty in beginning the inertia identification process, and the motor has trouble holding a zero speed, this value should be increased to widen the bound from which the motor will start the inertia identification process.

4.7.1.11 ST_MIN_ID_SPEED_PU

This identifies the minimum speed of the motor prior to running SpinTAC Velocity Identify. This value is calculated from ST_MIN_ID_SPEED_RPM. This is the value that will be compared against in the user project.

4.7.1.12 ST_ID_INCOMPLETE_ERROR

This error is triggered anytime that SpinTAC Velocity Identify fails and the system inertia remains unknown. The value refers to the specific error code produced by the SpinTAC Velocity Identify component. Do not modify this value.

4.7.1.13 ST_VARS_DEFAULTS

This identifies the default values that should be loaded into the ST_Vars_t structure described in Section 4.7.2.5. These default values are used to initialize the structure and to provide the correct values at startup. Do not modify these values.

4.7.2 Type Definitions

Type definitions (typedefs) organize the code that calls SpinTAC and simplify the API experience for the user.

4.7.2.1 VEL_Params_t / POS_Params_t

This structure identifies which SpinTAC component will be used in the project. This structure is included as part of ST_Obj discussed in Section 4.7.2.2. This structure should not be declared in your project.

4.7.2.2 ST_Obj

This is the main structure of the SpinTAC components. This structure is designed to align the SpinTAC components around a single motor axis. This structure contains substructures that separate the SpinTAC components into the portion of the motor axis that they work on. This structure should be declared in the main source file of your project.

4.7.2.3 ST_Handle

This handle is used to represent the address of the main SpinTAC structure. It should be used in the user project to pass the address of the main SpinTAC structure into the user functions so that they will not be operating on a global variable. The lab projects included in MotorWare have examples of how to setup user functions to use the handle for interfacing.

Copyright © 2013–2019, Texas Instruments Incorporated

4.7.2.4 ST_PlanButton_e

This enumeration identifies the states that can be set to control the operation of SpinTAC Plan. This is done to construct a small state machine to handle enabling, disabling, and resetting SpinTAC Plan. This should be used in the main source file to establish a button that can be used to control the operation of SpinTAC Plan.

4.7.2.5 ST_Vars_t

This is a structure that contains the user interface variables that should be used to operate on the SpinTAC components. This structure should not be declared in the main source file. It is already declared as part of MOTOR_Vars_t in main.h and main_position.h. More information about MOTOR_Vars_t can be found in the lab documentation.

4.7.3 Functions

These functions are used to initialize and configure the SpinTAC components.

4.7.3.1 ST_init

This function initializes the SpinTAC structures. It should be called prior to the forever loop in the main source file. This function will pass the memory locations of the SpinTAC components and will return a handle to be used to interface to these components.

4.7.3.2 ST_setupPosConv

This function sets up the default values for the SpinTAC Position Converter. It should be called after ST_init, but before the forever loop in the main source file. This function extracts configuration values from the macro definitions in user.h and spintac.h. These values should be modified to fit your system.

4.7.3.3 ST_setupVelCtl (Velocity Control Only)

This function sets up the default values for the SpinTAC Velocity Control. It should be called after ST_init, but before the forever loop in the main source file. This function extracts configuration values from the macro definitions in user.h and spintac_velocity.h. These values should be modified to fit your system.

4.7.3.4 ST_setupPosCtl (Position Control Only)

This function sets up the default values for SpinTAC Position Control. It should be called after ST_init, but before the forever loop in the main source file. This function extracts configuration values from the macro definitions in user.h and spintac_position.h. These values should be modified to fit your system.

4.7.3.5 ST_setupVelMove (Velocity Control Only)

This function sets up the default values for SpinTAC Velocity Move. It should be called after ST_init, but before the forever loop in the main source file. This function extracts configuration values from the macro definitions in user.h and spintac_velocity.h. These values should be modified to fit your system.

4.7.3.6 ST_setupPosMove (Position Control Only)

This function sets up the default values for SpinTAC Position Move. It should be called after ST_init, but before the forever loop in the main source file. This function extracts configuration values from the macro definitions in user.h and spintac_position.h. These values should be modified to fit your system.

4.7.3.7 ST_setupVelPlan (Velocity Control Only)

This function is used to setup SpinTAC Velocity Plan. This function should be called after the ST_init function, and should be declared and written in the main source file of the project. For more information about how to configure SpinTAC Plan, see Section 13.5.

Copyright © 2013–2019, Texas Instruments Incorporated

4.7.3.8 ST_setupPosPlan (Position Control Only)

This function is used to setup SpinTAC Position Plan. This function should be called after the ST_init function, and should be declared and written in the main source file of the project. For more information about how to configure SpinTAC Plan, see Section 13.5.

4.7.3.9 ST_setupVelId (Velocity Control Only)

This function sets up the default values for the SpinTAC Velocity Identify. It should be called after ST_init, but before the forever loop in the main source file. This function extracts configuration values from the macro definitions in user.h and spintac_velocity.h. These values should be modified to fit your system.

4.7.3.10 ST_runPosConv

This function is used to run the SpinTAC Position Converter in the ISR. This function should be decimated at the rate defined in ISR_TICKS_PER_SPINTAC_TICK. This function should be declared and written in the main source file of the project. The InstaSPIN-MOTION lab projects 12 through 13e provide an example of how to call the SpinTAC Position Converter as part of your ISR.

4.7.3.11 ST_runVelCtl (Velocity Control Only)

This function is used to run SpinTAC Velocity Control in the ISR. This function should be decimated at the rate defined in ISR_TICKS_PER_SPINTAC_TICK. This function should be declared and written in the main source file of the project. The InstaSPIN-MOTION lab projects 05d through 06d provide an example of how to call SpinTAC Velocity Control as part of your ISR.

4.7.3.12 ST_runPosCtl (Position Control Only)

This function is used to run SpinTAC Position Control in the ISR. This function should be decimated at the rate defined in ISR_TICKS_PER_SPINTAC_TICK. This function should be declared and written in the main source file of the project. The InstaSPIN-MOTION lab projects 13a through 13e provide an example of how to call SpinTAC Position Control as part of your ISR.

4.7.3.13 ST_runVelMove (Velocity Control Only)

This function is used to run SpinTAC Velocity Move in the ISR. This function should be decimated at the rate defined in ISR_TICKS_PER_SPINTAC_TICK. This function should be declared and written in the main source file of the project. The InstaSPIN-MOTION lab projects 06a through 06d provide an example of how to call SpinTAC Velocity Move as part of your ISR.

4.7.3.14 ST_runPosMove (Position Control Only)

This function is used to run SpinTAC Position Move in the ISR. This function should be decimated at the rate defined in ISR_TICKS_PER_SPINTAC_TICK. This function should be declared and written in the main source file of the project. The InstaSPIN-MOTION lab projects 13b through 13e provide an example of how to call SpinTAC Position Move as part of your ISR.

4.7.3.15 ST_runVelPlan (Velocity Control Only)

This function is used to run the main component of SpinTAC Velocity Plan. This function can be called in either the ISR or in the main loop of the project. This function should be declared and written in the main source file of the project. The InstaSPIN-MOTION lab projects 06b and 06c provide an example of how to call SpinTAC Velocity Plan as part of your ISR. The InstaSPIN-MOTION lab project 06d provides an example of how to call SpinTAC Velocity Plan as part of your main loop.

4.7.3.16 ST_runVelPlanTick (Velocity Control Only)

This function is used to run the timer component of SpinTAC Velocity Plan in the ISR. This function should be decimated at the rate defined in ISR_TICKS_PER_SPINTAC_TICK. The InstaSPIN-MOTION lab projects 06b through 06d provide an example of how to call this component of SpinTAC Velocity Plan as part of your ISR.

Copyright © 2013–2019, Texas Instruments Incorporated

4.7.3.17 ST_runPosPlan (Position Control Only)

This function is used to run the main component of SpinTAC Position Plan. This function can be called in either the ISR or in the main loop of the project. This function should be declared and written in the main source file of the project. The InstaSPIN-MOTION lab project 13c provides an example of how to call SpinTAC Position Plan as part of your ISR. The InstaSPIN-MOTION lab project13d provides an example of how to call SpinTAC Position Plan as part of your main loop.

4.7.3.18 ST_runPosPlanTick (Position Control Only)

This function is used to run the ISR component of SpinTAC Position Plan in the ISR. This function should be decimated at the rate defined in ISR_TICKS_PER_SPINTAC_TICK. The InstaSPIN-MOTION lab projects 13c through 13d provide an example of how to call this component of SpinTAC Position Plan as part of your ISR.

4.7.3.19 ST_runVelId (Velocity Control Only)

This function is used to run SpinTAC Velocity Identify in the ISR. This function should be decimated at the rate defined in ST_ISR_TICKS_PER_SPINTAC_TICK. This function should be declared and written in the main source file of the project. The InstaSPIN-MOTION lab project 05cincluded in MotorWare provides an example of how to call SpinTAC Velocity Identify as part of the ISR.

4.8 Setting ACIM Motor Parameters in user.h

The parameters provided in user.h for ACIM motors are listed below:

```
#if (USER_MOTOR == User_ACIM)
#define USER_MOTOR_TYPE                 MOTOR_Type_Induction
#define USER_MOTOR_NUM_POLE_PAIRS       (2)
#define USER_MOTOR_Rr                   (5.054793)
#define USER_MOTOR_Rs                   (7.801885)
#define USER_MOTOR_Ls_d                 (0.03334743)
#define USER_MOTOR_Ls_q                 (USER_MOTOR_Ls_d)
#define USER_MOTOR_RATED_FLUX           (0.8165*230.0/60.0)
#define USER_MOTOR_MAGNETIZING_CURRENT  (1.134086)
#define USER_MOTOR_MAX_CURRENT          (5.0)
```

Table 4-1 summarizes all the parameters that are required in user.h header file when ACIM motor identification is bypassed.

Table 4-1. ACIM Motor Parameters in user.h

ACIM Motor Parameter in user.h	ACIM Motor Parameter and Units	ACIM Motor Model Symbol
USER_MOTOR_Rr	Rotor Resistance (Ω)	R_R
USER_MOTOR_Rs	Stator Resistance (Ω)	R_s
USER_MOTOR_Ls_d	Stator Series Inductance (H)	$L_{\sigma S}$
USER_MOTOR_RATED_FLUX	Rated Rotor Flux (V/Hz)	ψ_R
USER_MOTOR_MAGNETIZING_CURRENT	Rated Magnetizing Current (A)	i_{sd}

The following section covers each of these parameters and how to get them from a typical motor manufacturer's datasheet.

Copyright © 2013–2019, Texas Instruments Incorporated

4.8.1 Getting Parameters From an ACIM Datasheet

Figure 4-1 corresponds to an ACIM motor datasheet used as an example. The motor's part number is: 56H17T2011A, from company: Marathon Electric (www.marathonelectric.com).

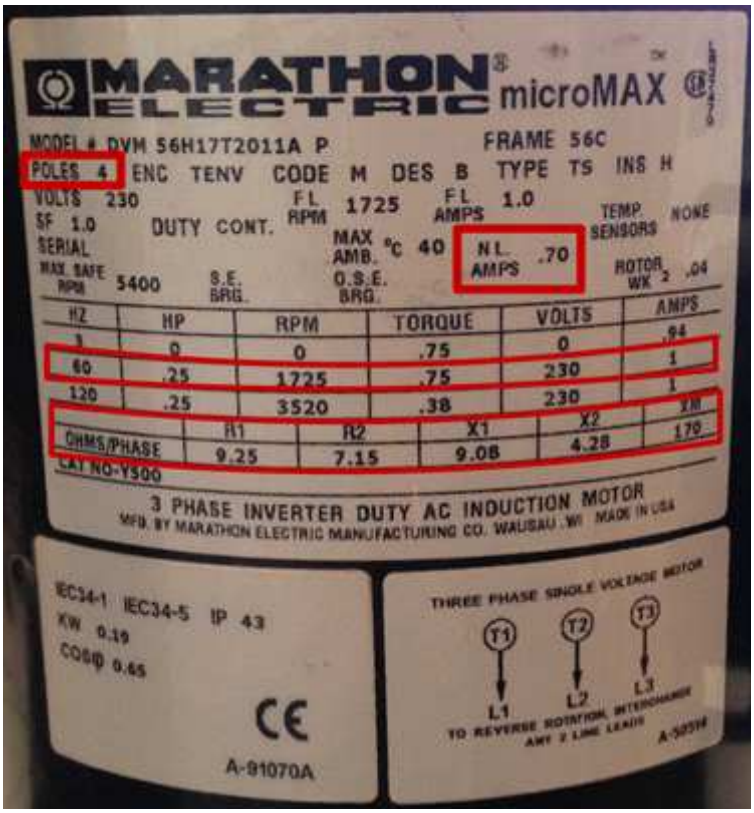


Figure 4-1. Example ACIM Motor Datasheet

4.8.1.1 Number of Pole Pairs

The number of pole pairs is used to calculate speeds in revolutions per minute (RPM) and for some flux calculations, as shown in the rated flux calculation example. We simply use the number of pole pairs from the motor's datasheet into user.h as follows. Keep in mind that sometimes this value is provided in number of poles, so we simply divide number of poles by two to get number of pole pairs:

```
#define USER_MOTOR_NUM_POLE_PAIRS                (2)
```

4.8.1.2 Entering ACIM Motor Parameters From a Datasheet to user.h

With the information we have in the motor's name plate we can enter these values into user.h with the following parameter conversion, assuming a rated frequency of 60 Hz for the impedance calculations.

$$R_R = \left(\frac{XM}{XM + X2}\right)^2 R2 = 6.8031\Omega$$

$$R_S = R1 = 9.25\Omega$$

$$L_{\sigma S} = \frac{XM}{(2\pi)f}\left(\frac{XM + X1}{XM} - \frac{XM}{XM + X2}\right) = 0.0352\ H$$

Where:

R1: Stator resistance (Ω)

Copyright © 2013–2019, Texas Instruments Incorporated

R2: Stator referenced rotor resistance (Ω)

X1: Stator leakage reactance (Ω)

X2: Stator referenced rotor leakage reactance (Ω)

XM: Magnetizing reactance (Ω)

f: Rated frequency of the motor (Hz)

R_R: Stator referenced scaled rotor resistance (Ω)

R_s: Stator resistance (Ω)

$L_{\sigma S}$: Stator series inductance (H)

Note that in order to convert reactance (X) to inductance (L) the user must use the same frequency that was used to define the reactance value itself, which is typically the rated frequency of the motor. In this example, the rated frequency of the motor is 60 Hz, hence *f* = 60 Hz.

With these values we can then enter motor's parameters in user.h:

```
#define USER_MOTOR_Rr                   (6.8031)
#define USER_MOTOR_Rs                   (9.25)
#define USER_MOTOR_Ls_d                 (0.0352)
#define USER_MOTOR_Ls_q                 (USER_MOTOR_Ls_d)
```

There are cases where the ACIM motor parameters are provided in terms of inductance values instead of reactance values. For example, if we convert the values from the motor's datasheet into inductances, we would have the following values:

$$L_{\sigma s} = \frac{X1}{(2\pi)f} = \frac{9.08}{(2\pi)60} = 0.0241\,H$$

$$L_{\sigma r} = \frac{X2}{(2\pi)f} = \frac{4.28}{(2\pi)60} = 0.0114\,H$$

$$L_m = \frac{XM}{(2\pi)f} = \frac{170}{(2\pi)60} = 0.4509\,H$$

And now the conversion from these set of values to what we need in user.h is as follows:

$$R_R = \left(\frac{L_m}{L_m + L_{\sigma r}}\right)^2 R2 = 6.8031\,\Omega$$

$$R_s = R1 = 9.25\,\Omega$$

$$L_{\sigma S} = L_m\left(\frac{L_m + L_{\sigma s}}{L_m} - \frac{L_m}{L_m + L_{\sigma r}}\right) = 0.0352\,H$$

Where:

$L_{\sigma s}$: Stator leakage inductance (H)

$L_{\sigma r}$: Stator referenced rotor leakage inductance (H)

L_m: Magnetizing inductance (H)

4.8.1.3 Getting the Rated Magnetizing Current of an ACIM

To get the rated magnetizing current of an ACIM motor, we can calculated from the no load current specified in a motor's name plate. In this particular example, the no load current (i^{noload}) is 0.7 A, which is usually a value provided in RMS. Usually this no load current is approximately the same as the rated magnetizing current (i_d^{rated}) with $i_q \cong 0$. The value we need to define in user.h is in maximum amplitude, so we calculate it as follows:

Copyright © 2013–2019, Texas Instruments Incorporated

$$i_d^{rated} = i^{noload}\sqrt{2} = 0.7\sqrt{2} = 0.9899 \text{ A}$$

Now we can enter this value in user.h:

```
#define USER_MOTOR_MAGNETIZING_CURRENT                  (0.9899)
```

4.8.1.4 Getting the Rated Flux of an ACIM

To get the rated flux of an ACIM, we need to calculate the rated stator flux based on the name plate values, and subtract the flux produced by the inductance to get the rated rotor flux. The following equation is used to calculate the rated rotor flux:

$$\psi_s^{rated} = \left(\frac{\sqrt{2}}{\sqrt{3}}\right)\frac{v_{line-to-line}^{rated}}{(2\pi)f^{rated}} = \left(\frac{\sqrt{2}}{\sqrt{3}}\right)\frac{230}{(2\pi)60} = 0.4981 \text{ Wb}$$

$$\psi_R^{rated} = \psi_s^{rated} - L_{\sigma S} i_d^{rated} = 0.4981 - (0.0352)(0.9899) = 0.4633 \text{ Wb}$$

$$USER_\psi_R^{rated} = (2\pi)\psi_R^{rated} = 2.9107 \text{ V / Hz}$$

Now we can set this value in user.h:

```
#define USER_MOTOR_RATED_FLUX                   (2.9107)
```


Copyright © 2013–2019, Texas Instruments Incorporated

Managing Motor Signals

For the InstaSPIN FAST observer (flux, rotor flux angle, shaft speed and torque), voltage and current signals from the motor are required to be measured by the ADC. The accuracy of these signals has direct impact on the performance of the observer. This section discusses the required signals, which parameters in the InstaSPIN software are used to configure for these signals and their related circuits. The following sections are considered "prerequisites", required software and hardware configuration for a successful motor identification and the running of InstaSPIN.

Topic **Page**

5.1 Software Prerequisites **223**
5.2 Hardware Prerequisites **228**

Copyright © 2013–2019, Texas Instruments Incorporated

5.1 Software Prerequisites

Below are the parameters that require configuration by in the user's software to manage the motor signals required by InstaSPIN's FAST observer. Each parameter is discussed in this section.

- IQ full-scale frequency - set to motor's max electrical frequency with 20-30% headroom
- IQ full-scale voltage - set to motor's max voltage with 20-30% headroom
- IQ full-scale current - set to motor's max measureable current with 20-30% headroom
- Max Current - set to motor manufacturer's max current (peak) with 0% headroom
- Decimation rates - multiple loop rates and settings
- System Frequency - set to MCU's max CPU clock speed
- PWM Frequency - default is 20 KHz, increase with lower inductance motors
- Max Duty Cycle - 100% duty use 3-shunt current measurements

5.1.1 IQ Full-Scale Frequency

IQ Full-Scale Frequency represents the electrical frequency of the motor in a per unit value. In other words, the electrical frequency of the motor is normalized with the value in this define. It is recommended to have a value of IQ full-scale frequency equal to the absolute maximum electrical frequency that the motor will run in the application.

Set IQ full-scale frequency equal to the motor's absolute maximum electrical frequency.

To illustrate a typical example of this value, consider a PMSM motor with four pole pairs, running at an absolute maximum speed of 15,000 RPM. In this case, the absolute maximum frequency of the motor is 15000/60*4 = 1000 Hz. The following setting of the IQ full-scale frequency is recommended in this case:

```
//! \brief Defines the full-scale frequency for IQ variable, Hz
#define USER_IQ_FULL_SCALE_FREQ_Hz (1000.0)
```

It is important to note that this value must be higher than any allowable frequency in the motor, so it is recommended to add 20-30% headroom to this value, higher than the maximum expected frequency of the motor.

5.1.2 IQ Full-Scale Voltage

Similar to IQ Full-Scale Frequency, the IQ full-scale voltage value is used to normalize all the voltage terms inside of the library to a per unit value. For that reason, this define must be greater than any voltage provided to the motor windings, including voltages present inside the motor. These voltages inside the motor can be greater than the input voltage itself in cases where the motor is operated in field weakening, which is operating the motor beyond its rated speed.

Voltages inside the motor can be greater than the input voltage. Set IQ Full-Scale Frequency greater than any voltage inside the motor.

To illustrate, consider a PMSM motor with a rated speed of 4000 RPM. If the motor is driven with a 24V power supply, and no field weakening is used, all the voltages outside and inside the motor will be equal or less than 24V. However, if field weakening is used to double the speed of the motor to a maximum of 8000 RPM, then inside the motor, the back EMF voltage might be up to twice the input voltage of 24V, reaching up to 48V. In this scenario it is recommended to set the IQ full-scale voltage define to 48V as shown in the following code example:

```
//! \brief Defines the full-scale voltage for the IQ variable, V
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
#define USER_IQ_FULL_SCALE_VOLTAGE_V (48.0)
```

CAUTION

In the following flux calculation of ψ_d^{max} and ψ_q^{max}, if any of these two values is equal to or greater than 2.0, a numerical overflow condition will occur, since this value is represented in a numeric format that has a max integer range of 2 (IQ30 actually has a max value very close to 2 which is ($2 - 2^{-30}$), see the IQmath library for more details on this format).

$$V_{IQ}^{max} = USER_IQ_FULL_SCALE_VOLTAGE_V = ?$$

$$\omega_{filter_pole} = 2 \times \pi \times USER_IQ_FULL_SCALE_FREQ_Hz = 2 \times \pi \times 714.15Hz = 4487.1rad/s$$

$$I_{motor}^{max} = USER_MOTOR_MAX_CURRENT = 4.2A$$

$$L_{s_d} = USER_MOTOR_Ls_d = 0.0006H$$

$$L_{s_q} = USER_MOTOR_Ls_q = 0.0006H$$

$$\psi_d^{max} = I_{motor}^{max} \times L_{s_d} \times \frac{\omega_{filter_pole}}{V_{IQ}^{max}} < 2.0 \overset{yields}{\rightarrow} V_{IQ}^{max} > \frac{1}{2} \times I_{motor}^{max} \times L_{s_d} \times \omega_{filter_pole}$$

$$\psi_q^{max} = I_{motor}^{max} \times L_{s_q} \times \frac{\omega_{filter_pole}}{V_{IQ}^{max}} < 2.0 \overset{yields}{\rightarrow} V_{IQ}^{max} > \frac{1}{2} \times I_{motor}^{max} \times L_{s_q} \times \omega_{filter_pole}$$

In the above example, both inductances are the same ($L_{s_d} = L_{s_q}$), hence the V_{IQ}^{max} voltage must be greater than:

$$V_{IQ}^{max} > \frac{1}{2} \times I_{motor}^{max} \times L_{s_d} \times \omega_{filter_pole}$$

$$V_{IQ}^{max} > \frac{1}{2} \times 4.2A \times 0.0006\ H \times 4487.1\ rad/s$$

$$V_{IQ}^{max} > 5.65\ V$$

It is recommended to have 20-30% headroom on top of this minimum value. In the example above, the motor can be operated up to 48 V, and since this voltage is greater than V_{IQ}^{max} (with headroom) we can just simply set 48 V for our full-scale voltage:

```
//! \brief Defines the full-scale voltage for the IQ variable, V
#define USER_IQ_FULL_SCALE_VOLTAGE_V (48.0)
```

User must select parameters so that this overflow is prevented. If the inductance is unknown, a rough estimation must be used in the above calculation to know if there will be an overflow condition.

In addition to a minimum value set in USER_IQ_FULL_SCALE_VOLTAGE_V, there is a maximum value to be set here also. The maximum value relates to the minimum flux that can be identified by InstaSPIN. The minimum flux that can be identified is calculated as follows:

Minimum Flux (V/Hz) = USER_IQ_FULL_SCALE_VOLTAGE_V/USER_EST_FREQ_Hz/0.7

For example, if a motor has a flux of 0.001 V/Hz (this value is not unusual when working with hobby motors with extremely low flux values), and running the estimator at 20 kHz, then the maximum USER_IQ_FULL_SCALE_VOLTAGE_V that can be used to identify this motor is:

USER_IQ_FULL_SCALE_VOLTAGE_V < 0.001 * 20000 / 0.7 = 28.57 V

Copyright © 2013–2019, Texas Instruments Incorporated

30% of headroom is recommended to allow a stable identification. So in the above example, a USER_IQ_FULL_SCALE_VOLTAGE_V of 20.0V is recommended.

5.1.3 IQ Full-Scale Current

IQ full-scale current serves the same purpose as the previous IQ full-scale values for the frequency and voltage, but for the current feedback. IQ full-scale current is used to normalize the current feedback into a per unit value. This value must be greater than any measurable current.

IQ full-scale current must be greater than any measurable current

For example, if the motor has a peak current value of 8 A per phase, the IQ full-scale value should be set to a higher value with 20-30% headroom, in this example, to 10 A.

```
//! \brief Defines the full-scale current for the IQ variables, A
#define USER_IQ_FULL_SCALE_CURRENT_A (10.0)
```

CAUTION

If the measured current is greater than the IQ full-scale current at any point, there might be a numerical overflow condition in the software. Make sure the measurable current is less than this value to avoid an undesirable software behavior. In order to avoid this issue, user must make sure that (USER_IQ_FULL_SCALE_CURRENT_A * 2) is always greater that the measurable current by the ADC. The "multiply by 2" factor is because the USER_IQ_FULL_SCALE_CURRENT_A parameter ranges from zero to maximum amplitude (peak), while the USER_ADC_FULL_SCALE_ CURRENT_A is from peak to peak.

Following the guideline below prevents numerical overflow on the current measurement:

```
(USER_IQ_FULL_SCALE_CURRENT_A * 2) >= USER_ADC_FULL_SCALE_CURRENT_A
```

5.1.4 Max Current

Max Current defines the maximum output of the speed controller, different from the IQ Full-Scale Current which defines a normalization factor of currents measured with the ADC converter. Max Current must always be lower than IQ Full-Scale Current since Max Current is a software limit only, while IQ Full-Scale Current represents a maximum software representation of a maximum hardware input.

Max Current must always be lower than IQ Full-Scale Current.

The definition of the maximum current sets a maximum software limit. It indicates that the maximum current commanded by the speed controller will be clamped to the maximum current definition. For example, if the maximum current definition is set to 4.2 A, and the speed controller requires an increase on the torque demand through the current controllers, the maximum commanded current will be 4.2 A, or whatever is set in this definition. It is recommended to have a maximum current less than or equal to the maximum recommended current by the motor manufacturer to avoid damage to the motor.

Max current less than or equal to motor manufacturer's recommended max current.

Copyright © 2013–2019, Texas Instruments Incorporated

For example, the Anaheim motor provided with the DRV8312 Revision D board has a rated torque of 21 oz-in and a torque constant of 5 oz-in/A, which leads us to a rated current of 4.2 A to produce rated torque. 4.2 A is set to the motor maximum current (peak current amplitude) is shown in the following code example:

```
#define USER_MOTOR_MAX_CURRENT (4.2)
```

Note that the Max Current defined in user.h does not provide a hardware limitation or protection against over currents. In other words, this is not a hardware current limit, instead, this is a software limit that only limits the maximum input of the current controllers, and not their output.

Figure 5-1 shows a representation of where this USER_MOTOR_MAX_CURRENT is used in InstaSPIN. As can be seen in the diagram, the maximum current does not limit a current cycle by cycle, but it provides a saturation of the speed controller integral portion output as well as a saturation of the overall speed controller output before providing the reference to the current controller.

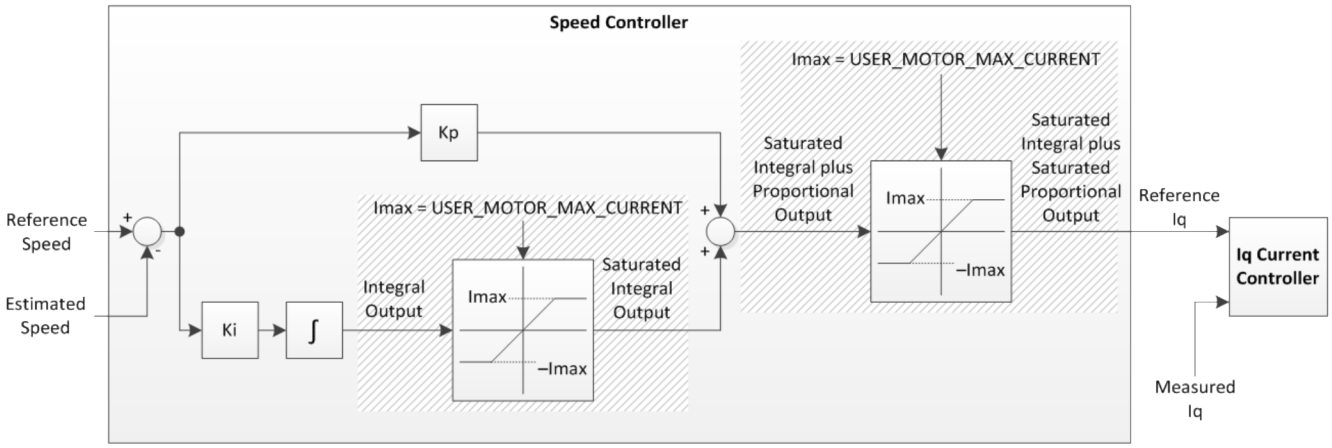


Figure 5-1. USER_MOTOR_MAX_CURRENT in InstaSPIN

5.1.5 Decimation Rates

Decimation rates allow the user to configure each loop rate to meet their code execution requirements. It is recommended to use the default decimation rates as a starting point. The user must verify real-time scheduling is met, verifying that a single interrupt period allows execution of all software in the ISR. This can be done by simply toggling (2) GPIO pins, one at the start and the other at the end of the ISR, and observing on an oscilloscope. If real-time scheduling is not met then InstaSPIN performance is not predictable.

Real-time scheduling is required for consistent InstaSPIN performance.

Below are the default decimation rates:

```
// Defines the number of pwm clock ticks per isr clock tick
// Note: Valid values are 1, 2 or 3 only
#define USER_NUM_PWM_TICKS_PER_ISR_TICK       (1)
// Defines the number of isr ticks (hardware) per controller clock tick (software)
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK      (1)
// Defines the number of controller clock ticks per estimator clock tick
#define USER_NUM_CTRL_TICKS_PER_EST_TICK      (1)
// Defines the number of controller clock ticks per current controller clock tick
#define USER_NUM_CTRL_TICKS_PER_CURRENT_TICK  (1)
// Defines the number of controller clock ticks per speed controller clock tick
#define USER_NUM_CTRL_TICKS_PER_SPEED_TICK    (10)
// Defines the number of controller clock ticks per trajectory clock tick
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
#define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK     (10)
```

If the interrupt period does not allow these decimation rates to complete, see Section 9.1 to learn more about changing the decimation rates.

5.1.6 System Frequency

System Frequency is the clock rate of the MCU. It is recommended to run at the highest frequency possible so that the code is executed as fast as possible. There are two sections where user must configure the system frequency to the maximum.

Configure MCU to fastest CPU clock for best real-time performance.

The first section is in user.h file. This #define will make sure the calculations for all the timing blocks are calculated accordingly.

The second section is in the file hal.c using the function HAL_setParams, see the code example shown in Table 5-1 configuring the PLL to run at a maximum frequency of 90 MHz for the 2806x device, and 60 MHz for the 2805x and 2802x devices.

Table 5-1. hal.c Configuring the PLL

Device	Fastest System Frequency	user.h	hal.c
2806x	90 MHz	`#define USER_SYSTEM_FREQ_MHz (90)`	`HAL_setupPll(handle,PLL_ClkFreq_90_MHz)`
2805x	60 MHz	`#define USER_SYSTEM_FREQ_MHz (60)`	`HAL_setupPll(handle,PLL_ClkFreq_60_MHz)`
2802x	60 MHz	`#define USER_SYSTEM_FREQ_MHz (60)`	`HAL_setupPll(handle,PLL_ClkFreq_60_MHz)`

5.1.7 PWM Frequency

PWM Frequency is set in the file **user.h**. Some motors require more PWM frequency than others, depending on the motor inductance. As a general rule, the lower the inductance of the motor, the higher the PWM frequency needed to avoid too much current ripple. In general, 20 kHz is recommended for the majority of motors, special cases where the PWM frequency is suggested to be higher are discussed in subsequent sections of this document.

Lower inductance motors require higher PWM frequency.

The following code example shows how to set the software for 20 kHz PWM frequency:

```
//! \brief Defines the Pulse Width Modulation (PWM) frequency, kHz
#define USER_PWM_FREQ_kHz (20.0)
```

5.1.8 Max Voltage Vector

Maximum voltage vector is set in the file user.h and is used to set the maximum magnitude for the output of the Id and Iq PI current controllers. The Id and Iq current controller outputs are Vd and Vq. The relationship between Vs, Vd, and Vq is:

Vs = sqrt(Vd^2 + Vq^2)

Vq = sqrt(USER_MAX_VS_MAG^2 - Vd^2)

In this FOC controller, the Vd value is set equal to:

USER_MAX_VS_MAG*USER_VD_MAG_FACTOR.

Copyright © 2013–2019, Texas Instruments Incorporated

USER_MAX_VS_MAG_PU can go up to 1.0, in global IQ format, or _IQ(1.0), if current reconstruction is not used. For further discussion and examples, see Labs 10a-x.

```
//! \brief Defines the voltage vector magnitude
#define USER_MAX_VS_MAG_PU (1.0)
```

Besides this definition of the maximum voltage vector magnitude, a member of the controller object can be changed to allow changing the output of the current controllers, which is then the input of the space vector modulation (SVM). This is important to note, because even though the maximum voltage vector magnitude is defined in user.h to be a maximum of 1.0 (or 100%), the inputs to the SVM can go up to 4.0/3.0 = 1.3333 allowing overmodulation. An input into the SVM above 1.0 is in the overmodulation region. An input of 2/SQRT(3) = 1.1547 is where the crest of the sine wave touches the 100% duty cycle. At an input of 1.3333, the SVM generator produces a trapezoidal waveform. The following code example changing the output of the current controllers to 1.3333 allowing maximum overmodulation:

```
// Set the maximum current controller output for the Iq and Id current
// controllers to enable overmodulation.
CTRL_setMaxVsMag_pu(ctrlHandle, _IQ(pUserParams->maxVsMag_pu));
```

Table 5-2 describes the different ranges of the maximum SVM input and what it means for the space vector modulation module (SVM).

Table 5-2. Maximum SVM Input Ranges

#define USER_MAX_VS_MAG_PU (value)	CTRL_setMaxVsMag_pu(handle, value)	Duty Cycle on EPWM at Peak	Waveform Type	Current Reconstruction
(1.0)	IQ(1.0)	86.6%	Perfect Sinusoidal	Not needed
$\frac{2}{\sqrt{3}} = 1.1547$	_IQ(2/SQRT(3))	100.0%	Quasi-Sinusoidal	Required
$\frac{4}{3} = 1.3333$	_IQ(1.3333)	100.0%	Trapezoidal	Required

When operating in the overmodulation region, voltage waveforms start turning from sinusoidal to trapezoidal depending on how much overmodulation is applied. Motor vibration and torque ripple should be expected as motor operation goes deeper into overmodulation. The SVM module is explained in detail in Table 8-1.

Maximum Voltage Magnitude that causes a peak duty cycle of 100% requires three shunt current measurement.

The actual duty cycle range caused by the maximum voltage magnitude depends on the number of shunt resistors used to sample the currents. If 100% duty cycles are required by the application, user must use three shunt resistors to sample the phase currents of the motor. Having only two shunt resistors limits the duty cycle to be less than 100%. The maximum duty cycle allowed when using two shunt resistors depends on the OPAMP parameters and the layout itself. The details of choosing the right components for the current feedback are covered in Chapter 17.

5.2 Hardware Prerequisites

There are a few hardware dependent parameters that need to be set correctly in order to identify the motor properly and run the motor effectively using InstaSPIN. The following parameters are related to this, each will be discussed in detail:

- Current feedback gain - maximize ADC input range
- Current feedback polarity - match software with hardware polarity
- Voltage feedback

Copyright © 2013–2019, Texas Instruments Incorporated

- Voltage filter pole
- Number of shunt resistors
- Dead-time configuration
- Analog inputs configuration
- PWM outputs configuration

The following sections describe each one of these parameters.

5.2.1 Current Feedback Gain

In order to measure bidirectional currents, that is, positive and negative currents, the circuits below require a reference voltage of 1.65 V. This voltage is generally not available in 3.3-V systems, but can be created very easily by a voltage follower. Figure 5-2 is a circuit example that generates a 1.65-V reference from a 3.3-V input, which is available in 3.3-V systems. For subsequent circuits connecting to 1.65 V, this circuit is assumed to be used.

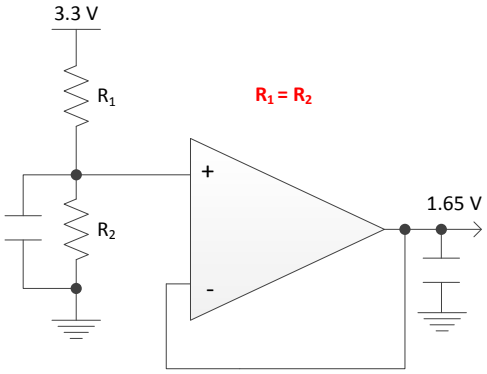


Figure 5-2. 1.65-V Reference from 3.3-V Input Circuit Example

Figure 5-3 shows a typical differential amplifier configuration for the current measurement.

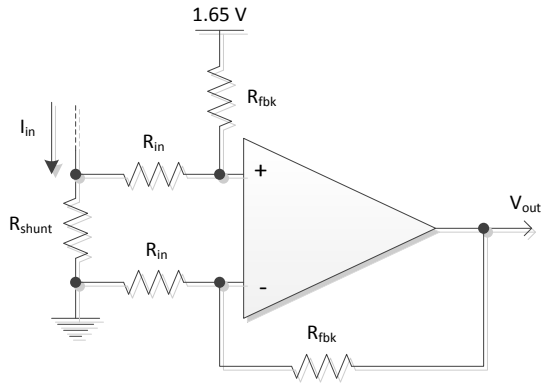


Figure 5-3. Typical Differential Amplifier Circuit

The transfer function of this circuit is given by Equation 3.

$$V_{out} = 1.65 + I_{in} \times R_{shunt} \times \frac{R_{fbk}}{R_{in}} \quad (3)$$

In order to illustrate some values on this circuit, let's consider a motor with a maximum phase current defined to be 10 A. The maximum current to be measured by the microcontroller in this example is ±10 A with this circuit producing a maximum voltage of 1.65V (±1.65V) to support the 3.3 V ADC input range.

For the worst case of 10A, consider a 0.01-Ohm shunt resistor.

Copyright © 2013–2019, Texas Instruments Incorporated

$$\frac{R_{fbk}}{R_{in}} = \frac{V_{out} - 1.65}{I_{in} \times R_{shunt}} = \frac{3.3 - 1.65}{10 \times 0.01} = 16.5 \quad (4)$$

Now if we let the input resistance be 1.0 kOhm, we can calculate the feedback resistance based on the input resistance and the required ratio.

$$R_{fbk} = 16.5 \times R_{in} = 16.5 k\Omega \quad (5)$$

The calculated resistance values lead to the circuit shown in Figure 5-4, providing a voltage range of 0-3.3 V to represent a measured phase current of ±10 A.

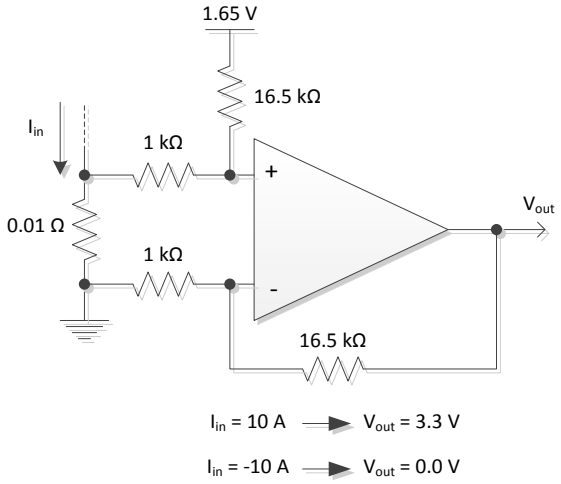


Figure 5-4. Calculated Resistance Values Circuit

As shown in this example, the maximum peak to peak current measurable by the microcontroller is 20 A, which is the peak to peak value of ±10 A. The following code snippet shows how this is defined in user.h:

```
//! \brief Defines the maximum current at the AD converter
#define USER_ADC_FULL_SCALE_CURRENT_A (20.0)
```

The slew rate of the OPAMP plays an important role in the current measurement quality. For more details, see Chapter 17.

5.2.2 Current Feedback Polarity

Correct polarity of the current feedback is also important so that the microcontroller has an accurate current measurement.

5.2.2.1 Positive Feedback

In this hardware configuration, the negative pin of the shunt resistor, which is connected to ground, is also connected to the **inverting** pin of the operational amplifier. Figure 5-5 shows a positive polarity in hardware and its configuration in software. The highlighted sign is required to be configured in order to have correct polarity for the current feedback in software. The code is shown below, a variable "current_sf" is negative in HAL_readAdcData( ) for inverting operation. For positive feedback, the current offsets are negative in **user.h** to correct the polarity mismatch. The lines of code shown below illustrate offset negation done in **user.h**. Functions HAL_readAdcData( ) is located in **hal.h** file. No change is needed in these functions.

Copyright © 2013–2019, Texas Instruments Incorporated

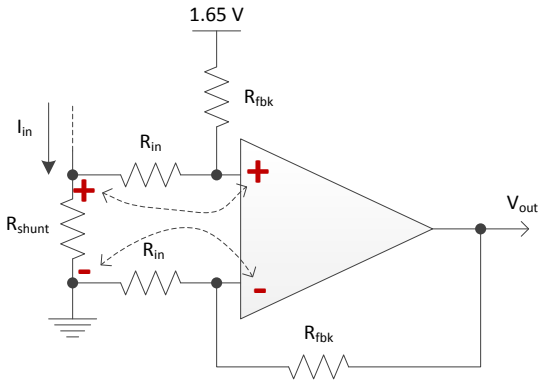


Figure 5-5. Positive Feedback

```
static inline void HAL_readAdcData(HAL_Handle handle,HAL_AdcData_t *pAdcData)
{
  HAL_Obj *obj = (HAL_Obj *)handle;

  _iq value;
  _iq current_sf = -HAL_getCurrentScaleFactor(handle);
  _iq voltage_sf = HAL_getVoltageScaleFactor(handle);

  // convert current A
  // sample the first sample twice due to errata sprz342f, ignore the first sample
  value = (_iq)ADC_readResult(obj->adcHandle,ADC_ResultNumber_1);
  value = _IQ12mpy(value,current_sf) - obj-
>adcBias.I.value[0];      // divide by 2^numAdcBits = 2^12
  pAdcData->I.value[0] = value;

  // convert current B
  value = (_iq)ADC_readResult(obj->adcHandle,ADC_ResultNumber_2);
  value = _IQ12mpy(value,current_sf) - obj-
>adcBias.I.value[1];      // divide by 2^numAdcBits = 2^12
  pAdcData->I.value[1] = value;
```

```
#define   I_A_offset    (-0.8641905189)
#define   I_B_offset    (-0.8677659631)
#define   I_C_offset    (-0.8685645461)
```

5.2.2.2 Negative Feedback

On the other hand, Figure 5-6 represents a negative feedback. In this hardware configuration, the negative pin of the shunt resistor, which is connected to ground, is also connected to the **noninverting** pin of the operational amplifier. The code is shown below, a variable "current_sf" is positive in HAL_readAdcData( ) for noninverting operation. For negative feedback, the current offsets are positive in **user.h**. The lines of code shown below illustrate the offset in **user.h**. Functions HAL_readAdcData( ) is located in **hal.h** file; no change is needed in these functions also.

Copyright © 2013–2019, Texas Instruments Incorporated

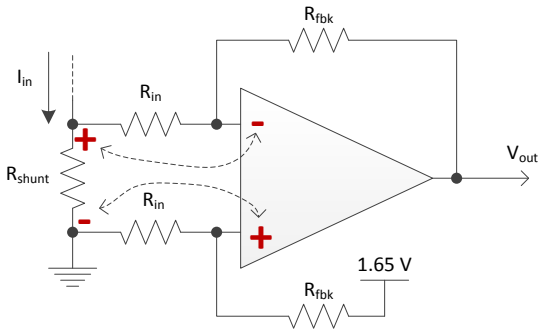


Figure 5-6. Negative Feedback

```
static inline void HAL_readAdcData(HAL_Handle handle,HAL_AdcData_t *pAdcData)
{
  HAL_Obj *obj = (HAL_Obj *)handle;

  _iq value;
  _iq current_sf = HAL_getCurrentScaleFactor(handle);
  _iq voltage_sf = HAL_getVoltageScaleFactor(handle);

  // convert current A
  // sample the first sample twice due to errata sprz342f, ignore the first sample
  value = (_iq)ADC_readResult(obj->adcHandle,ADC_ResultNumber_1);
  value = _IQ12mpy(value,current_sf) - obj-
>adcBias.I.value[0];      // divide by 2^numAdcBits = 2^12
  pAdcData->I.value[0] = value;

  // convert current B
  value = (_iq)ADC_readResult(obj->adcHandle,ADC_ResultNumber_2);
  value = _IQ12mpy(value,current_sf) - obj-
>adcBias.I.value[1];      // divide by 2^numAdcBits = 2^12
  pAdcData->I.value[1] = value;
```

```
#define   I_A_offset    (0.9902871847)
#define   I_B_offset    (0.9889662266)
#define   I_C_offset    (0.9897354245)
```

5.2.3 Voltage Feedback

Voltage feedback is needed in the FAST estimator to allow the best performance at the widest speed range. Other algorithms rely on software variables which fail to represent the voltage phases accurately. In FAST, phase voltages are measured directly from the motor phases instead of a software estimate. This is why the hardware setting for voltage feedback is another prerequisite for InstaSPIN and motor identification. This software value (USER_ADC_FULL_SCALE_VOLTAGE_V) depends on the circuit that senses the voltage feedback from the motor phases. Figure 5-7 is an example of a voltage feedback circuit based on resistor dividers.

Copyright © 2013–2019, Texas Instruments Incorporated

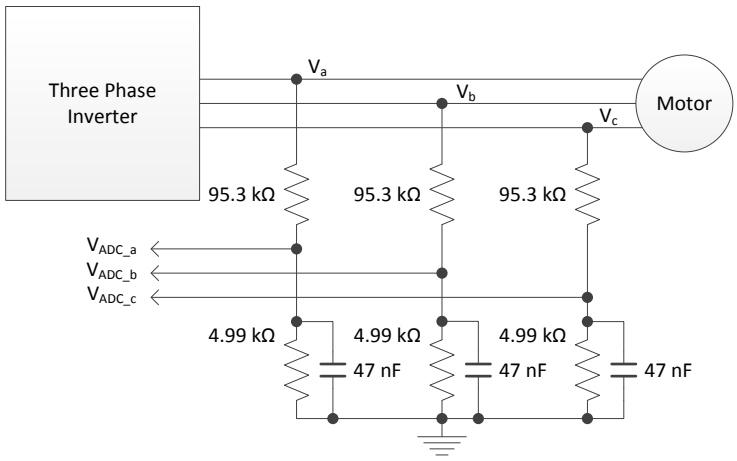


Figure 5-7. Voltage Feedback Circuit

The maximum phase voltage feedback measurable by the microcontroller in this example can be calculated as follows, considering the maximum voltage for the ADC input is 3.3V:

$$V_a^{max} = V_{ADC_a}^{max} \times \frac{(4.99k\Omega + 95.3k\Omega)}{4.99k\Omega} = 3.3\,V \times \frac{(4.99k\Omega + 95.3k\Omega)}{4.99k\Omega} = 66.3\,V \quad (6)$$

With that voltage feedback circuit, the following setting is done in user.h:

```
//! \brief Defines the maximum voltage at the input to the AD converter
#define USER_ADC_FULL_SCALE_VOLTAGE_V (66.3)
```

If we consider 20-30% headroom for this value, the maximum voltage input to the system is recommended to be between 66.3*0.7 = 46.4 V and 66.3*0.8 = 53, so for a motor of 48 V this voltage feedback resistor divider is ideal.

An example of a different nominal voltage is given next. If the motor to be driven has a nominal voltage of 24 V, then the voltage feedback circuit needs to be modified so that the ADC resolution is maximized for the measured voltage. Following the same recommendation for headroom, consider a nominal of 24 V, and a headroom value of 30%. This gives us a USER_ADC_FULL_SCALE_VOLTAGE_V of 24*1.3 = 31.2 V which is represented in Equation 7, where we fix one of the resistors to leave only one variable.

$$V_a^{max} = 31.2\,V = V_{ADC_a}^{max} \times \frac{(4.99k\Omega + R)}{4.99k\Omega} = 3.3\,V \times \frac{(4.99k\Omega + R)}{4.99k\Omega}$$

$$R = \frac{31.2\,V \times 4.99k\Omega}{3.3\,V} - 4.99k\Omega = 42.2k\Omega \quad (7)$$

That would give us the maximum voltage of 31.2 V configured as follows:

```
//! \brief Defines the maximum voltage at the input to the AD converter
#define USER_ADC_FULL_SCALE_VOLTAGE_V (31.2)
```

The voltage feedback circuit with those values is represented in Figure 5-8.

Copyright © 2013–2019, Texas Instruments Incorporated

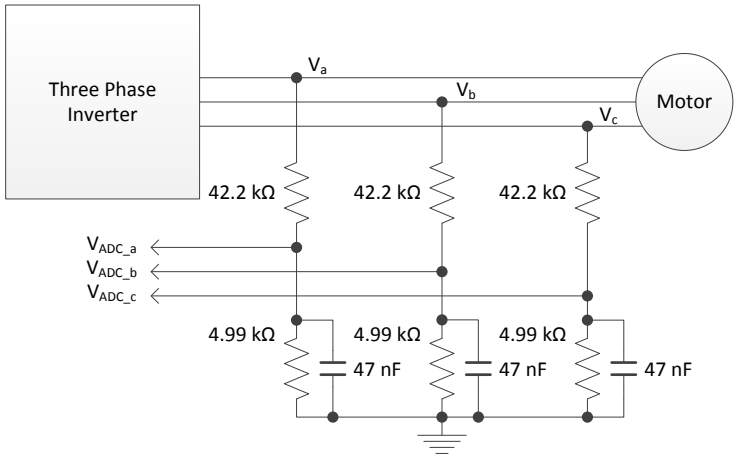


Figure 5-8. Voltage Feedback Circuit

5.2.4 Voltage Filter Pole

The voltage filter pole is needed by the FAST estimator to allow an accurate detection of the voltage feedback. The filter should be low enough to filter out the PWM signals, and at the same time allow a high-speed voltage feedback signal to pass through the filter.

As a general guideline, a cutoff frequency of a few hundred Hz is enough to filter out a PWM frequency of 10 to 20 kHz. The hardware filter should only be changed when ultra-high-speed motors are run, which generate phase-voltage frequencies in the order of a few kHz.

In this example, consider the Anaheim PMSM motor with a maximum speed of 8000 RPM with 4 pole pairs (533 Hz), along with the BOOSTXL-DRV8305 EVM with PWM running at 45 kHz. Considering the BOOSTXL-DRV8305 EVM hardware seen in HALF-BRIDGES & BEMF SENSE section of Figure 13 in the *BOOSTXL-DRV8305EVM User's Guide*, also shown below in Section 6.6.2.

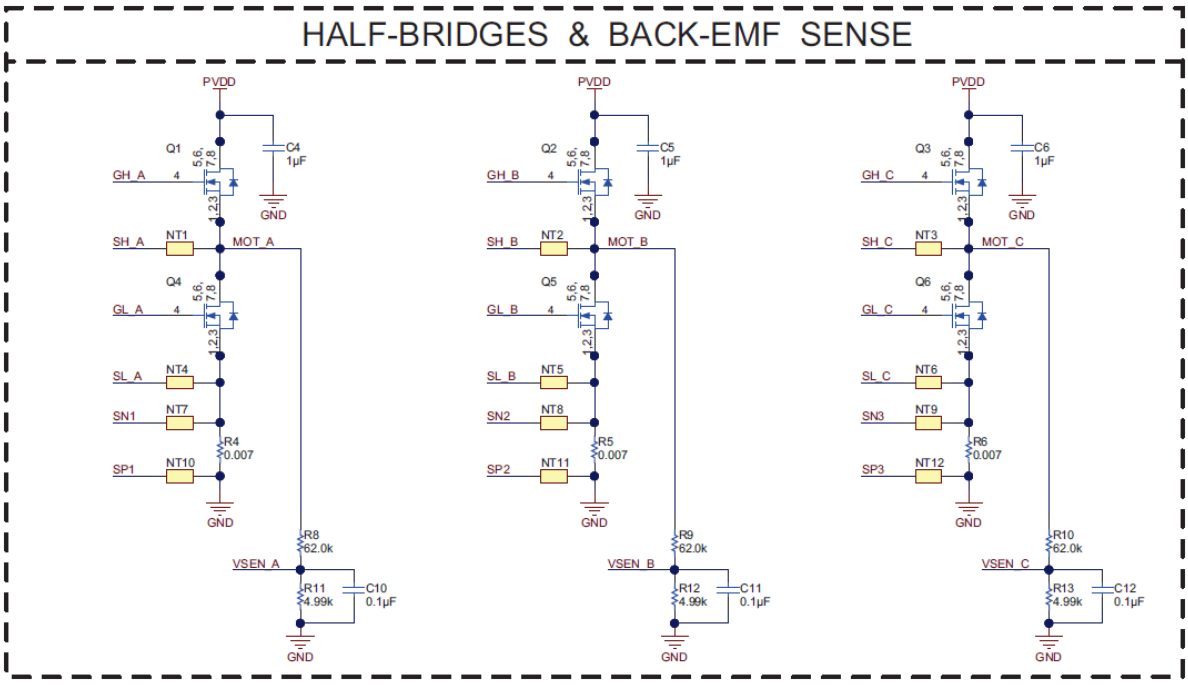


Figure 5-9. BOOSTXL-DRV8305 EVM HALF-BRIDGES & BACK-EMF SENSE Circuit

Copyright © 2013–2019, Texas Instruments Incorporated

The filter pole setting can be calculated as follows:

$$F_{filter_pole} = \frac{1}{2\pi \times R_{parallel} \times C} = \frac{1}{2\pi \times \left(\frac{62k\Omega \times 4.99k\Omega}{62k\Omega + 4.99k\Omega} \right) \times 0.1\mu F} = 344.618Hz \quad (8)$$

The following code example shows how this is defined in user.h:

```
#define USER_VOLTAGE_FILTER_POLE_Hz (344.62)
```

NOTE:

- Typical values for USER_VOLTAGE_FILTER_POLE_Hz fall between 300 Hz < pole < 400 Hz.
- USER_VOLTAGE_FILTER_POLE_Hz ≥ 200 Hz.
- IQ_FULLSCALE_FREQUENCY_Hz must be set to a value < 4.0 × USER_VOLTAGE_FILTER_POLE_Hz to avoid numerical saturation.

5.2.5 Number of Shunt Resistors

An important hardware configuration choice is the number of shunt resistors to use. This number is ultimately used by the Clarke transform to convert from a three phase system to a two phase system. Three shunt resistors are used if all of the phases have a shunt resistor from the bottom transistor to ground as shown in Figure 5-10.

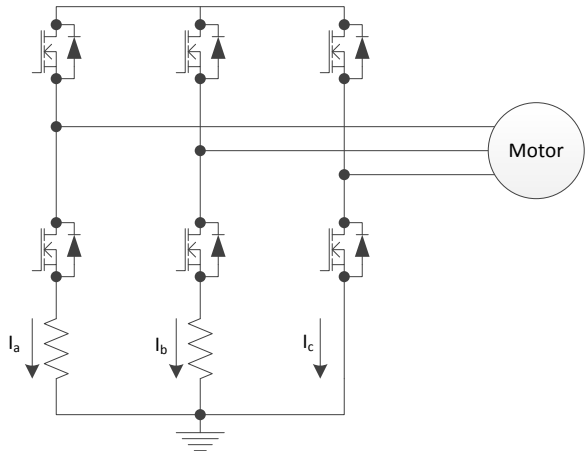


Figure 5-10. Shunt Resistors

If this configuration is present, the user should define three shunt resistors in the software for best results. The following code example shows how to configure the software to use three shunt resistors:

```
//! \brief Defines the number of current sensors used
#define USER_NUM_CURRENT_SENSORS (3)
```

If the hardware has two shunt resistors to measure the currents, the software must be configured for only two shunt resistors as follows:

```
//! \brief Defines the number of current sensors used
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
#define USER_NUM_CURRENT_SENSORS (2)
```

For more details about shunt resistor measurement requirements of InstaSPIN, see Chapter 17.

5.2.6 Dead-Time Configuration

Depending on the hardware used, the dead time must be configured correctly in order to avoid shoot-through between high-side and low-side transistors within the inverter (see Figure 5-11). For more details about EPWM module and dead-time configuration, see the microcontroller technical reference manual.

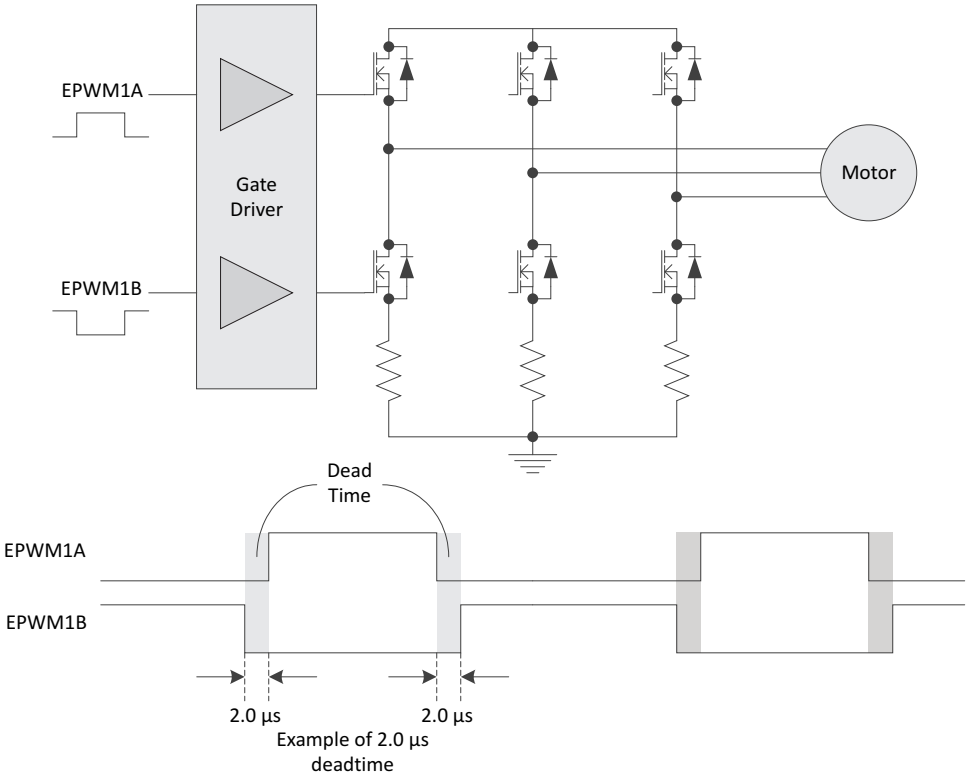


Figure 5-11. Dead-Time Configuration

Dead time depends on the transistor and gate driver circuit used, and is configured as shown in the following code example, based on system clock delay counts:

```
//! \bried Defines the PWM deadband falling edge delay count (system clocks)
//!
#define HAL_PWM_DBFED_CNT

//! \brief Defines the PWM deadband rising edge delay count (system clocks)
//!
#define HAL_PWM_DBRED_CNT
```

```
// setup the Dead-Band Rising Edge Delay Register (DBRED)
PWM_setDeadBandRisingEdgeDelay(obj->pwmHandle[cnt],HAL_PWM_DBRED_CNT);

// setup the Dead-Band Falling Edge Delay Register (DBFED)
PWM_setDeadBandFallingEdgeDelay(obj->pwmHandle[cnt],HAL_PWM_DBFED_CNT);
```


Copyright © 2013–2019, Texas Instruments Incorporated

Trip zones and comparators used to protect the hardware against overcurrent or overvoltage conditions depend on the particular hardware used, and it is the responsibility of the end user to make use of all the available feature of the EPWM and ADC modules to protect the hardware. Also, alternative dead-time implementation scenarios can be accomplished with the flexibility of the EPWM module, however, the scope of this document is limited to the functionality of InstaSPIN software, and does not cover all the EPWM implementation scenarios.

5.2.7 Analog Inputs Configuration

The analog pins must be configured in the software. To illustrate this, consider Figure 5-12, representing how the analog pins are connected when using a F2806xF device with a DRV8312 Revision D development board.

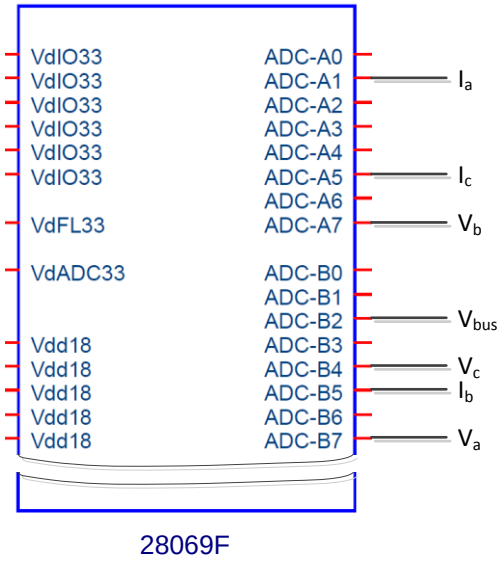


Figure 5-12. Analog Connections

For more details about the ADC configuration, see the *TMS320x2806x Piccolo Technical Reference Guide* (SPRUH18), the *TMS320x2805x Piccolo Technical Reference Guide* (SPRUHE5), and the *TMS320F2802x, TMS320F2802xx Piccolo System Control and Interrupts Reference Guide* (SPRUFN3). For more details about pins related to specific packages, see the *TMS320F2806x Piccolo Microcontrollers* data manual (SPRS698), the *TMS320F2805x Piccolo Microcontrollers* data manual (SPRS797), and the *TMS320F2802x, TMS320F2802xx Piccolo Microcontrollers* data manual (SPRS523).

The following code example shows how to configure the ADC input pins to represent the previous diagram. The lines of code doing the actual configuration are highlighted in the code example. This code is part of the function HAL_setupAdcs of **hal.c** file.

```
ADC_setSocChanNumber (obj->adcHandle,ADC_SocNumber_0,ADC_SocChanNumber_A1);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_0,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_0,ADC_SocSampleDelay_9_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_1,ADC_SocChanNumber_B5);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_1,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_1,ADC_SocSampleDelay_9_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_2,ADC_SocChanNumber_A5);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_2,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_2,ADC_SocSampleDelay_9_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_3,ADC_SocChanNumber_B7);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_3,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_3,ADC_SocSampleDelay_9_cycles);
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_4,ADC_SocChanNumber_A7);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_4,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_4,ADC_SocSampleDelay_9_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_5,ADC_SocChanNumber_B4);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_5,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_5,ADC_SocSampleDelay_9_cycles);

ADC_setSocChanNumber(obj->adcHandle,ADC_SocNumber_6,ADC_SocChanNumber_B2);
ADC_setSocTrigSrc(obj->adcHandle,ADC_SocNumber_6,ADC_SocTrigSrc_EPWM1_ADCSOCA);
ADC_setSocSampleDelay(obj->adcHandle,ADC_SocNumber_6,ADC_SocSampleDelay_9_cycles);
```

5.2.8 PWM Outputs Configuration

Figure 5-13 represents a configuration of the PWM pins, where phase A of the motor is driven by EPWM1A/EPWM1B, motor phase B by EPWM2A/EPWM2B, and motor phase C by EPWM3A/EPWM3B. This configuration in hardware is the same as the DRV8312 Revision D board, and it is taken as an example in this document.

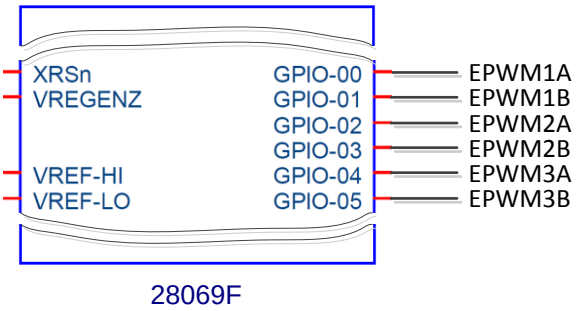


Figure 5-13. PWM Pin Configuration

For more details about the EPWM configuration, see the *TMS320x2806x Piccolo Technical Reference Guide* (SPRUH18), the *TMS320x2805x Piccolo Technical Reference Guide* (SPRUHE5), and the *TMS320F2802x, TMS320F2802xx Piccolo System Control and Interrupts Reference Guide* (SPRUFN3). For more details about pins related to specific packages, see the *TMS320F2806x Piccolo Microcontrollers* data manual (SPRS698), the *TMS320F2805x Piccolo Microcontrollers* data manual (SPRS797), and the *TMS320F2802x, TMS320F2802xx Piccolo Microcontrollers* data manual (SPRS523).

The following code example shows how to setup the PWM pins for the diagram shown above. This code example is part of the initialization of the driver object (HAL_init), contained in **hal.c** file. Once the handles are initialized to use the first three EPWM pairs, the rest of the PWM configuration is done to the initialized handles:

```
// initialize PWM handle
obj->pwmHandle[0] = PWM_init((void *)PWM_ePWM1_BASE_ADDR,sizeof(PWM_Obj));
obj->pwmHandle[1] = PWM_init((void *)PWM_ePWM2_BASE_ADDR,sizeof(PWM_Obj));
obj->pwmHandle[2] = PWM_init((void *)PWM_ePWM3_BASE_ADDR,sizeof(PWM_Obj));
```

The highlighted text indicates the correspondence between:

- pwmHandle[0] = EPWM1A/EPWM1B, which will drive phase A of the motor
- pwmHandle[1] = EPWM2A/EPWM2B, which will drive phase B of the motor
- pwmHandle[2] = EPWM3A/EPWM3B, which will drive phase C of the motor.

Copyright © 2013–2019, Texas Instruments Incorporated

Motor Identification and State Diagrams

We will look at the motor identification function comparing the process differences with identifying ACIM and PMSM, looking at the detail of each process, and discussing the special cases that require non-typical procedures.

Some sensorless motor control applications rely on a motor model. This motor model requires knowledge of certain motor parameters in order to have an accurate representation of the motor. This model is then used to run an estimator, which will then provide the unknown variables such as rotor flux angle or speed. A problem occurs when the motor parameters are unknown, or if the parameters change over time. Typically a well-defined motor is used for an end application, but in some cases, the motor has not been defined or there are several motors used during the product's life.

A way to determine the motor parameters and keep track of them is to have software that measures motor parameters. Although identifying the motor is not a must for all applications, it provides an easy and better out of the box experience to run any given motor sensorlessly. Other algorithms in the marketplace require an intensive tuning process upfront, even before running the motor in closed loop at all.

The following sections describe in detail the process of motor identification with the InstaSPIN solution. Significant efforts have been spent on ensuring both the algorithms and steps describe will successfully identify a large number of motor types. But, one should not expect that the algorithms and steps described will always successfully identify all motors or motor types. The troubleshooting section mentions a few specific motor types that require special attention.

Topic **Page**

6.1 InstaSPIN Motor Identification 240
6.2 Motor Identification Process Overview 242
6.3 Differences between PMSM and ACIM Identification Process 248
6.4 Prerequisites 249
6.5 Full Identification of PMSM Motors 252
6.6 Full Identification of ACIM Motors 270
6.7 Recalibration of PMSM and ACIM Motor Identification 285
6.8 Setting PMSM Motor Parameters in user.h 293
6.9 Troubleshooting Motor Identification 297

Copyright © 2013–2019, Texas Instruments Incorporated

6.1 InstaSPIN Motor Identification

The highlighted blocks in Figure 6-1 are related to the InstaSPIN Motor Identification feature.

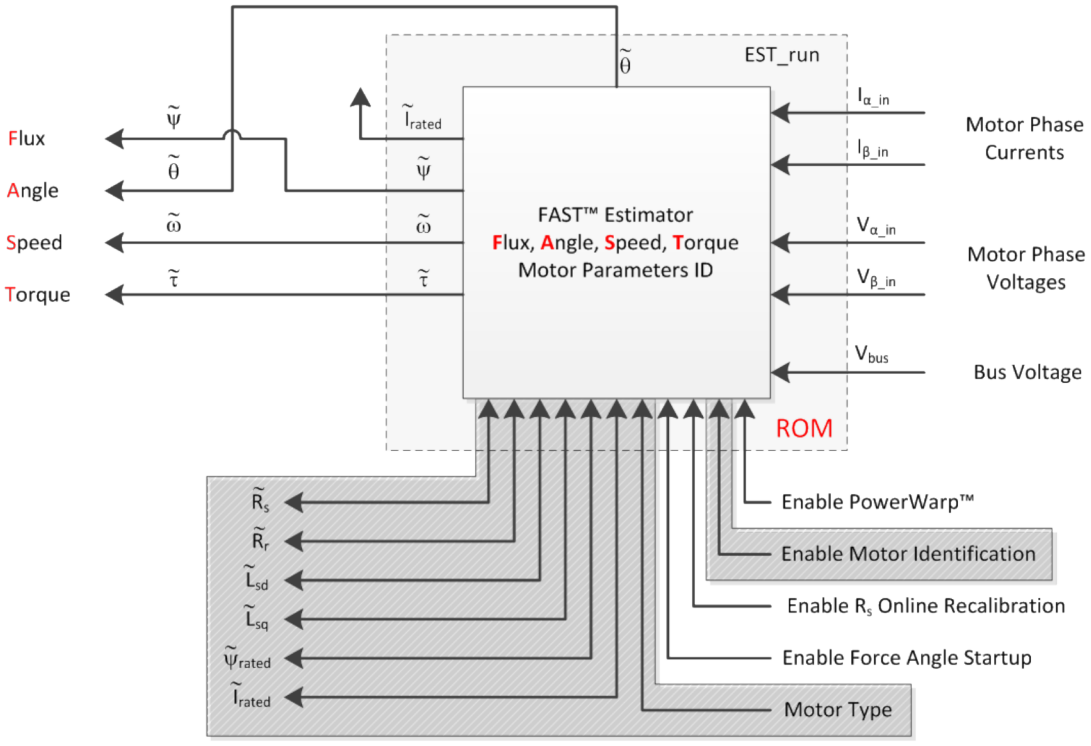


Figure 6-1. InstaSPIN Motor Identification Components

Motor identification is a feature added to InstaSPIN-FOC that allows the identification of the parameters needed by the estimator to run in closed loop sensorlessly. If the motor parameters are well known by the user, motor identification is optional. The motor identification feature of InstaSPIN enables users to run their motor to its highest performance even when motor parameters are unknown. In the case of a known motor or a previously identified motor, running InstaSPIN's motor identification is optional, since the required motor parameters can be recorded in a header file. An example of such a header file is user.h. The following example shows the required parameters for PMSM motors when bypassing motor identification:

```
#if (USER_MOTOR == User_PMSM)
#define USER_MOTOR_Rs (2.83)
#define USER_MOTOR_Ls_d (0.0115)
#define USER_MOTOR_Ls_q (0.0135)
#define USER_MOTOR_RATED_FLUX (0.502)
```

The following example shows the required parameters for ACIM motors when bypassing motor identification:

```
#elif (USER_MOTOR == User_ACIM)
#define USER_MOTOR_Rr (5.5)
#define USER_MOTOR_Rs (10.7)
#define USER_MOTOR_Ls_d (0.053)
#define USER_MOTOR_Ls_q USER_MOTOR_Ls_d
#define USER_MOTOR_MAGNETIZING_CURRENT (1.4)
```


Copyright © 2013–2019, Texas Instruments Incorporated

For more details about the motor parameters needed for PMSM and ACIM motors see Chapter 4.

Motor identification can be run with both full and minimum implementation of InstaSPIN, see Figure 6-2 and Figure 6-3. When running motor identification with InstaSPIN's minimum implementation users must include the provided blocks for field oriented control (FOC) included in InstaSPIN open source library.

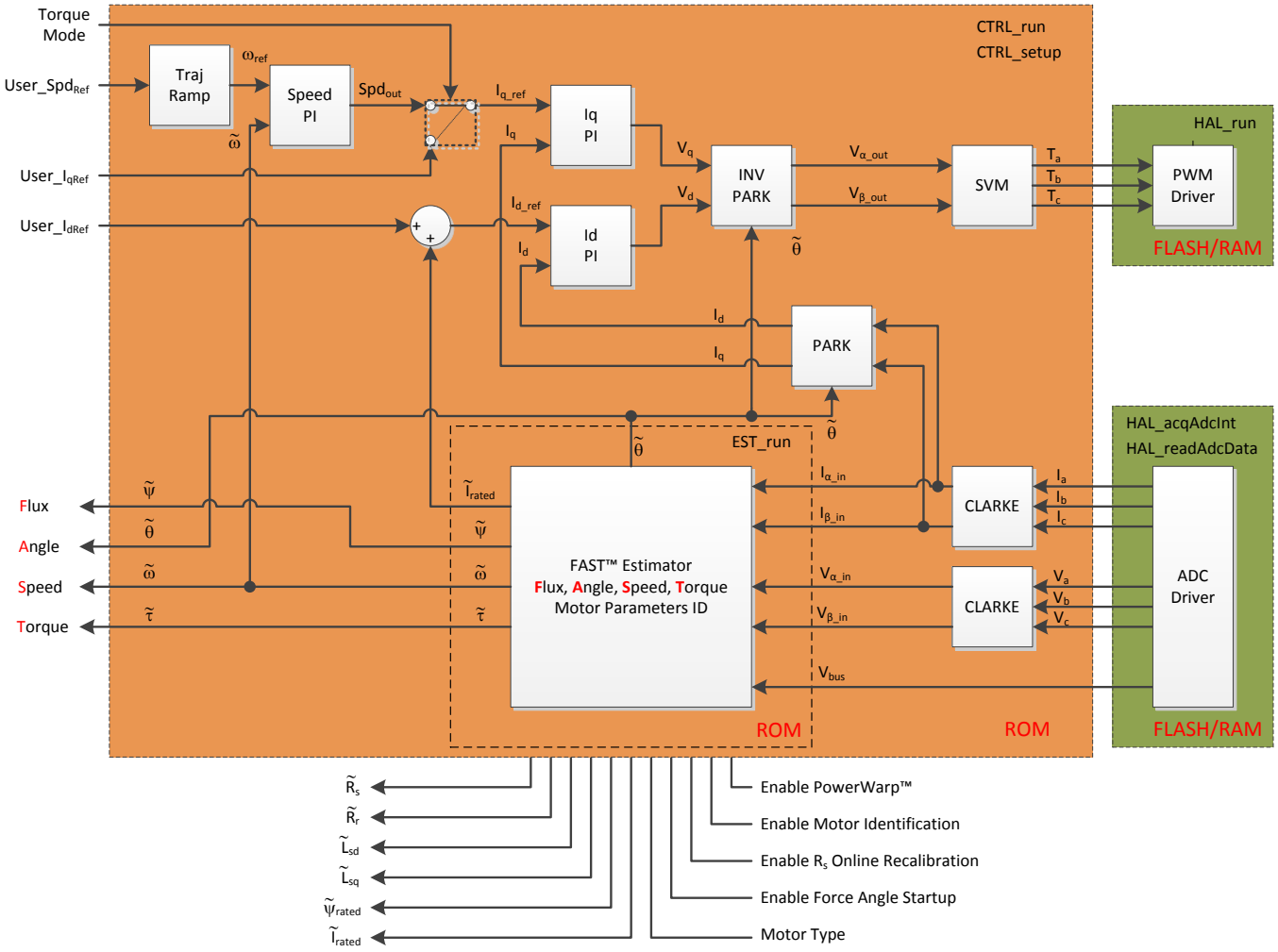


Figure 6-2. Full Implementation of InstaSPIN-FOC (F2805xF, F2805xM, F2806xF, and F2806xM Devices Only)

Copyright © 2013–2019, Texas Instruments Incorporated

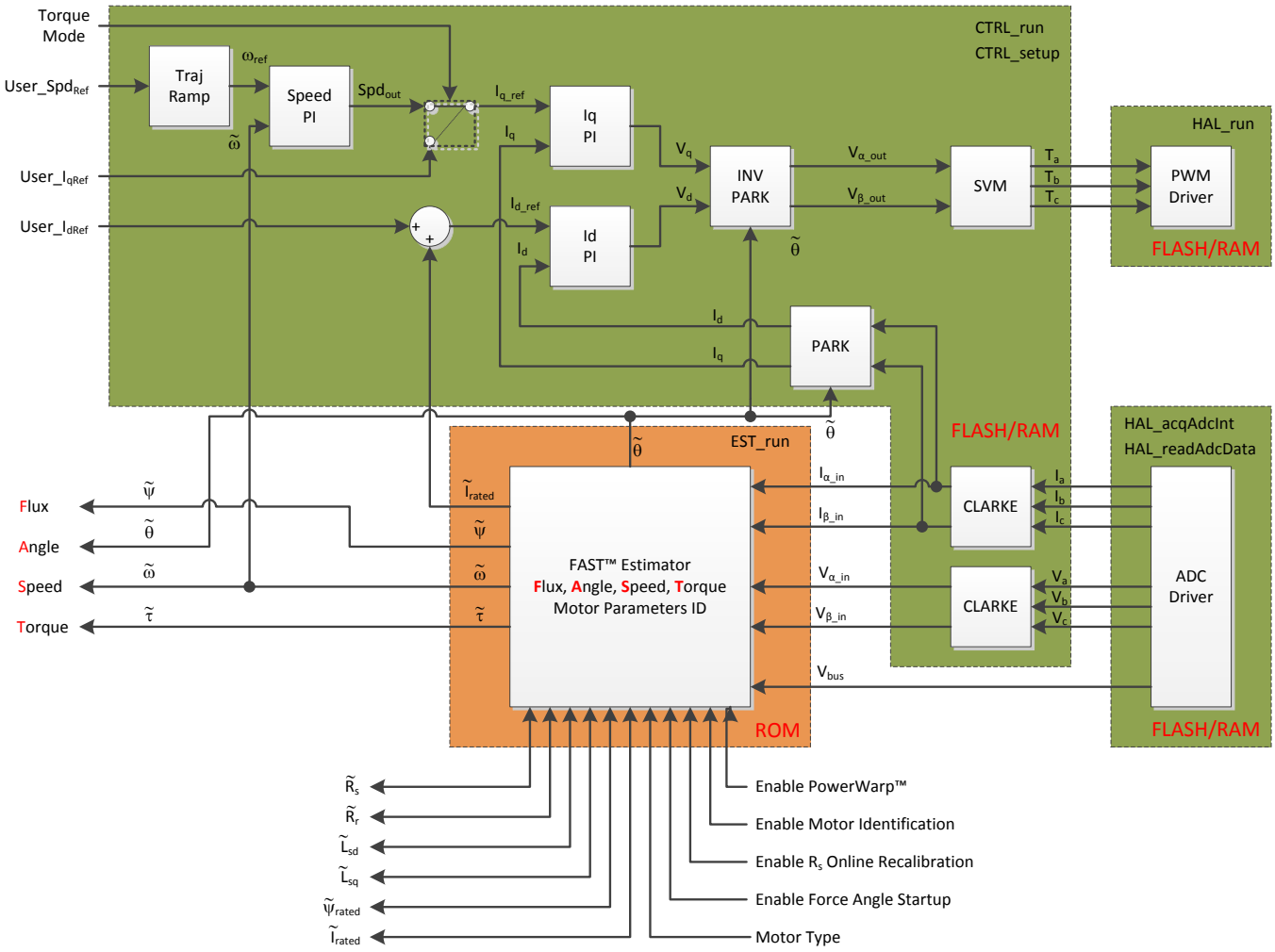


Figure 6-3. Minimum Implementation of InstaSPIN-FOC (F2802xF, F2805xF, F2805xM, F2806xF, and F2806xM Devices)

6.2 Motor Identification Process Overview

6.2.1 Controller (CTRL) State Machine

Table 6-1 summarizes all the states shown in Figure 6-4, with a brief description. A more detailed description is given in this document when the detailed motor identification process is explained.

Copyright © 2013–2019, Texas Instruments Incorporated

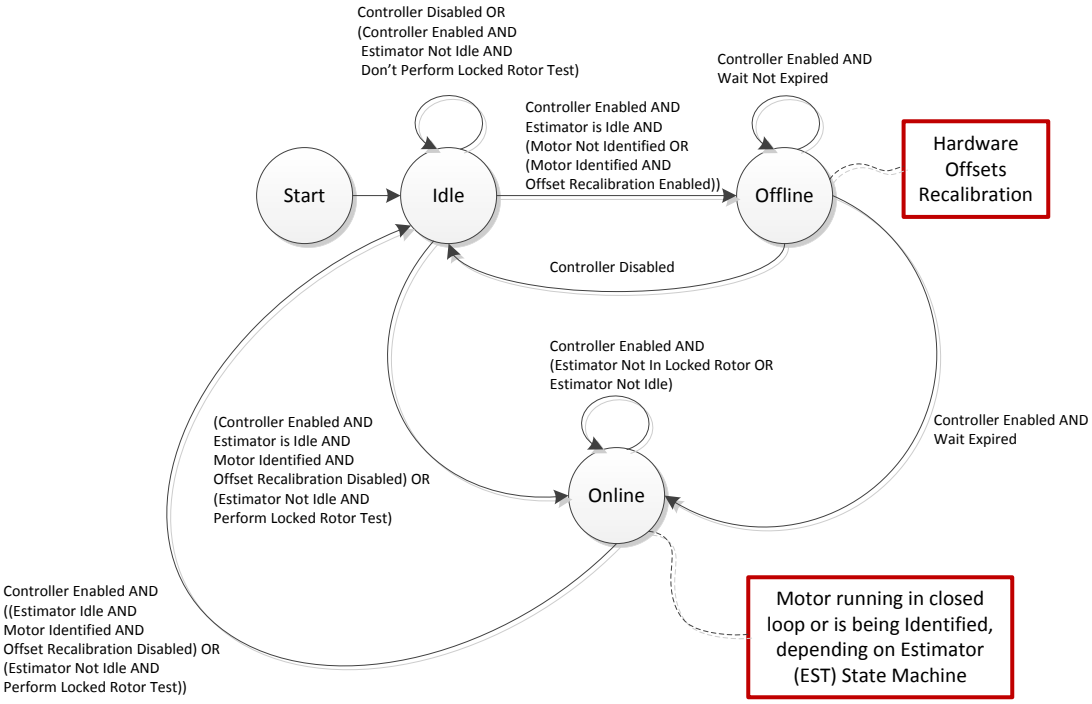


Figure 6-4. Controller (CTRL) State Diagram

Table 6-1. Controller (CTRL) States

Controller State	Brief Description
Start	The start state is only shown as a starting point of the entire state machine. However it does not actually exist in the controller state machine.
Idle **CTRL_State_Idle**	This state is present when the controller is waiting for user's input to start. This state is also present when the system is waiting for user's input to start doing the locked rotor test when identifying AC Induction Motors.
Offline **CTRL_State_OffLine**	The hardware offsets calibration is done during this state of the controller
Online **CTRL_State_OnLine**	Motor is running in closed loop, or is being identified. The entire estimator (EST) state machine is run when the controller (CTRL) state machine is Online

Table 6-2 summarizes all the state transition conditions shown in Figure 6-4.

Table 6-2. State Transitions for Controller (CTRL) State Diagram

Condition	Brief Explanation

Controller Disabled	This condition is present when the controller has not been enabled, can be checked with the following instruction:
	`if(CTRL_getFlag_enableCtrl(ctrlHandle)== FALSE)`
	Also, the controller can be disabled at any time with user's code by calling the following function with the indicated parameter:
	`CTRL_setFlag_enableCtrl(ctrlHandle,FALSE);`

Controller Enabled	This condition is present when the controller has been enabled, can be checked with the following instruction:
	`if(CTRL_getFlag_enableCtrl(ctrlHandle)== TRUE)`
	The controller can be enabled by using:
	`CTRL_setFlag_enableCtrl(ctrlHandle,TRUE);`

Estimator Not Idle	This condition is present when the estimator state is any state other than Idle. Can be checked with the following instruction: `if(EST_getState(obj->estHandle)!=EST_State_Idle)`

Copyright © 2013–2019, Texas Instruments Incorporated

Table 6-2. State Transitions for Controller (CTRL) State Diagram (continued)

Condition	Brief Explanation
Estimator Is Idle	This condition is present when the estimator state is in the Idle state. Can be checked with the following instruction: `if(EST_getState(obj->estHandle)==EST_State_Idle)`
Don't Perform Locked Rotor Test	This condition is internally checked in the state machine of the estimator, and is not publicly accessible with user's code. This condition is only checked internally when the motor type is set to induction motor.
Perform Locked Rotor Test	Same explanation as above for "Don't Perform Locked Rotor Test."
Motor Not Identified	This condition is true when the motor has not been identified, or when user motor's parameters haven't been loaded into the controller object. The Motor Not Identified condition can be checked by using the following example: `if(EST_isMotorIdentified(obj->estHandle)== FALSE)`
Motor Identified	After motor identification is completed, or after user motor's parameters are loaded into the controller, the motor identified condition is true. This condition can also be checked by using the following example: `if(EST_isMotorIdentified(obj->estHandle)==TRUE)`
Offset Recalibration Enabled	This condition is true when the hardware offsets recalibration has been enabled. Offsets recalibration is enabled by default. This can be checked with the following instruction example: `if(CTRL_getFlag_enableOffset(ctrlHandle)==TRUE)`
Offset Recalibration Disabled	When hardware offsets recalibration has been disabled, this condition is true. It can be checked using the following example: `if(CTRL_getFlag_enableOffset(ctrlHandle)==FALSE)`

Wait Not Expired	This is an internal condition which is checked while the offset recalibration is being executed. The time taken by the offset recalibration is defined by the following instruction in user.c file:
	`pUserParams->ctrlWaitTime[CTRL_State_OffLine] =`
	`(uint_least32_t)( 5.0 * USER_CTRL_FREQ_Hz);`
	Where the USER_CTRL_FREQ_Hz is defined in user.h.

Estimator Not In Locked Rotor	This condition is present when the estimator state machine is not in locked rotor test. Such condition can be checked by the user using the following instruction example: `if(EST_getState(obj->estHandle) !=EST_State_LockRotor);`
Wait Expired	The state transitions that have time dependencies are based on an internal counter compared to the value stored in the respective wait time in user.c: `uint_least32_t ctrlWaitTime[CTRL_numStates];`

Copyright © 2013–2019, Texas Instruments Incorporated

6.2.2 Estimator (EST) State Machine

Table 6-3 summarizes all the states shown in Figure 6-5, with a brief description. A more detailed description is given in this document when the detailed motor identification process is explained.

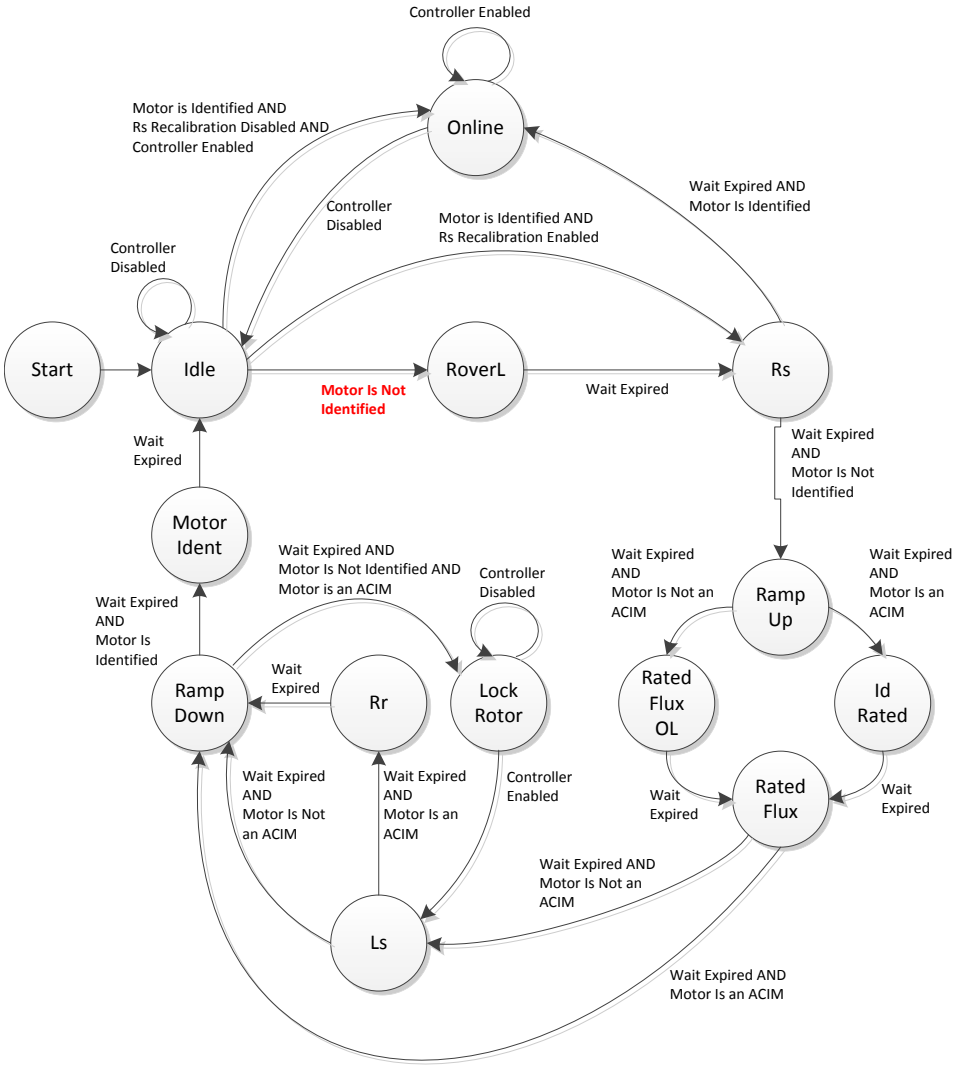


Figure 6-5. Estimator (EST) State Diagram

Table 6-3. Estimator (EST) States

Estimator State	Brief Description
Start	The start state is only shown as a starting point of the entire state machine. However it does not actually exist in the estimator state machine.
Idle **EST_State_Idle**	During the idle state, the estimator state machine does not execute any code. It is simply waiting for the controller state machine to change the state of the estimator.
RoverL **EST_State_RoverL**	The R over L state of the estimator is executed during the motor identification process in order to measure the electrical constant of the motor. The resulting R over L ratio is used at the end of this state to calculate the ID and IQ current controller gains.
Rs **EST_State_Rs**	The estimator is in the Rs state when identifying the stator resistance of the motor for the first time, or when it is recalibrating the stator resistance after the motor has been fully identified.
Online **EST_State_OnLine**	The online state of the estimator is present when the motor is operating in closed loop. In order to be in this state, the motor had to be fully identified or motor parameters had to be provided to a header file in user.h, and the controller had been run. This is the state where the speed reference can be changed, and when full load can be applied to the motor's shaft.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 6-3. Estimator (EST) States (continued)

Estimator State	Brief Description
Ramp Up **EST_State_RampUp**	The ramp up state of the estimator is to get the motor spinning up to a configured frequency in order to perform other identification tasks, such as flux and inductance identification. During this state, there aren't any parameters estimated, it only spins the motor up to a certain frequency.
Rated Flux OL **EST_State_RatedFluxOL**	This is a transitional state of the estimator prior to identifying the rated flux of the machine.
Rated Flux **EST_State_RatedFlux**	During this state, the flux linkage of the motor from the rotor to the stator is identified.
Id Rated **EST_State_IdRated**	Only applicable to ACIM motors, this state is present when the motor's magnetizing current is being identified for the commanded flux defined in user.h.
Ls **EST_State_Ls**	The stator inductance is identified during this state of the estimator.
Rr **EST_State_Rr**	Only applicable to ACIM motors, during this state, the rotor resistance is identified. The rotor must be locked in order to perform the rotor resistance identification.
Lock Rotor **EST_State_LockRotor**	This state is used to let users know that the rotor must be locked, and then controller must be re-enabled after rotor is locked, to proceed with the rest of the identification process. The controller is put in idle when the estimator is in lock rotor state.
Ramp Down **EST_State_RampDown**	After all parameters of the motor have been identified, the ramp down state is present to allow some time to remove the currents flowing through the motor windings. No parameters are identified during this state since it is only a transitional state before the end of the identification process.
Motor Ident **EST_State_MotorIdentified**	The motor identified state is also a transitional state to let the estimator state machine know that the motor is fully identified, and after this state is done, the state machine is put back in idle. The controller state machine is also put in idle state after the motor identified state is finished.

Table 6-4 summarizes all the state transition conditions shown in Figure 6-5.

Table 6-4. State Transitions for Estimator (EST) State Diagram

Condition	Brief Explanation
Motor Is Not Identified	Explained in the controller state machine
Motor Is Identified	Explained in the controller state machine

Rs Recalibration Disabled	The stator resistance recalibration feature of InstaSPIN, also known as Rs Offline recalibration, is used when the motor is at stand still, right before running the motor in closed loop when the motor has been fully identified, or when motor parameters have been provided by the user in user.h file. This condition where the Rs recalibration is disabled is present when this feature has been disabled, which can be checked using the following code example:
	`if(EST_getFlag_enableRsRecalc(obj->estHandle)==FALSE)`
	The Rs recalibration feature can be disabled by calling this function with the indicated parameter before enabling the controller:
	`EST_setFlag_enableRsRecalc(obj->estHandle, FALSE);`
	The Rs recalibration feature is enabled by default.

Rs Recalibration Enabled	This condition where the Rs recalibration is enabled is present when this feature has been enabled, which can be checked using the following code example:
	`if(EST_getFlag_enableRsRecalc(obj->estHandle)==TRUE)`
	The Rs recalibration feature can be enabled by calling this function with the indicated parameter before enabling the controller:
	`EST_setFlag_enableRsRecalc(obj->estHandle, TRUE);`
	The Rs recalibration feature is enabled by default.

Controller Disabled	Explained in the controller state machine
Controller Enabled	Explained in the controller state machine
Motor Is Not an ACIM	This condition is true when the motor type is not an ACIM, which can be checked by the following example: `if(CTRL_getMotorType(ctrlHandle)!=MOTOR_Type_Induction)`
Motor Is an ACIM	This condition is true when the motor type is an ACIM, which can be checked by the following example: `if(CTRL_getMotorType(ctrlHandle)==MOTOR_Type_Induction)`

Copyright © 2013–2019, Texas Instruments Incorporated

Table 6-4. State Transitions for Estimator (EST) State Diagram (continued)

Condition	Brief Explanation

Wait Expired	The state transitions that have time dependencies are based on an internal counter compared to the value stored in the respective wait time in userParams.h:
	`uint_least32_t estWaitTime[EST_numStates];`
	`uint_least32_t FluxWaitTime[EST_Flux_numStates];`
	`uint_least32_t LsWaitTime[EST_Ls_numStates];`
	`uint_least32_t RsWaitTime[EST_Rs_numStates];`

6.2.3 Controller (CTRL) and Estimator (EST) State Machine Dependencies

The controller state machine governs the estimator state machine. In fact, all of the estimator state transitions happen only when the controller state is Online. To illustrate this, consider the following simplified controller state machine (Figure 6-6), with a magnified online state showing the entire state machine of the estimator inside of it.

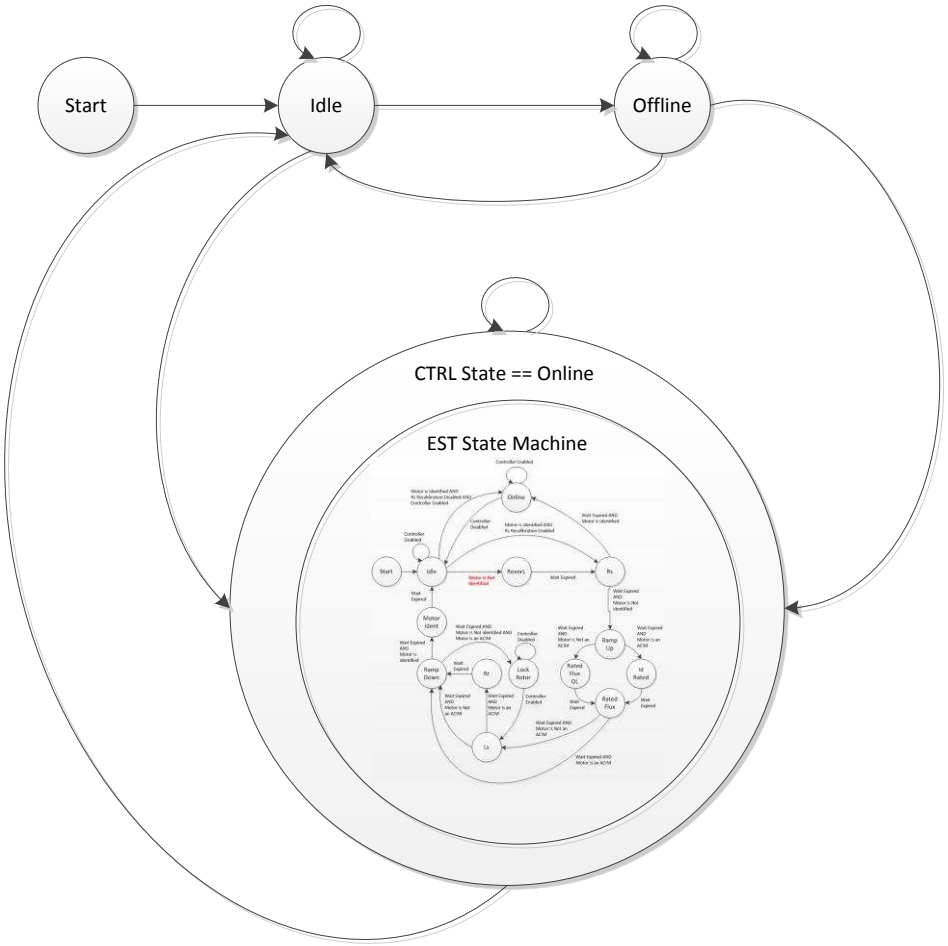


Figure 6-6. Controller and Estimator State Diagrams - Dependency Shown

Copyright © 2013–2019, Texas Instruments Incorporated

6.3 Differences between PMSM and ACIM Identification Process

There are certain states during the identification of the motors that differ between a PMSM and an ACIM motor. Figure 6-7 highlights the states that are related to PMSM motors, the ones related to ACIM motors, and also the states that are related to both.

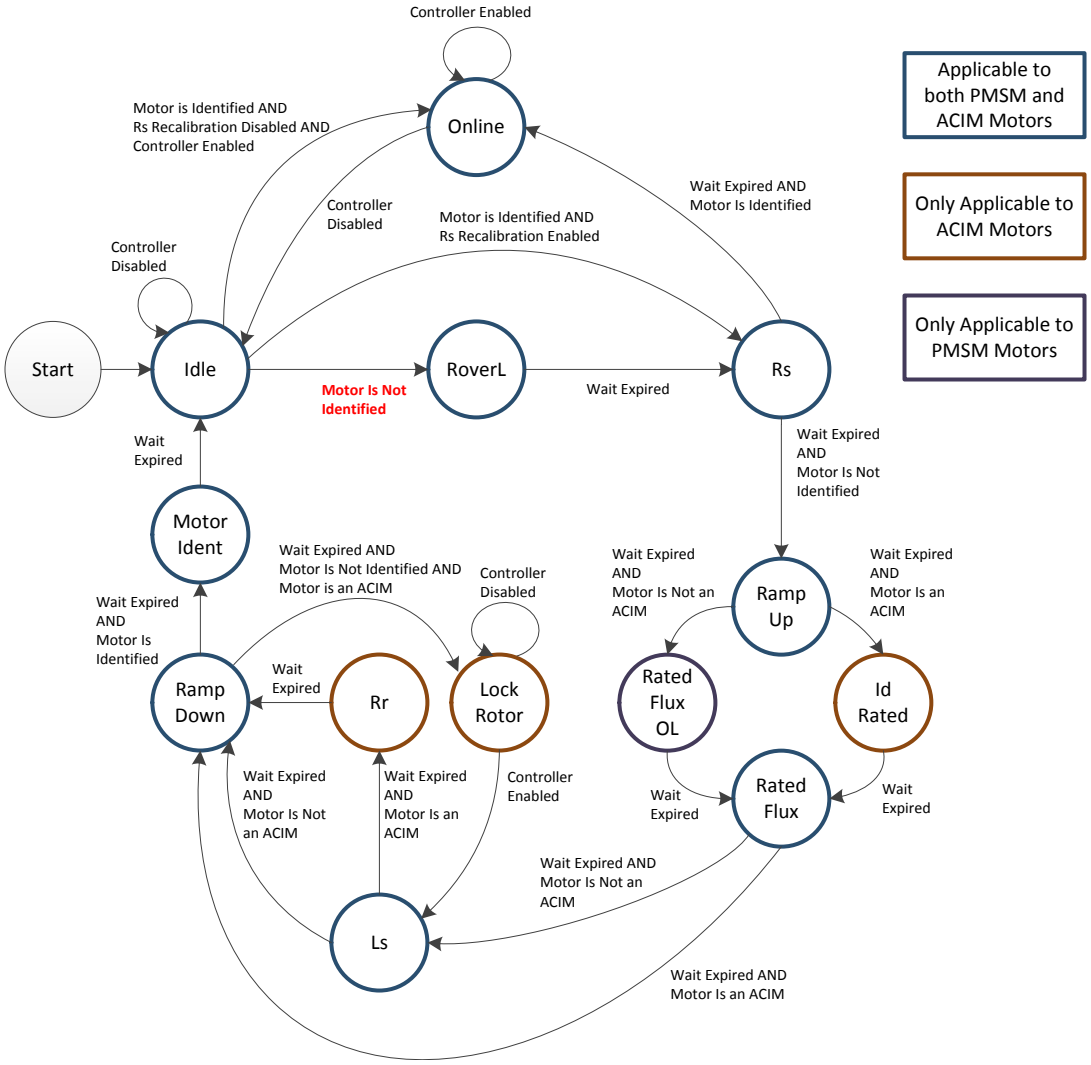


Figure 6-7. PMSM and ACIM States in EST State Diagram

As a supplement to Figure 6-7, consider Table 6-5, listing all the states, and the motors to which they apply.

Table 6-5. Listing of PMSM and ACIM EST States

Motors the State Applies to	Estimator States
Both PMSM and ACIM	Idle, Online, R over L, Rs, Ramp Up, Rated Flux, Ls, Ramp Down and Motor Identified
ACIM Only	Id Rated, Lock Rotor and Rr
PMSM Only	Rated Flux OL

Copyright © 2013–2019, Texas Instruments Incorporated

6.4 Prerequisites

There are several prerequisites in order to have a successful identification of a PMSM motor. The following sections describe these prerequisites grouped in three main areas: mechanical, hardware and software prerequisites.

6.4.1 Mechanical Prerequisites

6.4.1.1 Motor Connection

Before running motor identification, the motor must be connected to the driver board. This board can be a development board or a user's board, in either case the software needs to be configured to operate with the specific board.

6.4.1.2 Order of the Phases

The order of the phases to be connected to the board does not matter for controlling the dynamics of the motor, except direction. If direction is important, it is recommended to connect the motor for the required direction prior to starting identification. If the direction of the motor rotation is not the desired direction, swap two of the motor phases, and the direction will reverse. Then try the identification again.

6.4.1.3 Minimum Mechanical Load

It is also important to have as minimum mechanical load as possible. This is because the identification runs a series of motor tests, some of them in open loop. These open loop tests do not have the ability to produce maximum torque on the motor's shaft hence it is required to have the motor's shaft with as little load as possible, knowing that no load is ideal. Section 6.9.2.1 covers some guidelines to follow when identifying motors when the mechanical load cannot be removed, such as in a compressor or direct drive washing machine.

6.4.2 Hardware Prerequisites

For more details, see Chapter 5.

6.4.3 Software Prerequisites

For more details, see Chapter 5.

6.4.4 Software Configuration for PMSM Motor Identification

- Motor Type
- Number of Pole Pairs
- Frequency for Rhf and Lhf
- Current for Rs
- Current for Ls
- Frequency for Ls and Flux

6.4.4.1 Motor Type

User must know their motor type in order to run motor identification. For PMSM motor identification, set the following definition to PMSM motor type (MOTOR_Type_Pm) as shown below. This definition is in user.h:

```
#define USER_MOTOR_TYPE MOTOR_Type_Pm
```


Copyright © 2013–2019, Texas Instruments Incorporated

If the wrong motor is selected, for example a PMSM when an ACIM motor is connected, the estimator will not be able to identify the correct parameters. Motor identitication cannot identify what type of motor is connected, instead it identifies the motor parameters. Having the correct motor definition specified is required, otherwise motor identification will not work.

6.4.4.2 Number of Pole Pairs

The number of pole pairs is not critical but should be set correctly in order to have a correct measurement of speed in revolutions per minute (RPM) and torque. For example, consider a motor with four pole pairs. The following code example shows how this is set in user.h:

```
#define USER_MOTOR_NUM_POLE_PAIRS (4)
```

6.4.4.3 Frequency for Rhf and Lhf

This frequency is used by the motor identification algorithm to estimate initial values for stator resistance and stator inductance in order to calculate current controller gains. This estimation is done at this frequency, set by default to 100 Hz as shown in the following code example from user.h:

```
//! \brief Defines the R/L estimation frequency, Hz
//!
#define USER_R_OVER_L_EST_FREQ_Hz (100)
```

This frequency will be covered in more detail in the identification procedure discussed in upcoming sections of this document.

6.4.4.4 Current for Rs

The stator resistance is estimated by injecting a constant current, which is set by this parameter. Although this part of the identification process will be explained in detail, a general guideline to set this value is 10% to 20% of the rated phase current of the motor. For example, if a motor requires 4 A to produce rated torque, then the current to estimate the stator resistance (Rs) should be around 0.5 A.

```
#define USER_MOTOR_RES_EST_CURRENT (0.5)
```

Later in the motor identification process, this current is also used to get the motor spinning in open loop. If the motor does not spin through the entire ramp up process, increase in 10% of the rated current increments until the shaft is in motion during the entire ramp up process.

6.4.4.5 Current for Ls

As a general rule, the current commanded to estimate the stator inductance (Ls) should be 10% to 20% of the rated phase current (negative current). Considering the same example as the resistance estimation, if we consider the same motor, with 4 A of rated current, the recommended current would be about -0.5A, as shown below:

```
#define USER_MOTOR_IND_EST_CURRENT (-0.5)
```

Keep in mind that the current used to estimate Ls is independent of the current used for Rs identification. Even though the general guideline for both currents is 10-20% of the rated current of the motor, if the motor required more current for Rs in order to get it through the open loop ramp, the Ls current might not need to as high since the motor is already spinning.

Copyright © 2013–2019, Texas Instruments Incorporated

6.4.4.6 Frequency for Ls and Flux

This frequency is used to ramp up the motor being identified to estimate the stator inductance (Ls) and flux. Typically, for PMSM motors a frequency of 20 Hz is high enough to estimate a stator inductance from a few tens of µH and higher. If the inductance is known to be in the single digits of µH and lower, then a higher frequency is recommended, up to 60 Hz for very low inductances. As an example, consider the Anaheim motor from the DRV8312 kit (Revision D board):

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

If the typical value of 20 Hz exceeds the rated speed of the motor, set the frequency lower to stay less than the motor's rated speed.

6.4.5 Software Configuration for ACIM Motor Identification

- Motor Type
- Number of Poles
- Frequency for Rhf and Lhf
- Rated Flux
- Current for Rs
- Frequency for IdRated, Ls and Rr

6.4.5.1 Motor Type

As stated earlier, user must know their motor type in order to run motor identification. For ACIM motor identification, set the following definition to ACIM motor type (MOTOR_Type_Induction) as shown below. This definition is in user.h:

```
#define USER_MOTOR_TYPE MOTOR_Type_Induction
```

6.4.5.2 Number of Pole Pairs

The same criteria needs to be followed as explained in Section 6.4.4.2.

6.4.5.3 Frequency for Rhf and Lhf

The same criteria needs to be followed as explained in Section 6.4.4.3.

6.4.5.4 Rated Flux

The rated flux for an ACIM motor is required for a full identification. This rated flux is set in user.h as follows:

```
#define USER_MOTOR_RATED_FLUX (0.8165*220.0/60.0)
```

The way this flux is calculated is by the name plate of the motor. For example, if the motor is rated for single phase 220VAC and 60 Hz then this value is calculated as follows:

$$\text{RatedFlux} = \sqrt{2} \times \frac{1}{\sqrt{3}} \times \frac{220\text{VAC}}{60\text{Hz}} = 0.8165 \times \frac{220\text{VAC}}{60\text{Hz}} = 2.9938 \quad (9)$$

Another example is a motor with the same input voltage of 220 VAC but a rated frequency of 50 Hz. This would result in a rated flux of:

Copyright © 2013–2019, Texas Instruments Incorporated

$$\text{RatedFlux} = \sqrt{2} \times \frac{1}{\sqrt{3}} \times \frac{220\text{VAC}}{50\text{Hz}} = 0.8165 \times \frac{220\text{VAC}}{50\text{Hz}} = 3.5926\text{V / Hz} \quad (10)$$

The 2 sqrt 2 term is to convert single phase RMS voltage value to peak voltage, and the 1 3 1 over sqrt 3 term is to convert motor line-to-line voltage to motor line-to-neutral voltage. Based on the rated flux of a given motor, it is to identify the Id Rated current for half of the rated flux. In such a case, simply enter the desired flux or a portion of it. Notice that the voltage needed to calculate rated flux is in the motor line to motor neutral, hence the conversion from 220 VAC (motor line to motor line) to VDC (from motor line to motor line), and then from VDC (from motor line to motor line) to VDC (from motor line to motor neutral).

6.4.5.5 Current for Rs

The same criteria needs to be followed as explained in Section 6.4.4.4.

6.4.5.6 Frequency for IdRated, Ls and Rr

This frequency is used to ramp up the motor being identified, in order to estimate the Id Rated current, stator inductance and rotor resistance of the ACIM motor. A typical frequency used for induction motors is 5 Hz:

```
#define USER_MOTOR_FLUX_EST_FREQ_Hz (5.0)
```

6.5 Full Identification of PMSM Motors

When running full identification of PMSM motors, Figure 6-8 shows the sequence of events that happen inside of the controller and estimator state machines.

Copyright © 2013–2019, Texas Instruments Incorporated

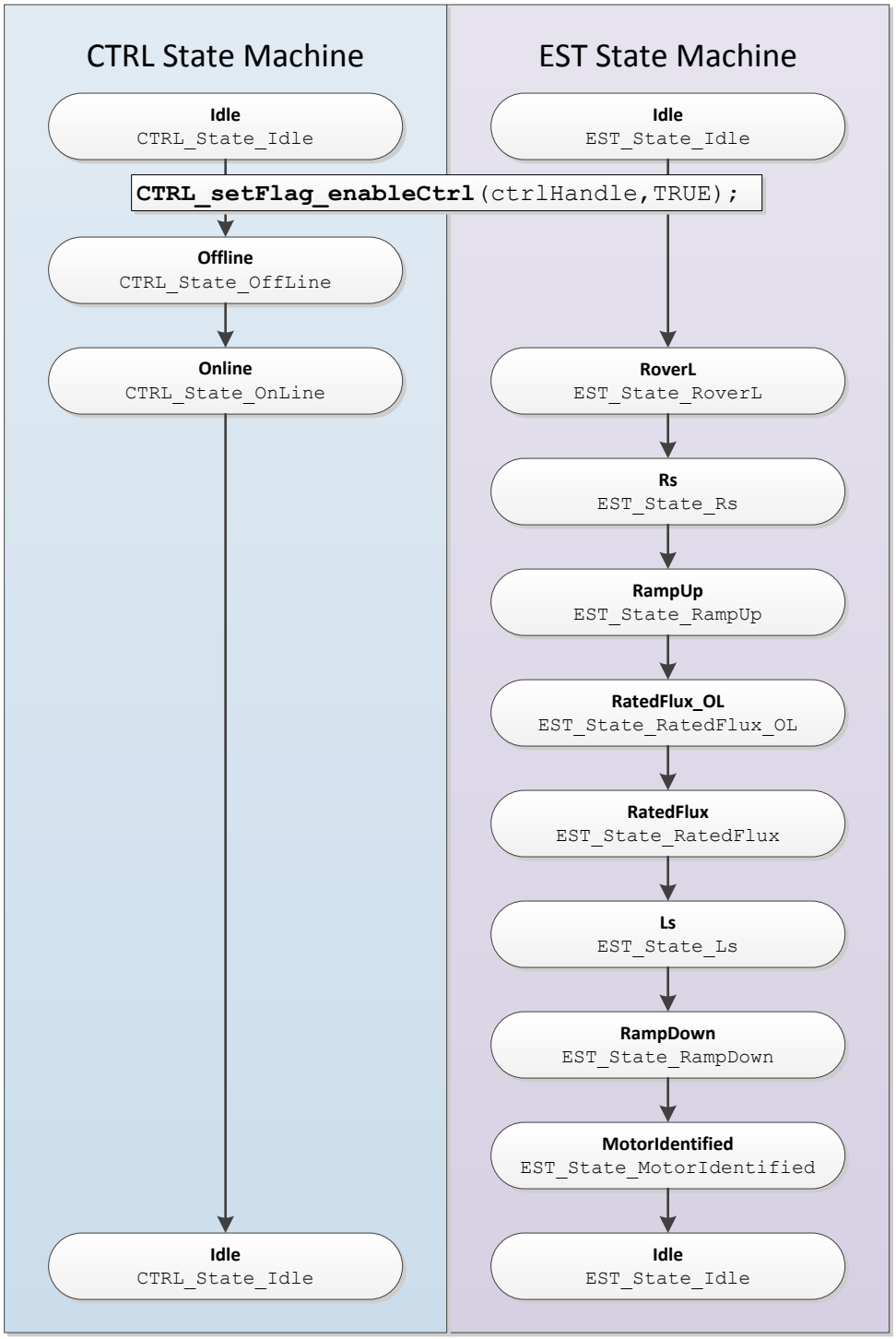


Figure 6-8. Full PMSM Identification - CTRL and EST Sequence of States

Prior to enabling the controller, the code knows that a full motor identification will be done when these two conditions are true:

1. motor has not been identified
2. no parameters are used from user.h.

Copyright © 2013–2019, Texas Instruments Incorporated

```
if( (EST_isMotorIdentified(obj->estHandle) == FALSE) &&
    (CTRL_getFlag_enableUserMotorParams(ctrlHandle) == FALSE))
```

In the next sections of this document, each state during the identification will be explained in detail.

6.5.1 CTRL_State_Idle and EST_State_Idle

Before the controller is enabled both the controller and estimator state machines are in the idle state, denoted by CTRL_State_Idle and EST_State_Idle. This is also known as the inactive state of both state machines.

6.5.2 CTRL_State_OffLine and EST_State_Idle (Hardware Offsets Calibrated)

As soon as the controller is enabled, and full identification starts, the first task performed by the controller state machine is the offset calculation. This is denoted by the state of the controller state machine named: CTRL_State_OffLine. The estimator state stays in the idle state (EST_State_Idle) during the controller offline state.

The offsets calculation is done in order to set the zeros for current measurements and voltage measurements. In order to calculate the offsets, a 50% duty cycle is set on the EPWM pins for a pre-configured period of time. The time in which these offsets are calculated can be changed by the user, and it is configured in user.c file as shown below:

```
pUserParams->ctrlWaitTime[CTRL_State_OffLine]=(uint_least32_t)(5.0*USER_CTRL_FREQ_Hz);
```

In the example above, the offsets calibration is done for a period of 5 seconds. Although 5 seconds for offset calibration is enough for most of the hardware, if the user requires a shorter or longer time for their particular needs, simply change the 5.0 value of the line of code above, and the time to do offset calibration will change according to the new setting.

Once the offset calibration is done, the final result will be stored in the driver object (HAL_Obj). For more details about HAL_Obj, see Table 8-1. Figure 6-9 shows the final results of calibrating the offsets for the DRV8312 Revision D board.

drv.adcBias	struct _DRV_AdcDat...	{...}
I	struct _MATH_vec3_	{...}
value	long[3]	0x0000BB46@Data
[0]	long	0.8795804977 (Q-Value(24))
[1]	long	0.8918017149 (Q-Value(24))
[2]	long	0.8876305819 (Q-Value(24))
V	struct _MATH_vec3_	{...}
value	long[3]	0x0000BB4C@Data
[0]	long	0.2504473329 (Q-Value(24))
[1]	long	0.2495527864 (Q-Value(24))
[2]	long	0.2473165989 (Q-Value(24))

Figure 6-9. CCStudio Watch Window after Offset Calibration

The current offsets, also known as bias values, ideally should be:

$$\text{bias} = 0.5 \times \text{Current_sf} \tag{11}$$

Note: for definitions of the variables used in the following equations, see Section 4.1.

This current scale factor (*Current_sf* or USER_CURRENT_SF) is calculated in the following example with values for the DRV8312 board revision D:

Copyright © 2013–2019, Texas Instruments Incorporated

$$USER_CURRENT_SF = \frac{USER_ADC_FULL_SCALE_CURRENT_A}{USER_IQ_FULL_SCALE_CURRENT_A} = \frac{17.69}{10.0} = 1.769$$

$$Current_Bias^{Ideal} = \frac{1.65}{3.3} \times USER_CURRENT_SF = 0.5 \times 1.769 = 0.8845 \quad (12)$$

The 0.5 value comes from the fact that the current feedback circuit is bidirectional, providing an ideal zero at $V_{DD}/2$, or 1.65V.

The voltage bias is calculated as follows. First, the voltage scale factor is done as shown here:

$$USER_VOLTAGE_SF = \frac{USER_ADC_FULL_SCALE_VOLTAGE_V}{USER_IQ_FULL_SCALE_VOLTAGE_V} = \frac{66.32}{48.0} = 1.3817$$

$$Voltage_Bias^{Ideal} = \frac{\frac{V_{BUS}}{2}}{USER_ADC_FULL_SCALE_VOLTAGE_V} \times USER_VOLTAGE_SF = 0.25 \quad (13)$$

The ideal voltage bias is based on the fact that when introducing a 50% duty cycle to measure these offsets, the phase voltage will present a voltage close to VBUS * 50%, and then this is scaled down depending on the maximum voltage measured by the ADC. Considering a DRV8312 revision D board with a VBUS of 24V, the ideal voltage bias results in 0.25, as shown below:

$$Voltage_Bias^{Ideal} = \frac{\frac{24}{2}}{66.32} \times 1.3817 = 0.25 \quad (14)$$

In the oscilloscope plot shown in Figure 6-10, the 50% duty cycle is shown, as well as the cursors measuring the 5 second period to do the offset calibration. On the left plot, no PWM can be seen due to the resolution of the horizontal scaling. On the left side, 1.65 V of amplitude is shown, which represents a 50% duty cycle of a 3.3 V signal. On the right side, the actual PWM signal is shown zoomed in to 50 µs per division.

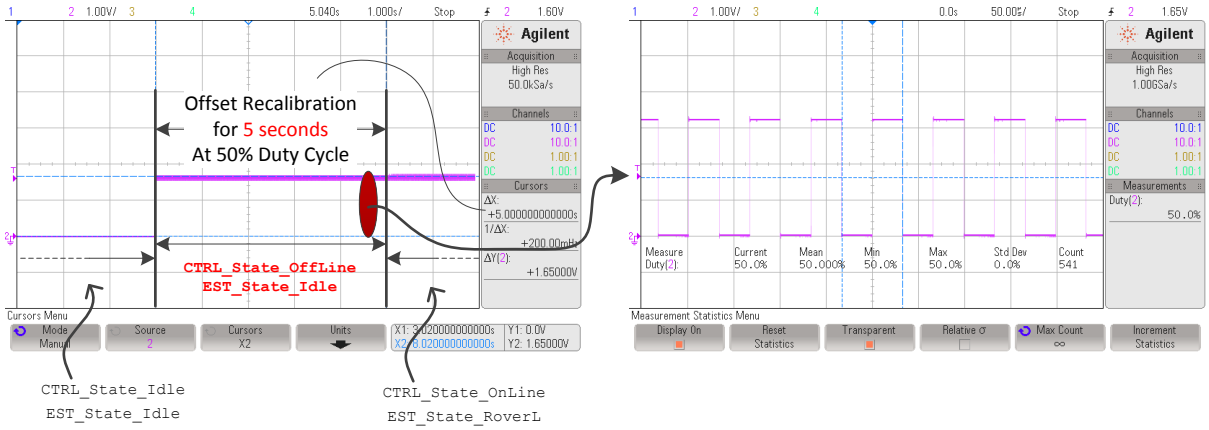


Figure 6-10. 50% PWM Duty for Offset Calculation

6.5.3 CTRL_State_OnLine and EST_State_RoverL

Once the offsets, also known as bias, are calibrated, the estimator is enabled and the following state of the identification process is started. The first state of the estimator state machine to be executed after being idle, is known as the R-over-L state, or RoverL (Figure 6-11).

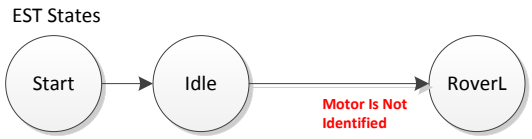


Figure 6-11. RoverL EST State

Copyright © 2013–2019, Texas Instruments Incorporated

This state of the estimator is used to measure the electrical time constant of the stator circuit by dividing the measured resistance and inductance. The RoverL time constant is used by the controller object in order to set the current controller gains, KP and KI of both IQ and ID current controllers. If the motor identification is bypassed, parameters provided in user.h are used to set the current controller gains.

The process of measuring the RoverL time constant is done by injecting a current of fixed amplitude, at a fixed frequency to the stator windings. Each of the injected current parameters is described below: amplitude, frequency and measurement time.

6.5.3.1 Amplitude of Injected Current for RoverL

In order to determine the amplitude of the current to be injected into the stator, the following parameter from user.h (USER_MOTOR_RES_EST_CURRENT) is divided by 2.

For example, the following parameter in user.h having a value of 1.0 for USER_MOTOR_RES_EST_CURRENT would inject ½ of the value, or 0.5 A. The ramp rate of this injected current is 0.5 seconds to reach the target current amplitude.

```
#define USER_MOTOR_RES_EST_CURRENT (1.0)
```

As a general guideline this current needs to be high enough to produce significant number of bits in the ADC measurement, but not too high that causes motor motion or motor overheating. This general rule results in a current of approximately 10% to 20% of the rated phase current of the motor.

6.5.3.2 Frequency of Injected Current for RoverL

The frequency of the current injection used to calculate RoverL is set by the following parameter in user.h, specifying the frequency in Hz.

```
//! \brief Defines the R/L estimation frequency, Hz
//!
#define USER_R_OVER_L_EST_FREQ_Hz (100)
```

For high speed motors, the default value of 100 Hz might cause the motor to spin or move back and forth. If that is the case for the motor under test, increase this frequency to a higher value using 50 Hz increments until the motor does not move at all during the R over L state.

6.5.3.3 Measurement Time for RoverL

The third parameter to consider when doing the R over L measurement is the time to do this measurement. This is configured in user.c as follows, and can be changed if needed, although the default setting would work for most of the cases:

```
pUserParams->estWaitTime[EST_State_RoverL] = (uint_least32_t)(5.0 * USER_EST_FREQ_Hz);
```

Figure 6-12 shows how this current is injected. The configured parameters are highlighter in red, 5 seconds duration, 1.0 A / 2 = 0.5 A of current amplitude, and 100 Hz.

Copyright © 2013–2019, Texas Instruments Incorporated

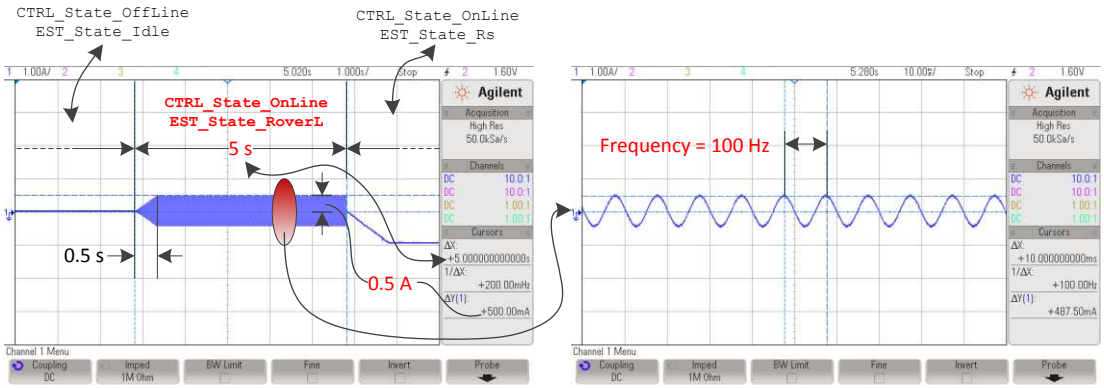


Figure 6-12. Injected Current for Measuring RoverL EST State

The resulting value of the R-over-L state can be read from the estimator by calling the following function, returning the RoverL ratio.

```
// Code example to get RoverL into a variable
float_t RoverL = CTRL_getRoverL(ctrlHandle);
```

Another method of checking the resulting estimation of the RoverL state is by calling two functions, one for the high frequency resistance estimation (Rhf) and one for the high frequency inductance estimation (Lhf). The following code example uses these two functions to have local copies of the resulting value during the RoverL state:

```
// Code example to get high frequency R (Rhf) and high frequency inductance
// (Lhf) to variables
float_t Rhf = CTRL_getRhf(ctrlHandle);
float_t Lhf = CTRL_getLhf(ctrlHandle);
float_t RoverL = Rhf/Lhf;
```

The resulting RoverL calculated in the above code example is identical to what the function CTRL_getRoverL() returns.

If motor identification is bypassed, users can use the following code example to calculate RoverL constant based on parameters provided in user.h file:

```
// Code example to get RoverL into a variable based on user.h parameters
#define USER_MOTOR_Rs (4.0)
#define USER_MOTOR_Ls_d (0.03)
float_t RoverL = USER_MOTOR_Rs/USER_MOTOR_Ls_d;
```


Copyright © 2013–2019, Texas Instruments Incorporated

Ultimately, the RoverL ratio is used by the controller object (CTRL_Obj) to initialize the current controller gains according to this ratio. Figure 6-13 shows how the current controller gains are internally set by the controller object (CTRL_Obj). The code listing shown here is for illustration purposes only, to show the math behind the initial setting of the current controllers. Since the code is implemented internally in the controller, user does not need to implement it.

```
// get the full scale current and voltage values from #defines in user.h
#define USER_IQ_FULL_SCALE_CURRENT_A (10.0)
#define USER_IQ_FULL_SCALE_VOLTAGE_V (48.0)

// deriving controller period in seconds from #defines in user.h
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK (1)
#define USER_PWM_FREQ_kHz      (15.0)
#define USER_PWM_PERIOD_usec   (1000.0/USER_PWM_FREQ_kHz)
#define USER_ISR_PERIOD_usec   USER_PWM_PERIOD_usec
#define USER_CTRL_PERIOD_usec  (USER_ISR_PERIOD_usec*USER_NUM_ISR_TICKS_PER_CTRL_TICK)
#define USER_CTRL_PERIOD_sec      ((float_t)USER_CTRL_PERIOD_usec/(float_t)1000000.0)

// get Lhf and RoverL from the controller object
float_t RoverL = CTRL_getRoverL(ctrlHandle);
float_t Lhf = CTRL_getLhf(ctrlHandle);

// get full scale current and voltage values in local variables
float_t fullScaleCurrent = USER_IQ_FULL_SCALE_CURRENT_A;
float_t fullScaleVoltage = USER_IQ_FULL_SCALE_VOLTAGE_V;

// get the controller period in seconds in a local variable
float_t ctrlPeriod_sec = USER_CTRL_PERIOD_sec;

// get the controller object handle
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;

// set the Id controller gains
Kp = _IQ((0.25*Lhf*fullScaleCurrent)/(ctrlPeriod_sec*fullScaleVoltage));
Ki = _IQ(RoverL*ctrlPeriod_sec);
Kd = _IQ(0.0);
PID_setGains(obj->pidHandle_Id,Kp,Ki,Kd);
PID_setUi(obj->pidHandle_Id,_IQ(0.0));
CTRL_setGains(ctrlHandle,CTRL_Type_PID_Id,Kp,Ki,Kd);

// set the Iq controller gains
Kp = _IQ((0.25*Lhf*fullScaleCurrent)/(ctrlPeriod_sec*fullScaleVoltage));
Ki = _IQ(RoverL*ctrlPeriod_sec);
Kd = _IQ(0.0);
PID_setGains(obj->pidHandle_Iq,Kp,Ki,Kd);
PID_setUi(obj->pidHandle_Iq,_IQ(0.0));
CTRL_setGains(ctrlHandle,CTRL_Type_PID_Iq,Kp,Ki,Kd);
```

Figure 6-13. Internal Code that Sets Kp and Ki Initial Gains for Current Controllers

6.5.3.4 Troubleshooting Current Controller Stability During RoverL Identification

See Section 6.9.

Copyright © 2013–2019, Texas Instruments Incorporated

6.5.3.5 Adjusting Resulting Current Controller Gains for High-Speeds

As can be seen, both ID and IQ current controllers are initialized with the same gains, calculated from RoverL and Lhf. The 0.25 factor is also introduced in the proportional gain of these two controllers. This factor is to set the proportional gain to ¼ of the theoretical limit. In applications where the motor needs to be run at much higher speeds compared to its rated speed, i.e. with field weakening, the proportional gains of both Id and Iq current controllers need to be scaled up to 4 times to get the gains to the theoretical limit. A simple way to scale these gains up to 4 is by using the following code example:

```
_iq Kp_Id = CTRL_getKp(handle,CTRL_Type_PID_Id);
_iq Kp_Iq = CTRL_getKp(handle,CTRL_Type_PID_Iq);
CTRL_setKp(handle,CTRL_Type_PID_Id, _IQmpy(Kp_Id, _IQ(4.0)));
CTRL_setKp(handle,CTRL_Type_PID_Iq, _IQmpy(Kp_Iq, _IQ(4.0)));
```

If the user would like to confirm that the current controller gains are set after the RoverL time constant has set the current controller, the following code example can be used:

```
// declare global variables for the Id controller gains
_iq gKp_Id, gKi_Id, gKd_Id;
// declare global variables for the Iq controller gains
_iq gKp_Iq, gKi_Iq, gKd_Iq;

// get the current controller gains for the Id controller
CTRL_getGains(ctrlHandle,CTRL_Type_PID_Id,&gKp_Id,&gKi_Id,&gKd_Id);
// get the current controller gains for the Iq controller
CTRL_getGains(ctrlHandle,CTRL_Type_PID_Iq,&gKp_Iq,&gKi_Iq,&gKd_Iq);
```

If the user chooses to bypass the gains set by the RoverL constant and decides to use their own gains, user simply needs to use the following functions to set the current controller gains, which are implemented in **ctrl.h**:

```
void CTRL_setKi(CTRL_Handle handle,const CTRL_Type_e ctrlType,const _iq Ki);
void CTRL_setKp(CTRL_Handle handle,const CTRL_Type_e ctrlType,const _iq Kp);
void CTRL_setGains(CTRL_Handle handle,const CTRL_Type_e ctrlType,
                   const _iq Kp,const _iq Ki,const _iq Kd);
```

6.5.4 CTRL_State_OnLine and EST_State_Rs

This state of the identification process performs the identification of the stator resistance (Figure 6-14).

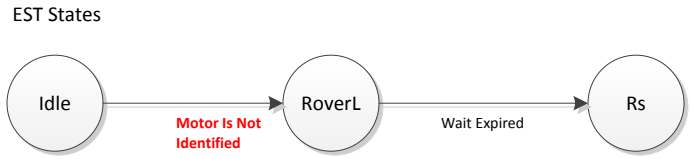


Figure 6-14. Rs EST State

A DC current is injected into the D-axis with the amplitude defined in user.h as follows:

```
#define USER_MOTOR_RES_EST_CURRENT (1.0)
```


Copyright © 2013–2019, Texas Instruments Incorporated

Note that this current is the same definition used for RoverL state, although RoverL uses half of this value, and the Rs state uses the full value in the definition. The injected current should be high enough to generate a significant measurement in the ADC converter, and at the same time low enough to avoid motor overheating. Typically, 10% to 20% of the rated current of the motor is enough to produce an accurate estimation of the stator resistance.

The time interval for this state is set by three time values in user.c, as follows:

Copyright © 2013–2019, Texas Instruments Incorporated

```
pUserParams->RsWaitTime[EST_Rs_State_RampUp] = (uint_least32_t)(1.0*USER_EST_FREQ_Hz);
pUserParams->RsWaitTime[EST_Rs_State_Coarse] = (uint_least32_t)(2.0*USER_EST_FREQ_Hz);
pUserParams->RsWaitTime[EST_Rs_State_Fine] = (uint_least32_t)(4.0*USER_EST_FREQ_Hz);
```

By default, the entire process of identifying the stator resistance, Rs, takes 7 seconds. The first part of the Rs identification process is a ramp-up time of 1 second. During this time, the defined DC current is injected into the D-axis. Once the ramp-up time is expired, the Rs identification process starts with a coarse tuning of the identified Rs. The coarse process takes the time defined previously by the time stored in RsWaitTime [*EST_Rs_State_Coarse*]. By default this time is set to 2 seconds, and it is known to be enough time to do a coarse calibration of all the motors tested for the InstaSPIN library release. However the time setup is flexible so users can tune if required, although tuning is not foreseen to be required.

Once the coarse process has finished, the fine Rs recalibration starts. The time taken by the identification process to complete the fine Rs recalibration is set by default to 4 seconds, and again the user has flexibility to change this by modifying the value stored in RsWaitTime [*EST_Rs_State_Fine*].

Figure 6-15 shows the entire Rs identification process, highlighting ramp times, amplitudes and duration of the process.

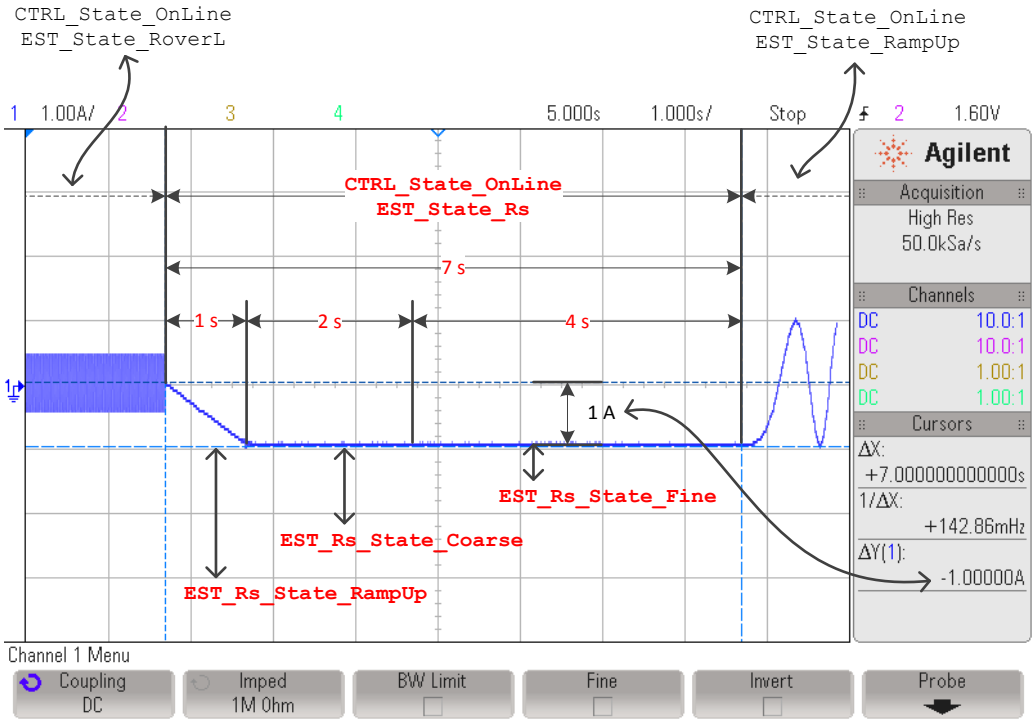


Figure 6-15. Phase Current During Rs Identification EST State

The user can monitor how the resistance is being estimated using the following code example. This is useful especially when tuning the amount of time spent identifying the resistance.

```
// get the stator resistance
gMotorVars.Rs_Ohm = EST_getRs_Ohm(obj->estHandle);
```

For example, monitoring the resistance value while it is being identified gives the user feedback on the amount of time required for the identified resistance to be stable. The time for the identified resistance to be stable can be configured in file user.c so next time the motor is identified the process is faster.

Copyright © 2013–2019, Texas Instruments Incorporated

6.5.4.1 Troubleshooting Current Controller Stability During Rs Identification

See Section 6.9.

6.5.5 CTRL_State_OnLine and EST_State_RampUp

After the stator resistance is done, a new estimator state is executed. This next state is called the Ramp-Up state, or EST_State_RampUp. During this state of the identification process, the motor is accelerated to a certain speed to start identification of other parameters. During this state, there is no identification of any parameters, but the conditions are started. Several factors influence this state.

6.5.5.1 Ramp-Up Current Amplitude

The first one is the amplitude of the currents used for the ramp-up process. This current is again the amplitude used for the stator resistance identification. Figure 6-16 shows a definition of this current, and in this case, 1 A is used for the ramp:

```
#define USER_MOTOR_RES_EST_CURRENT (1.0)
```

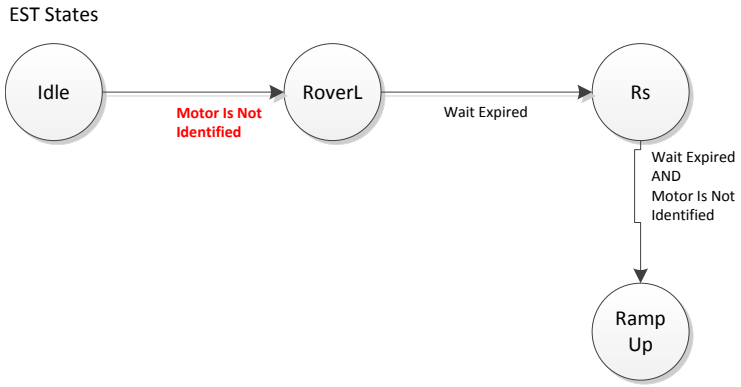


Figure 6-16. Ramp-Up EST State

6.5.5.1.1 Troubleshooting Motor Shaft Stopping During Ramp-Up

See Section 6.9.

6.5.5.2 Ramp-Up Time and Acceleration

The next parameter during the ramp-up state is the period of time while the motor is ramped up (Figure 6-17). This is set by default to 20 seconds as shown in the next code example from user.c file:

```
pUserParams->estWaitTime[EST_State_RampUp] = (uint_least32_t)(20.0*USER_EST_FREQ_Hz);
```


Copyright © 2013–2019, Texas Instruments Incorporated

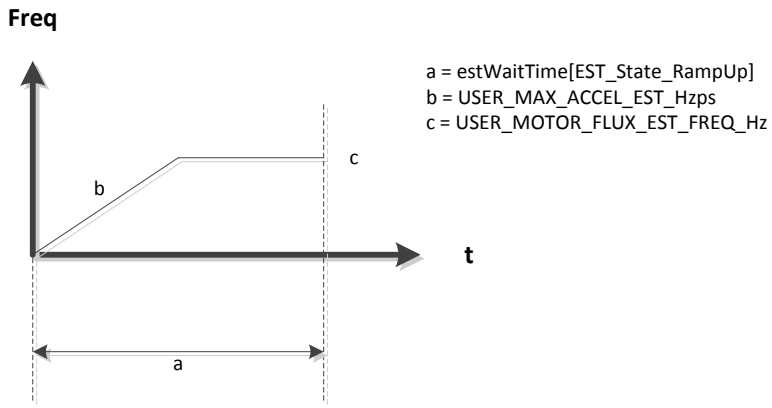


Figure 6-17. Ramp-Up Timing

This time can be changed to any desired value. Figure 6-18, taken using a current probe to measure phase current, shows the ramp up state time.

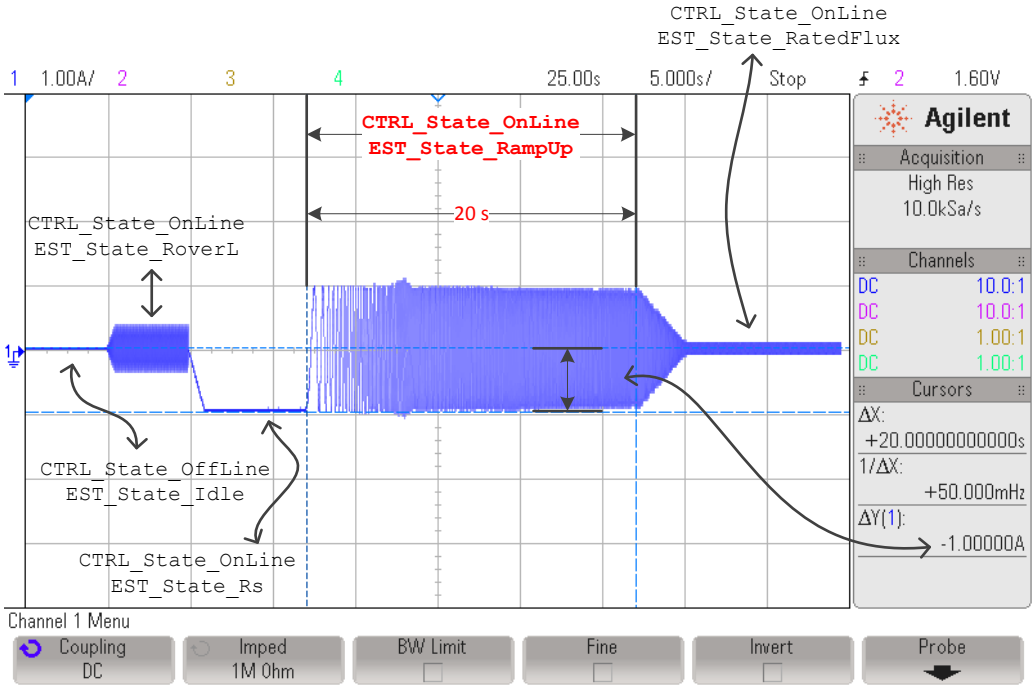


Figure 6-18. Phase Current During RampUp EST State

The acceleration of this ramp is another parameter set in user.c, which can be changed according to user requirements. In the same example of a high inertia load, this acceleration can be changed. The default value of the ramp-up acceleration is set to 2.0 Hz/s as shown below:

```
//! \brief Defines maximum acceleration for the estimation speed profiles, Hz/s
#define USER_MAX_ACCEL_EST_Hzps (2.0)
```

6.5.5.2.1 Troubleshooting Motor Shaft for Smoother Ramp

See Section 6.9.

Copyright © 2013–2019, Texas Instruments Incorporated

6.5.5.3 Ramp-Up Final Speed for PMSM

The final speed after the ramp-up is set in user.h as part of the motor parameters. This speed, specified in Hz, should be set depending on the phase inductance range. For single digit µH inductances this value should be around 50 Hz. For tens to hundreds of µH inductances, a value of 20 Hz should be enough to allow an accurate identification of the inductance.

```
// During Motor ID, maximum commanded speed in Hz
#define USER_MOTOR_FLUX_EST_FREQ_Hz (20.0)
```

Keep in mind that increasing this frequency might require increasing the ramp up time, so that the ramp up state is long enough to allow reaching the final frequency with the specified acceleration. The previous plot also shows the final frequency of 20 Hz.

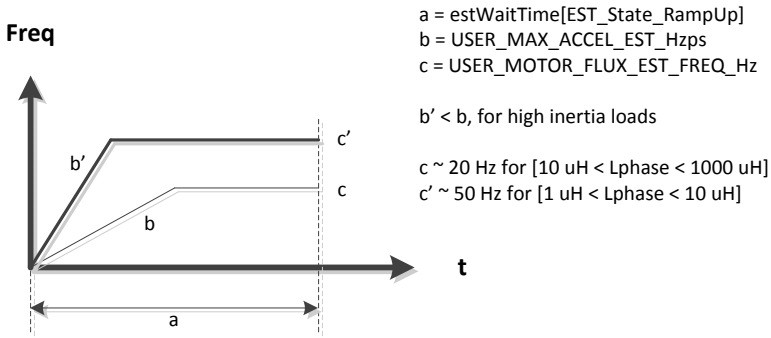


Figure 6-19. RampUp Timing with change in Acceleration and Final Speed

6.5.6 CTRL_State_OnLine and EST_State_RatedFlux

Once the motor is running at a commanded frequency set in user.h, the rated flux identification process starts (Figure 6-20).

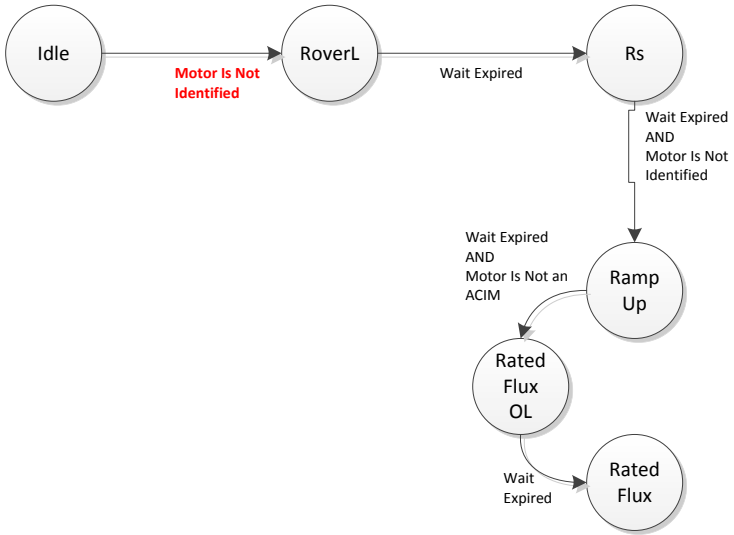


Figure 6-20. PMSM Rated Flux EST State

Copyright © 2013–2019, Texas Instruments Incorporated

6.5.6.1 Current Ramp-Down

The first thing that happens when identifying the flux is for a closed-loop to be enabled internally by the motor identification state machine. This closed-loop is not enabled by the user. Current consumption lowers to a minimum current value needed to keep the mechanical load spinning at the same frequency when this closed-loop is enabled. The slope at which current is lowered is a fixed value, Rs estimation per second divided by 3. The dividing factor of 3 was selected during design of the motor identification process to provide a slower slope.

In order to calculate this slope, users can use the following equation. In this example, 1 A was used for resistance identification:

$$\text{RatedFlux_CurrentSlope} = \frac{\text{USER_MOTOR_RES_EST_CURRENT}}{1\text{s}} \times \frac{1}{3} = 0.33\,\text{A/s} \tag{15}$$

The 0.33 A/s slope can be seen in Figure 6-21, showing how current is reduced as soon as the Rated Flux state is present. Also in the same plot, the time it takes to identify the rated flux is also highlighted.

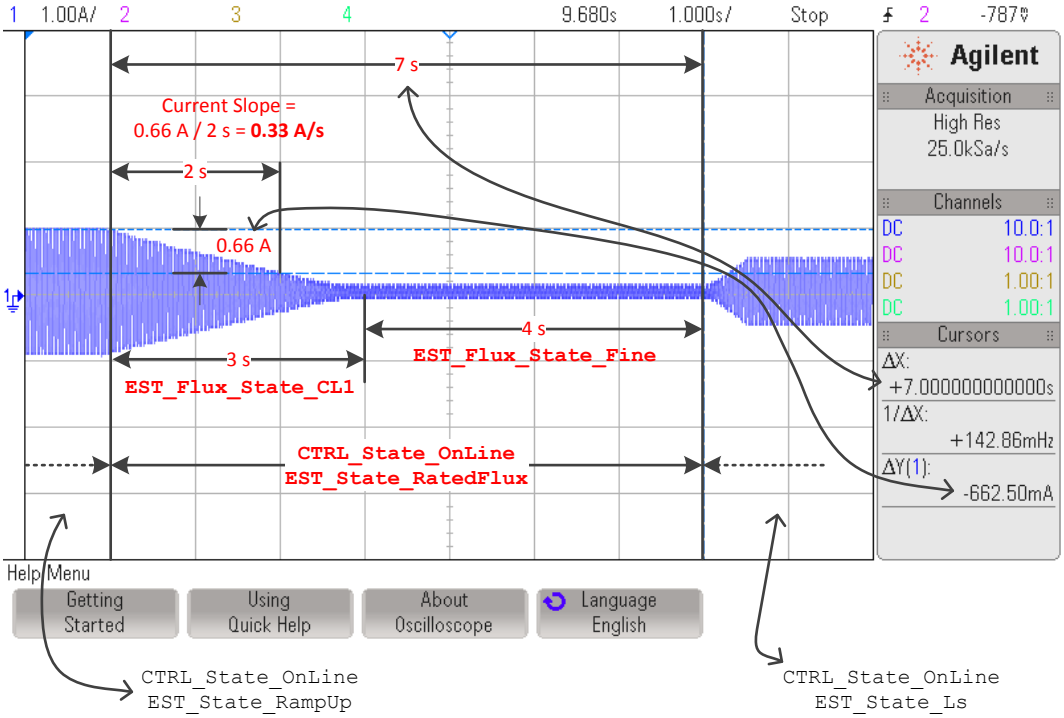


Figure 6-21. Phase Current During Rated Flux EST State

6.5.6.2 Total Measurement Time

The time shown in this plot is based on the default setting of 3 seconds for the current ramp-down (EST_Flux_State_CL1, CL1 stands for Closed Loop 1) and 4 seconds for the fine tuning of the rated flux (EST_Flux_State_Fine), for a total of 7 seconds. Both of these values are set in by function calls in file user.c, as shown below:

```
pUserParams->FluxWaitTime[EST_Flux_State_CL1] =(uint_least32_t)(3.0*USER_EST_FREQ_Hz);
pUserParams->FluxWaitTime[EST_Flux_State_Fine]=(uint_least32_t)(4.0*USER_EST_FREQ_Hz);
```

These default values are known to work for all the motors tested during the validation of the algorithm. Users can confirm that the time to identify is enough by monitoring the rated flux identified by the algorithm, and make sure that the identified value is stable while the state of the estimator is in EST_State_RatedFlux.

Copyright © 2013–2019, Texas Instruments Incorporated

The following code example shows how to monitor the value of the identified flux, and if the value does not vary more than a typical variation of about 5% in the watch window, the identified flux can be considered to be stable:

```
// get the flux
gMotorVars.Flux_VpHz = EST_getFlux_VpHz(obj->estHandle);
```

6.5.6.3 Troubleshooting Flux Measurement

See Section 6.9.

6.5.7 CTRL_State_OnLine and EST_State_Ls

Once the rated flux is measured the stator inductance identification process starts (Figure 6-22).

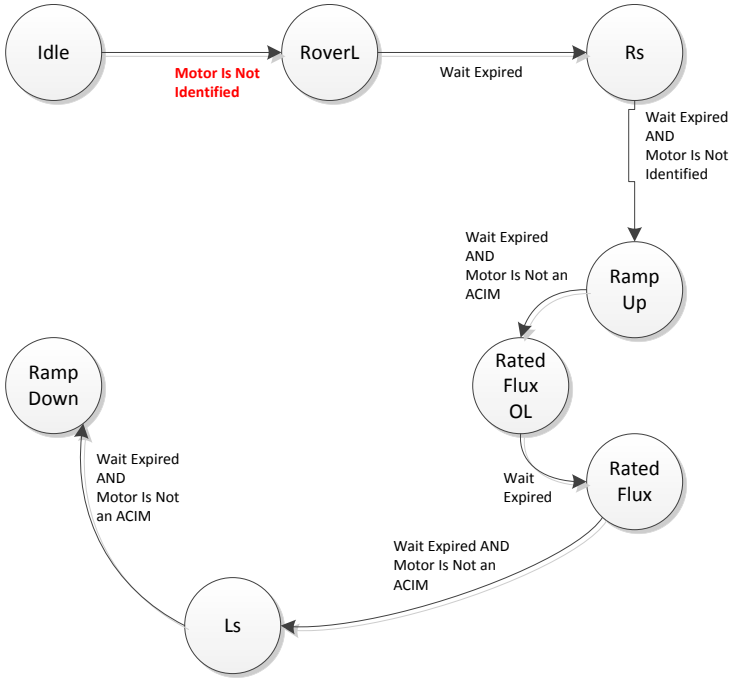


Figure 6-22. Stator Inductance EST State

In order to identify the stator inductance of the PMSM motor, the algorithm injects a current into the D-axis, also known as ID. The current must be negative, and it is set in user.h. As a general rule, this current should be between 10% and 20% of the rated current of the motor, negative in sign. The following value is set for a 4 A motor, hence the current used to identify the stator inductance was set to -0.5 A as shown:

```
#define USER_MOTOR_IND_EST_CURRENT (-0.5)
```

And the time spent for stator inductance identification is configured in user.c as follows:

```
pUserParams->LsWaitTime[EST_Ls_State_Init] = (uint_least32_t)( 3.0*USER_EST_FREQ_Hz);
pUserParams->LsWaitTime[EST_Ls_State_Fine] = (uint_least32_t)(30.0*USER_EST_FREQ_Hz);
```


Copyright © 2013–2019, Texas Instruments Incorporated

Figure 6-23 shows the time it takes to run the inductance identification state. It also shows the current amplitude injected. Although we set the current to be -0.5 A this current is injected into the D-axis, so it will be noticed in the phase current waveform as an amplitude of 0.5 A plus the current needed to keep the load moving. Since it is required to move all the mechanical load from the shaft, the current amplitude will be close to 0.5 A as shown.

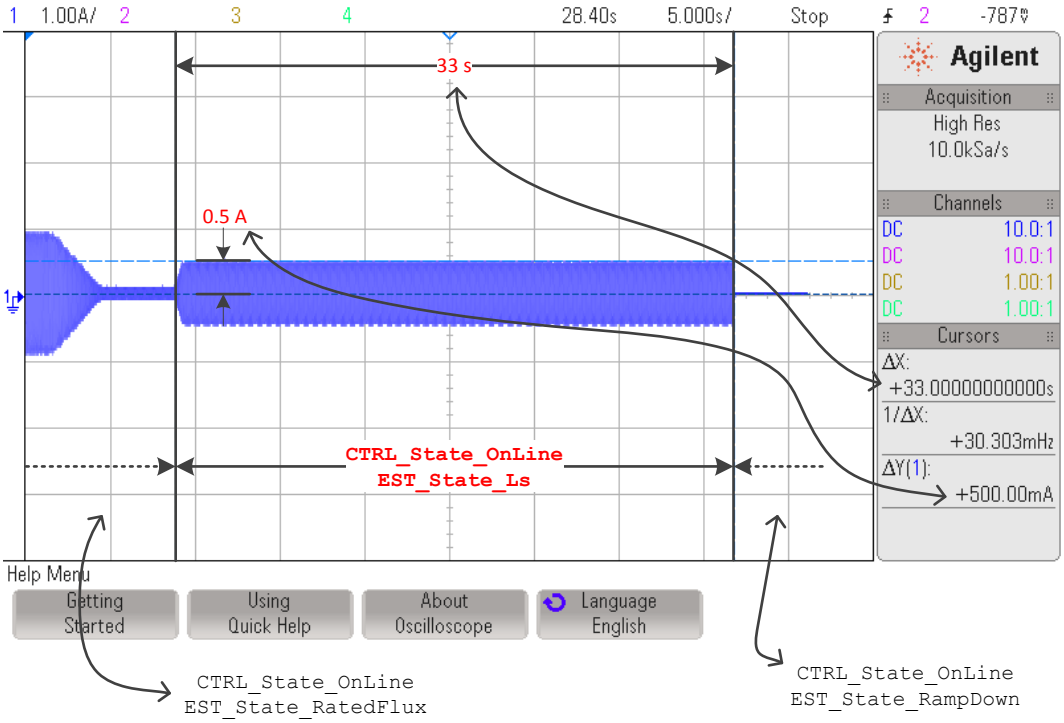


Figure 6-23. Injected Current for Ls Identification

6.5.7.1 Ramp-Up Current

The initial slope, when the current builds up to the specified current, is set by the current used for the resistance estimation for every second. For example, consider a resistance estimation current of 1 A configured as follows:

```
#define USER_MOTOR_RES_EST_CURRENT (1.0)
```

And -0.5 A for the inductance estimation configured as follows:

```
#define USER_MOTOR_IND_EST_CURRENT (-0.5)
```

With these two configurations, it will take a total of 0.5 s to build up 0.5 A of current into the D-axis, as shown in Figure 6-24.

Copyright © 2013–2019, Texas Instruments Incorporated

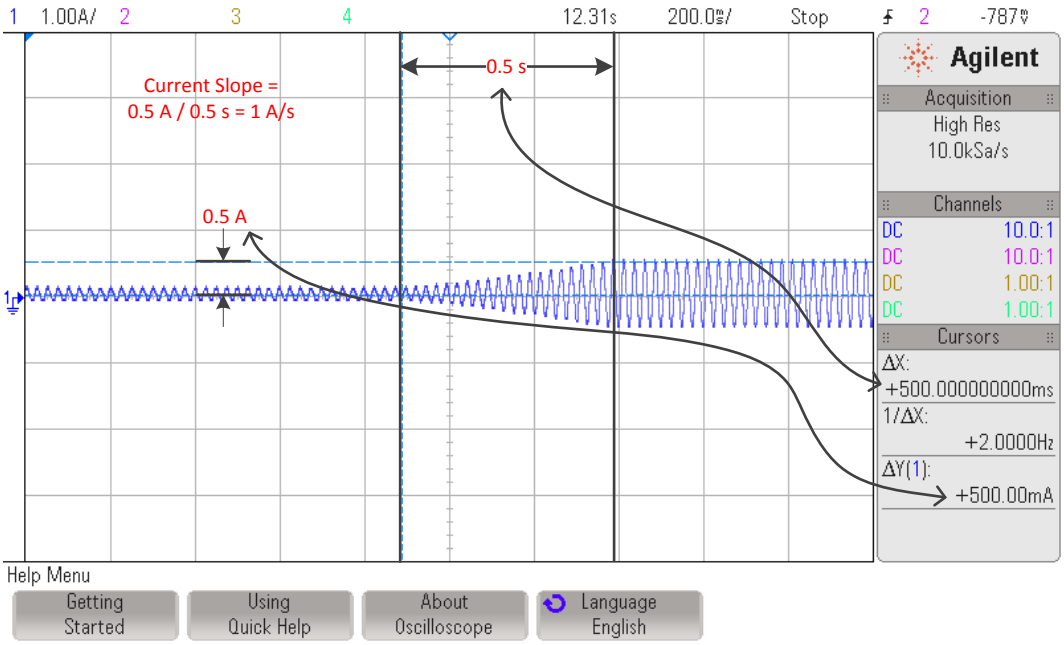


Figure 6-24. Current Ramp for Ls Identification

6.5.7.2 Troubleshooting Ls Identification

See Section 6.9.

6.5.7.3 Ls_d and Ls_q, Direct and Quadrature Stator Inductance

Keep in mind that the estimated inductance will be stored in both Ls_d and Ls_q, even if these values are different for a given motor. In other words, this version of InstaSPIN does not identify Ls_d and Ls_q separately, but it identifies an average Ls, which is then stored into Ls_d and Ls_q with the same value. In the future, when InstaSPIN identified Ls_d and Ls_q individually, then both functions will return a different value. If motor parameters are set in user.h with different Ls_d and Ls_q, and motor identification is bypassed, then the current version of InstaSPIN will also return different values when calling both functions EST_getLs_d_H and EST_getLs_q_H.

6.5.8 CTRL_State_OnLine and EST_State_RampDown

This state does not perform any particular action as far as the estimation process goes. It can be considered as a transition period of the state machine. Although there is a time associated with this state, as shown below, changing this time does not affect the identified variables.

```
pUserParams->estWaitTime[EST_State_RampDown] = (uint_least32_t)(2.0*USER_EST_FREQ_Hz);
```


Copyright © 2013–2019, Texas Instruments Incorporated

6.5.9 CTRL_State_OnLine and EST_State_MotorIdentified

The final state of the identification process is also a transitional state to let the internal state machine know that the motor has been identified. After the transitional state of EST_State_MotorIdentified is over, both state machines, CTRL and EST state machines, are put back to Idle. Users can check if the motor has been identified by calling the following function. If this function returns a TRUE, it means that the motor has been identified, either by going through all the described states, or by using motor parameters from a header file:

```
gMotorVars.Flag_MotorIdentified = EST_isMotorIdentified(obj->estHandle);
```

Once the motor has been fully identified, if users require running the motor identification process again, the controller must be re-initialized to set the state machines to an initial state with the motor identified flag back to FALSE. The following function call re-initializes the controller back to the initial state and the motor identified flag back to FALSE:

```
// set the default controller parameters
CTRL_setParams(ctrlHandle,&gUserParams);
```

To summarize the complete state machine of the identification process, see Figure 6-25.

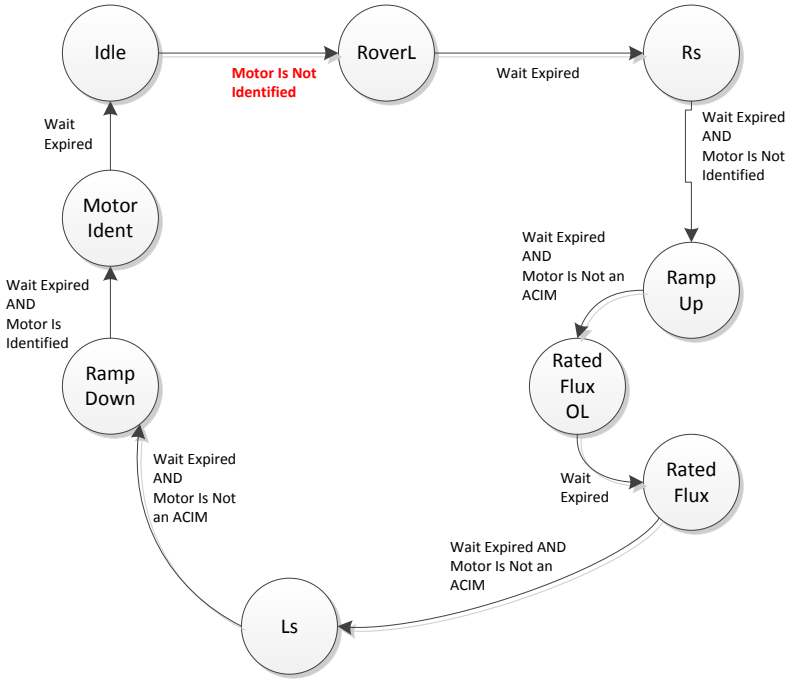


Figure 6-25. Complete PMSM Motor ID Process in EST State Diagram

The entire process of PMSM motor identification is also shown in Figure 6-26, where one phase current is plotted.

Copyright © 2013–2019, Texas Instruments Incorporated

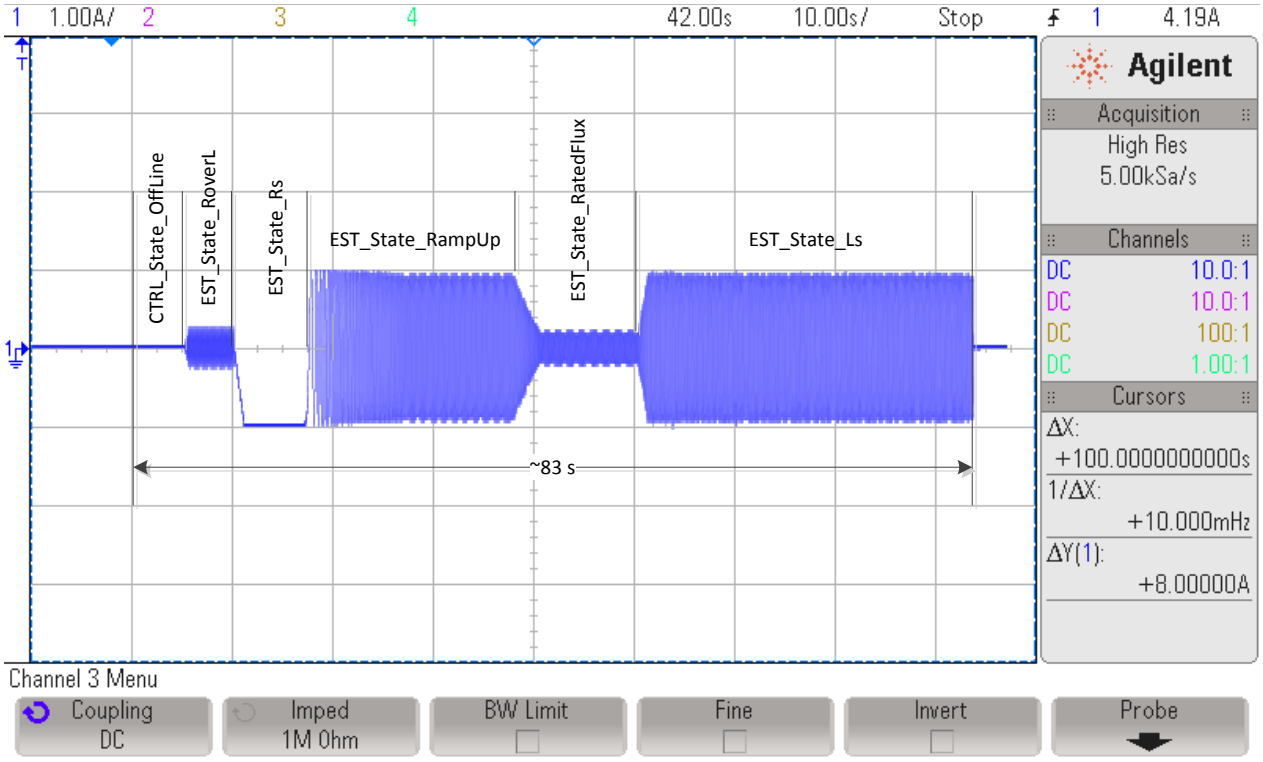


Figure 6-26. Phase Current Measurement of Entire PMSM Motor ID Process

6.5.10 CTRL_State_Idle and EST_State_Idle

After the motor is fully identified, both state machines are set to Idle.

6.6 Full Identification of ACIM Motors

When running full identification of ACIM motors, Figure 6-27 shows the sequence of events that happen inside of the controller and estimator state machines.

Copyright © 2013–2019, Texas Instruments Incorporated

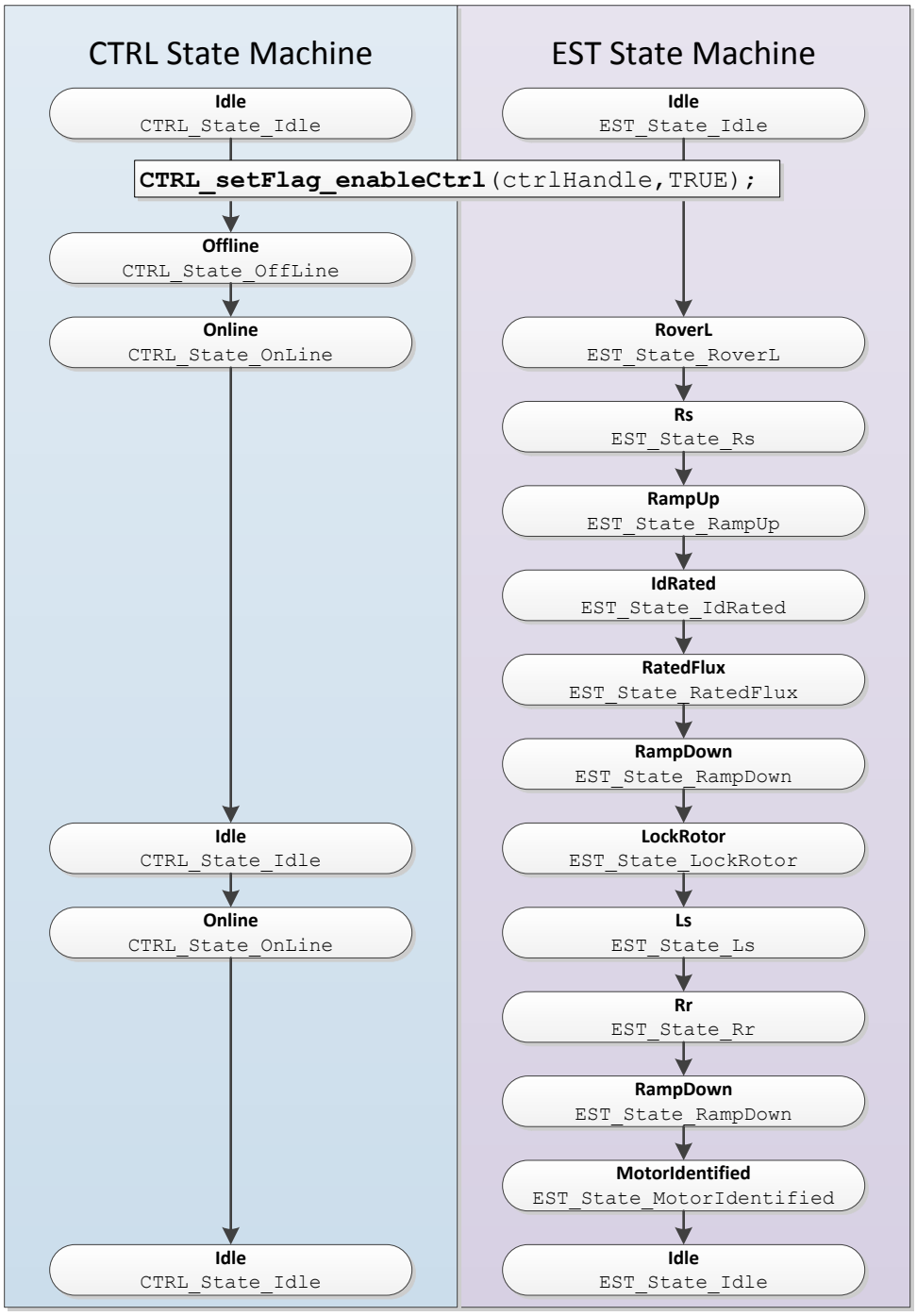


Figure 6-27. Full ACIM Identification - CTRL and EST Sequence of States

Prior to enabling the controller, the code knows that a full motor identification will be done when these two conditions are true:

1. motor has not been identified
2. no parameters are used from user.h.

```
if( (EST_isMotorIdentified(obj->estHandle) == FALSE) &&
    (CTRL_getFlag_enableUserMotorParams(ctrlHandle) == FALSE))
```


Copyright © 2013–2019, Texas Instruments Incorporated

In the next sections of this document, each state during the identification will be explained in detail.

6.6.1 CTRL_State_Idle and EST_State_Idle

The idle state of the controller and the estimator state machines are the same for both motors. For more information about this state, see Section 6.5.1.

6.6.2 CTRL_State_OffLine and EST_State_Idle

When the controller is in the Offline state, offsets are calibrated. For more information about this state, see Section 6.5.2.

6.6.3 CTRL_State_OnLine and EST_State_RoverL

In order to calculate the current controller gains, the same process and operations are done as in the PMSM motor. For more information about this state, see Section 6.5.3.

6.6.4 CTRL_State_OnLine and EST_State_Rs

When the estimator is in Rs state, the stator resistance is calibrated. For more information about this state, see Section 6.5.4.

6.6.5 CTRL_State_OnLine and EST_State_RampUp

After the stator resistance is done, a new estimator state is executed. This next state is called the Ramp-Up state, or EST_State_RampUp (Figure 6-28). During this state of the identification process, the motor is accelerated to a certain speed to start identification of other parameters. During this state, there is no identification of any parameters, but the conditions are started. Several factors influence this state.

In terms of functionality, for a detailed description of what happens during the EST_State_RampUp state, see Section 6.5.5.

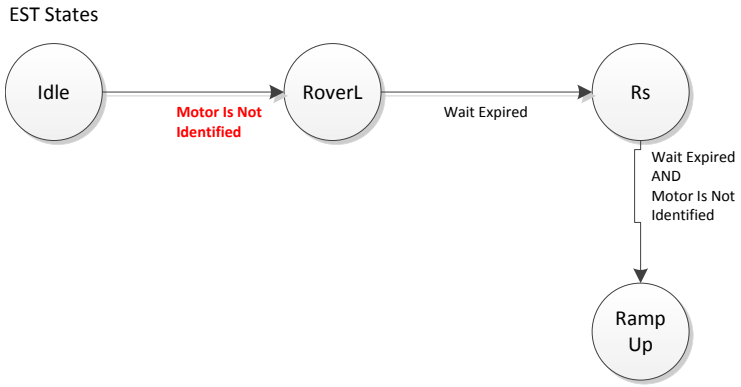


Figure 6-28. Ramp-Up EST State

6.6.5.1 Ramp-Up Final Speed for ACIM

The only difference between the PMSM and ACIM motor identification process in this state is that for ACIM motors a typical frequency of 5 Hz is used, as opposed to a typical of 20 Hz used for PMSM motors.

```
// During Motor ID, maximum commanded speed in Hz
#define USER_MOTOR_FLUX_EST_FREQ_Hz (5.0)
```


Copyright © 2013–2019, Texas Instruments Incorporated

The following oscilloscope plot shows a zoomed in plot of a phase current, where the frequency is ramped up to 5 Hz. Note that the default ramp-up acceleration of 2.0 Hz/s, shown previously for PMSM in Section 6.5.5, is used for the ACIM example shown in Figure 6-7.

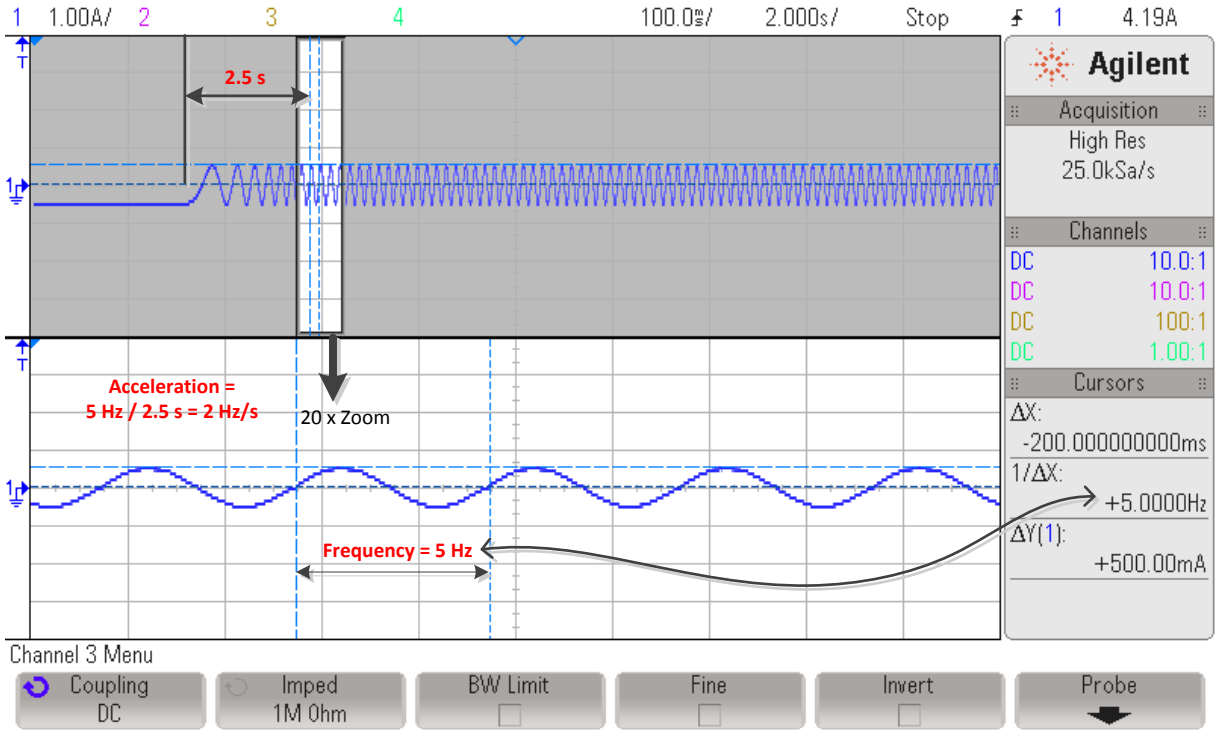


Figure 6-29. Oscilloscope Plot of ACIM RampUp Acceleration

6.6.6 CTRL_State_OnLine and EST_State_IdRated

Once the motor is running at a commanded frequency set in user.h, the IdRated identification process starts (Figure 6-30).

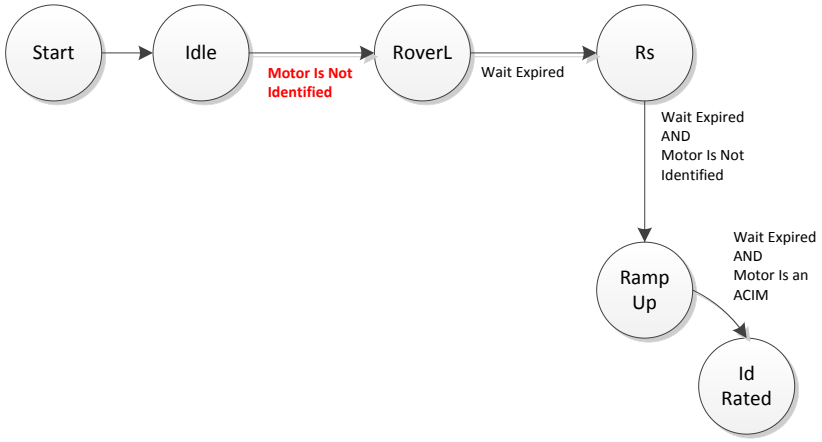


Figure 6-30. ACIM Id Rated EST State

In the EST_State_IdRated state, the estimator is used to calculate the current needed to produce a certain flux.

Copyright © 2013–2019, Texas Instruments Incorporated

During this state, several parameters in user.h and user.c are taken into account. The first one is the duration of this state which is configured in user.c file as follows:

```
pUserParams->estWaitTime[EST_State_IdRated]=(uint_least32_t)(20.0*USER_EST_FREQ_Hz);
```

During this time, the injected Id will start growing by increments defined by USER_IDRATED_DELTA in **user.h**:

```
//! \brief Defines the IdRated delta to use during estimation
//!
#define USER_IDRATED_DELTA  (0.00002)
```

Using the above delta, the current injected in the d-axis will increase until the produced flux has reached the value specified by USER_MOTOR_RATED_FLUX in user.h:

```
#define USER_MOTOR_RATED_FLUX  (0.8165*220.0/60.0)
```

For details on setting the rated flux for ACIM motors, see Section 6.4.5.4.

When the IdRated state starts the current will increase until the desired flux is present in the motor. Figure 6-31 shows the current is increase and then stabilize.

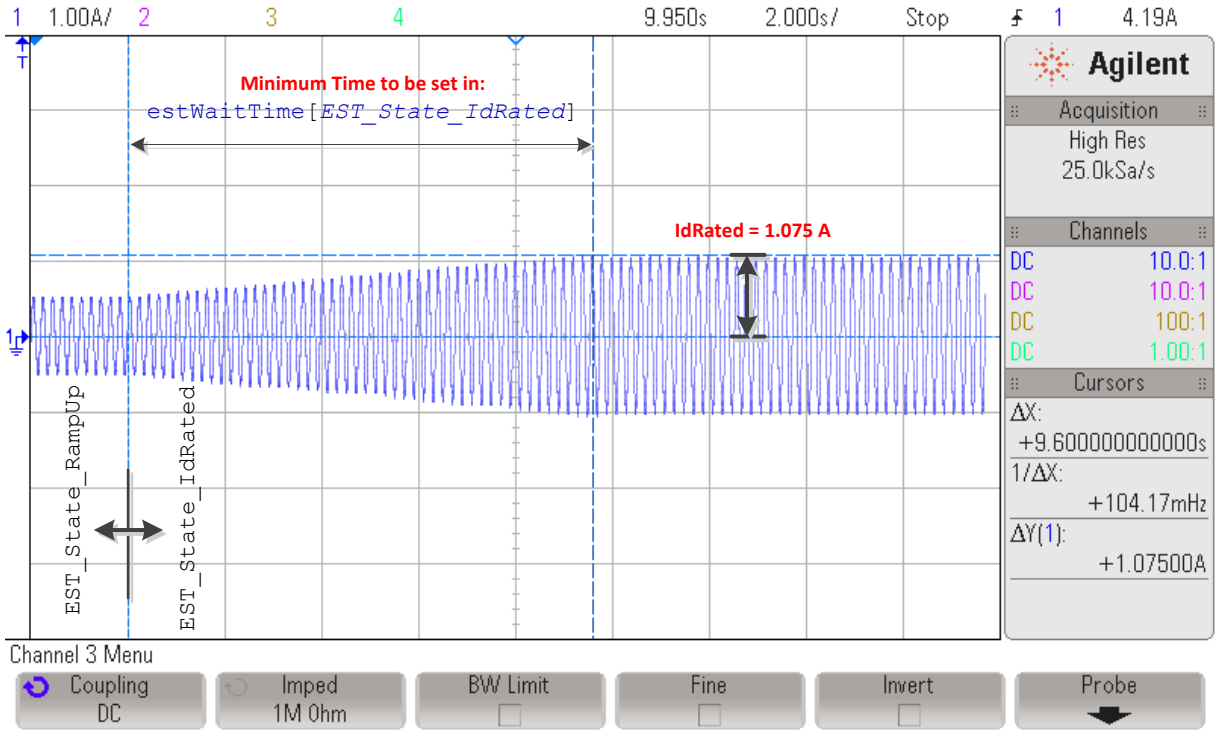


Figure 6-31. Oscilloscope Plot of Phase Current During Id Rated EST State

The time at which the Id current settles is motor dependent and it is recommended that the time configured in pUserParams->estWaitTime[*EST_State_IdRated*]is adjusted to allow the current to stabilize to a certain value without excessive oscillation.

Copyright © 2013–2019, Texas Instruments Incorporated

6.6.6.1 Reducing Oscillation to Improve Id Rated Measurement

Another parameter to tune while doing IdRated identification is the delta increments for this current. If this value is too high for a particular motor, a remaining oscillation will be present even when doing IdRated identification for a long period. For example, Figure 6-32 shows the current when the following parameter is used.

```
//! \brief Defines the IdRated delta to use during estimation
//!
#define USER_IDRATED_DELTA  (0.0001)
```

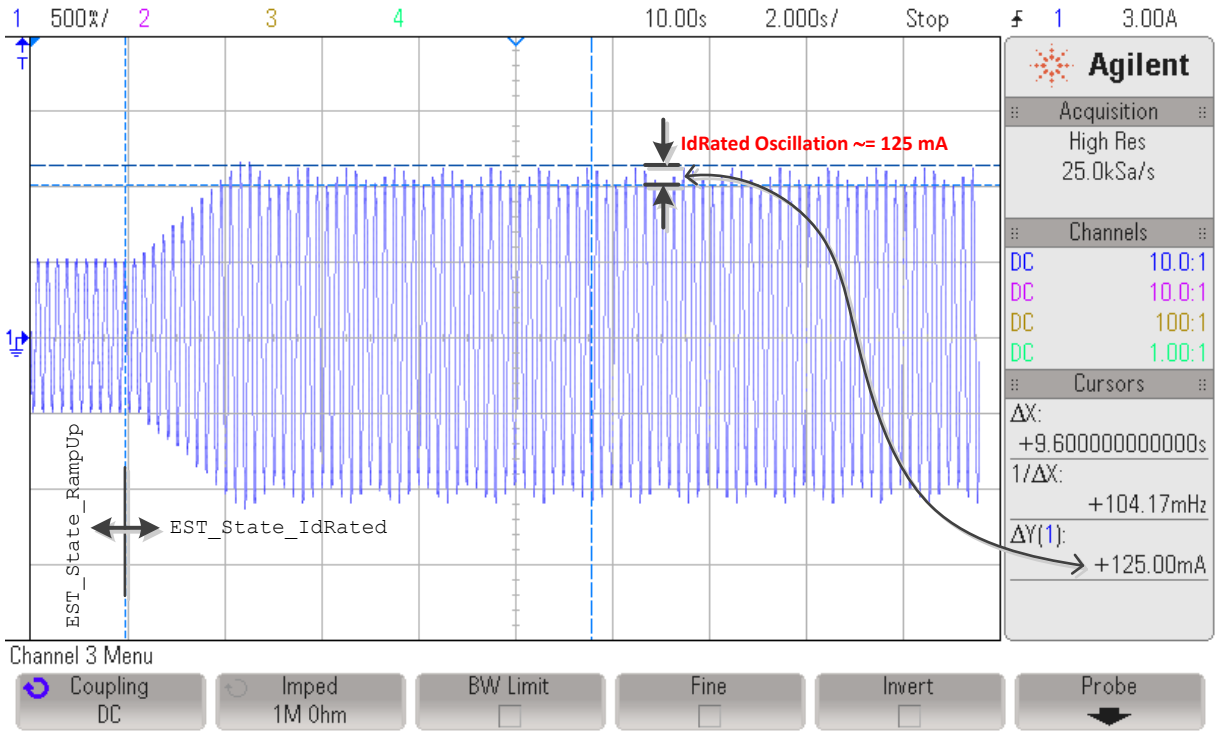


Figure 6-32. Phase Current Oscillation During Id Rated Measurement

As can be seen in the oscilloscope plot, even though the current grows much faster, the remaining oscillation does not allow the Id Rated to settle to a stable value. Trying a smaller value, 0.00002 instead of 0.0001 increases the stability of the steady state current as shown in Figure 6-33.

Copyright © 2013–2019, Texas Instruments Incorporated

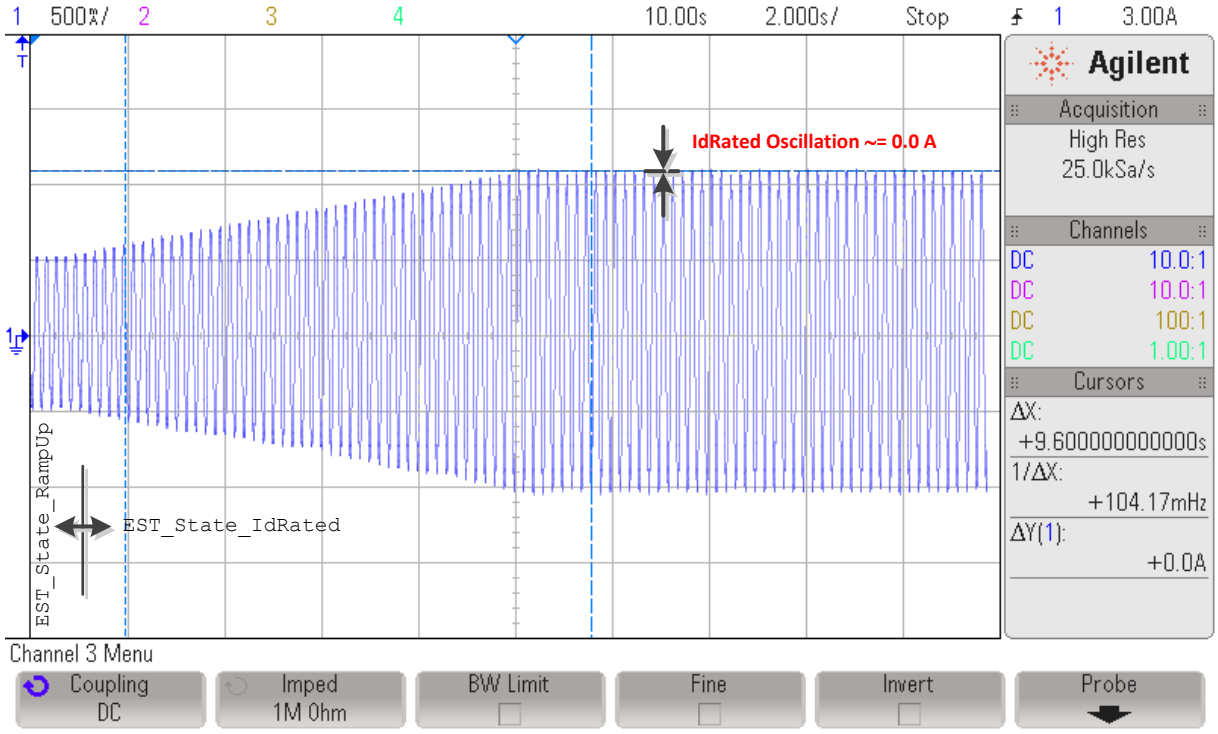


Figure 6-33. Reduced Phase Current Oscillation During Id Rated Measurement

6.6.6.2 Reading Id Rated Final Value

At the end of this state, users can read the identified Id Rated, also known as the rated magnetizing current of the ACIM motor, by using the following function:

```
// get the Id Rated, or rated magnetizing current
IdRated = EST_getIdRated(obj->estHandle);
```

6.6.7 CTRL_State_OnLine and EST_State_RatedFlux

Once the Id Rated has been identified, the next state is the Rated Flux state (Figure 6-34).

Copyright © 2013–2019, Texas Instruments Incorporated

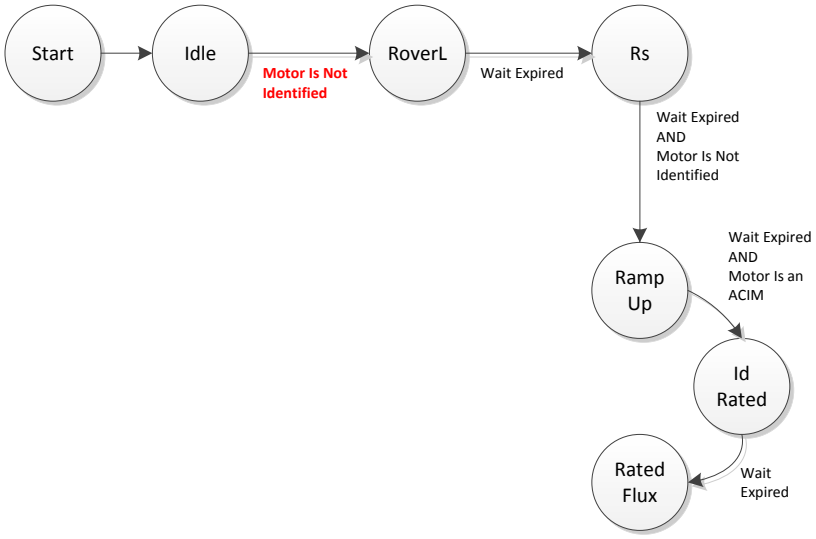


Figure 6-34. ACIM Rated Flux EST State

During this state, the previously identified Id Rated current is used in a closed loop system inside the estimator to calculate the flux linkage between rotor and stator. Only 50% of IdRated is used. At the end of this state, the computed rated flux using the Id Rated current is saved as the rated flux of the machine.

For the Rated Flux state, the current ramp-down and total measurement time are the same for both motors. For more information, see Section 6.5.6.1 and Section 6.5.6.2.

Figure 6-35 shows how the loop is closed, the current is much more stable than the IdRated state, and user might double check the estimated flux in the watch window to confirm that the flux estimation is what the original setting in user.h was.

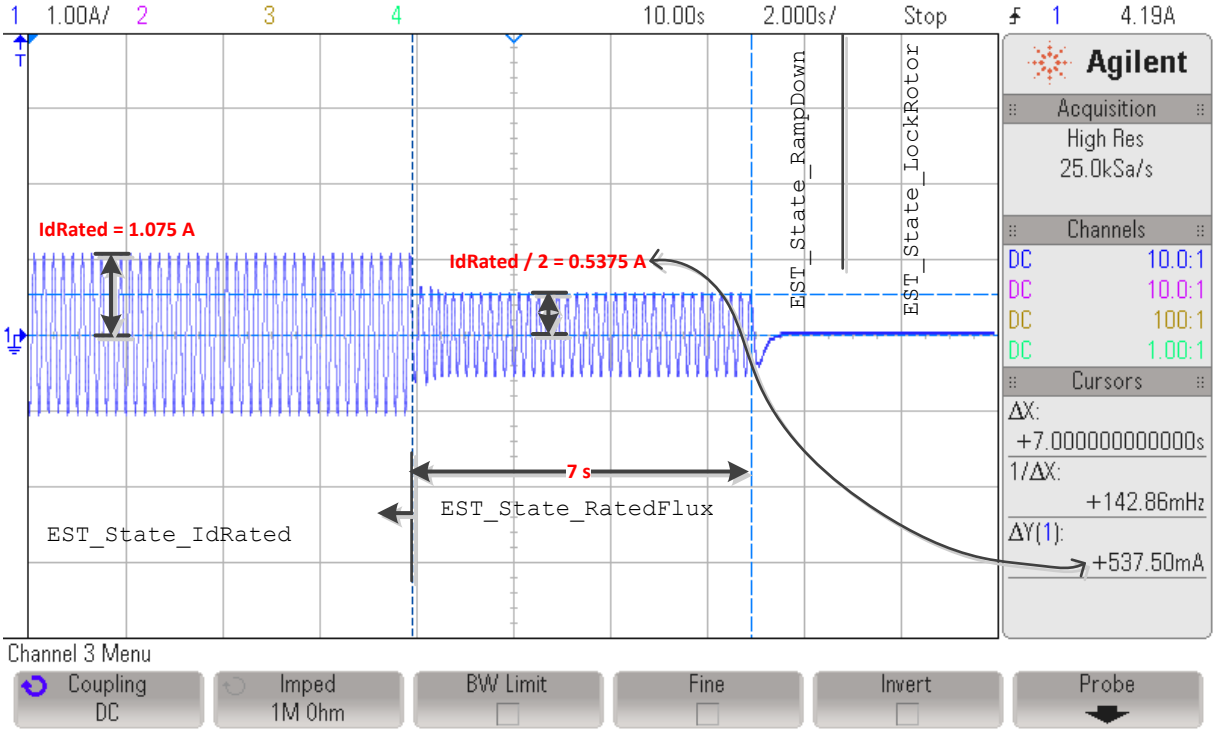


Figure 6-35. Phase Current During Id Rated EST State

Copyright © 2013–2019, Texas Instruments Incorporated

6.6.7.1 Troubleshooting Flux Measurement

See Section 6.9.

6.6.8 CTRL_State_OnLine and EST_State_RampDown

In order to remove the current from the motor windings, an intermediate state is run to ramp down the current flowing through the motor (Figure 6-36). This state is the RampDown state, with the duration set by the following in user.c file:

```
pUserParams->estWaitTime[EST_State_RampDown]=(uint_least32_t)(2.0*USER_EST_FREQ_Hz);
```

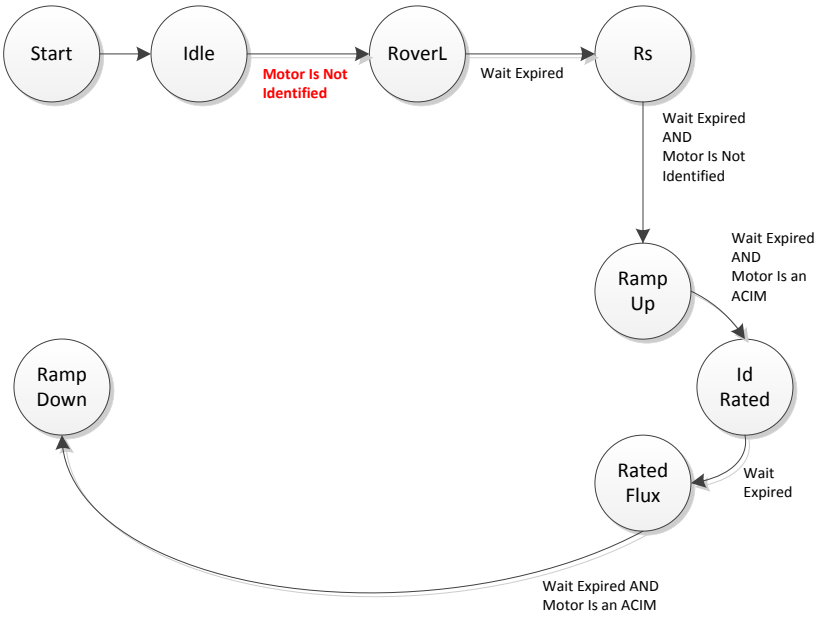


Figure 6-36. ACIM Ramp Down EST State

Figure 6-37, taken from the previous state, shows phase current being removed from the motor gradually to allow a smooth stop of the motor.

Copyright © 2013–2019, Texas Instruments Incorporated

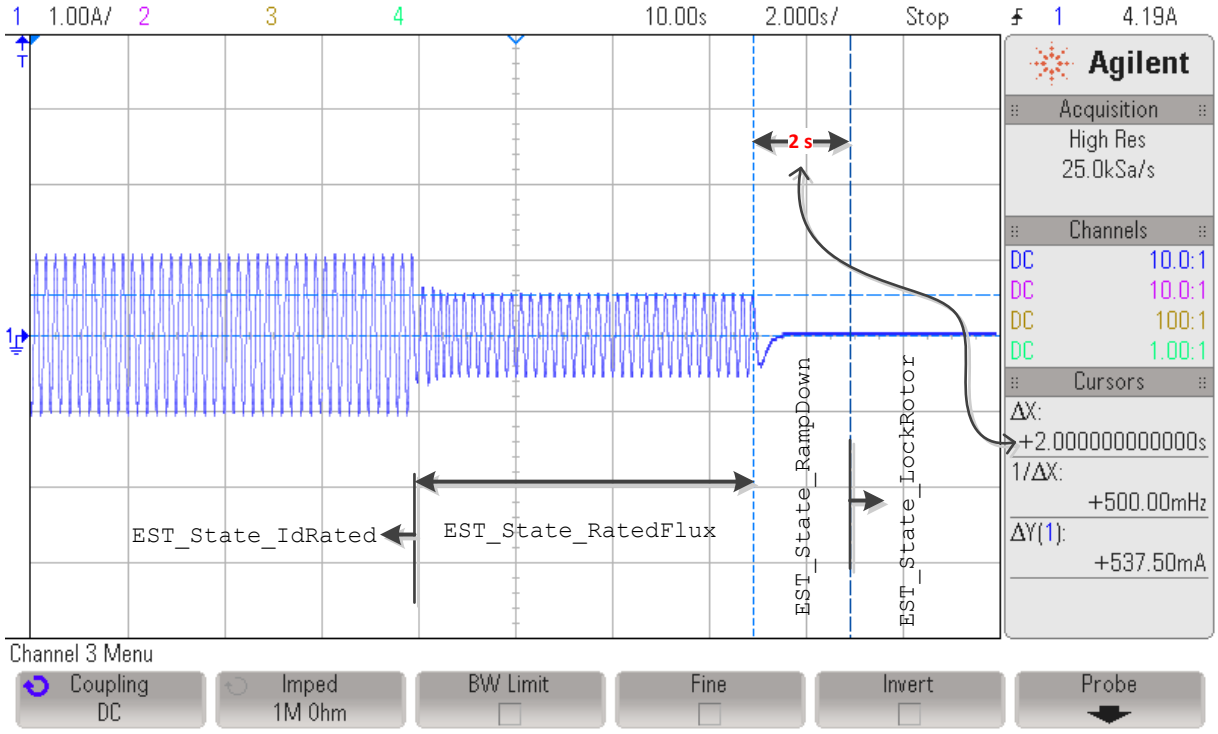


Figure 6-37. Ramp Down of ACIM Phase Current Prior to LockRotor State

6.6.9 CTRL_State_Idle and EST_State_LockRotor

During the Lock Rotor state, the state machine waits for the user to re-enable the controller once the motor's shaft has been locked (Figure 6-38). Locking the rotor is required to identify the rest of the ACIM motor parameters: series inductance (Ls) and rotor resistance (Rr).

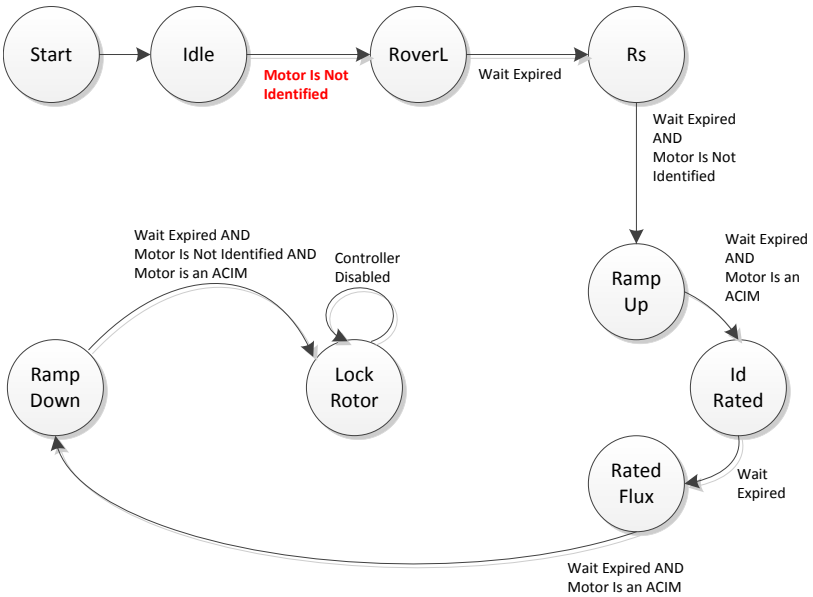


Figure 6-38. ACIM Lock Rotor EST State

Copyright © 2013–2019, Texas Instruments Incorporated

The state machine will remain in this state indefinitely until the user re-enables the controller by calling the following instruction:

```
// enable or disable the control
CTRL_setFlag_enableCtrl(ctrlHandle, TRUE);
```

6.6.9.1 Troubleshooting Locked Rotor Test

See Section 6.9.

6.6.10 CTRL_State_OnLine and EST_State_Ls

Once the locked rotor parameters are measured the stator inductance process starts (Figure 6-39).

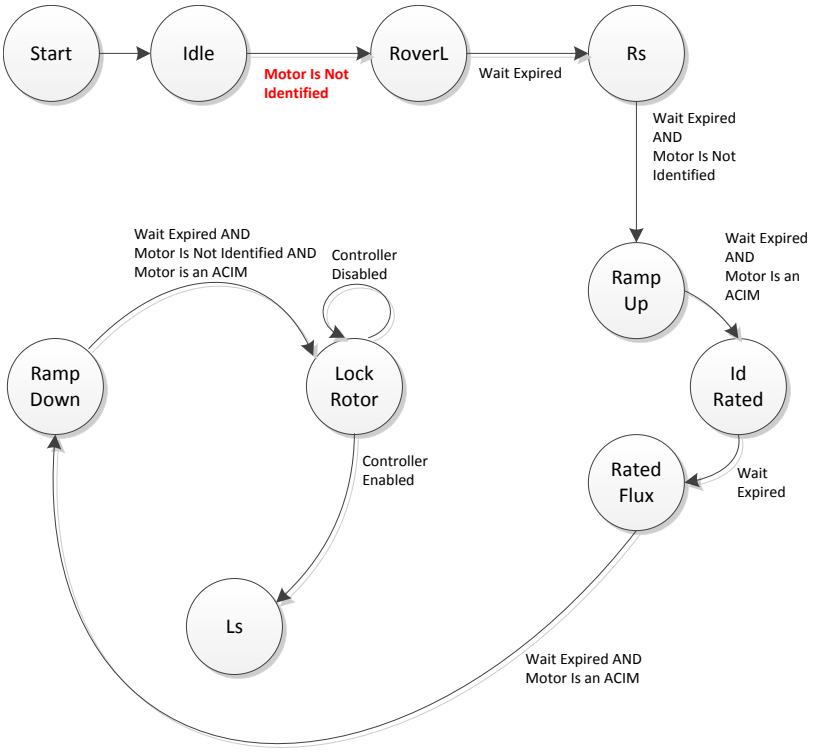


Figure 6-39. ACIM Stator Inductance EST State

For the Stator Inductance state, the steps are the same for both motors. For more information, see Section 6.5.7.

Figure 6-40 shows how the current for an ACIM motor during the Ls state and leading into the next states.

Copyright © 2013–2019, Texas Instruments Incorporated

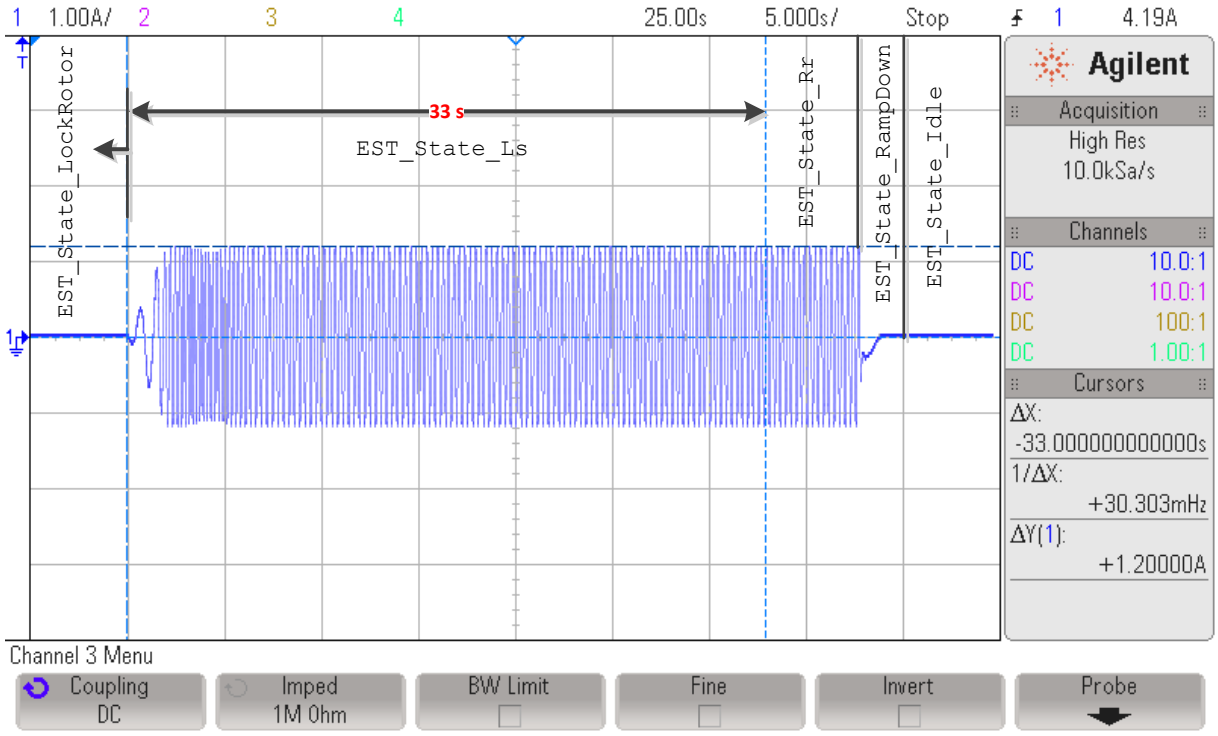


Figure 6-40. ACIM Current for the Stator Inductance EST State

6.6.11 CTRL_State_OnLine and EST_State_Rr

Once the stator inductance measurement is complete the rotor resistance process starts (Figure 6-41).

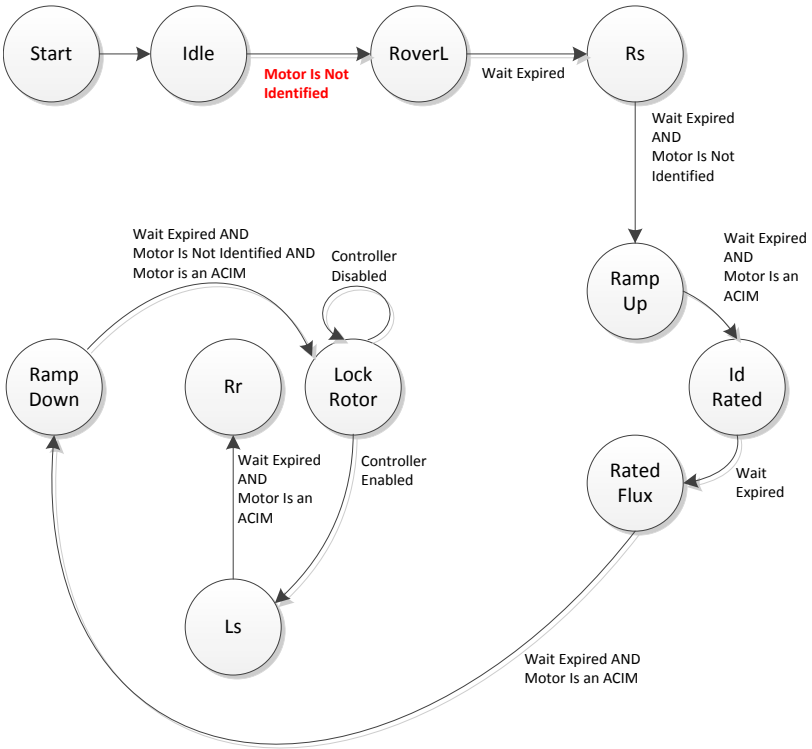


Figure 6-41. Rotor Resistance EST State

Copyright © 2013–2019, Texas Instruments Incorporated

The identification of the rotor resistance of the ACIM motor will be done for a period of time configured in this array member configured in user.c:

```
pUserParams->estWaitTime[EST_State_Rr]=(uint_least32_t)(5.0*USER_EST_FREQ_Hz);
```

Figure 6-42 shows the current being injected for Rr identification. In fact, there is no difference in the current waveform between Ls and Rr identification.

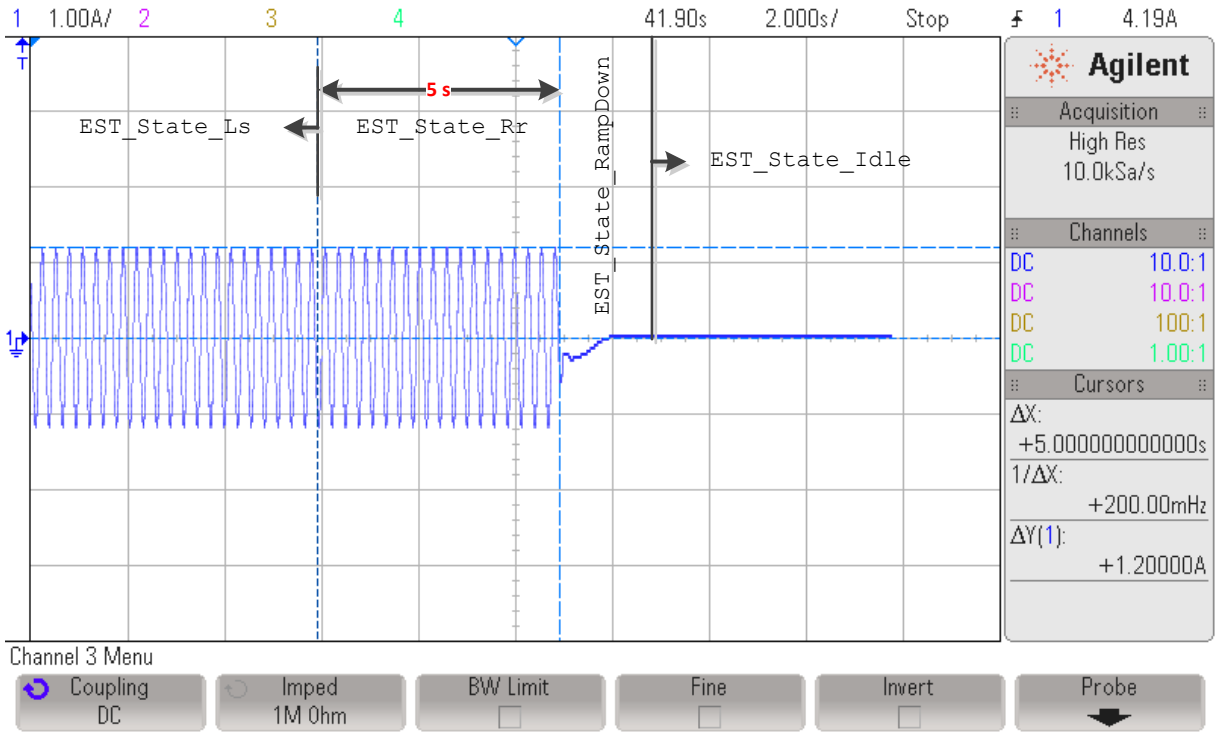


Figure 6-42. Injected Current for Rr Identification

6.6.12 CTRL_State_OnLine and EST_State_RampDown

Once the rotor resistance measurement is complete the ramp down process starts (Figure 6-43).

Copyright © 2013–2019, Texas Instruments Incorporated

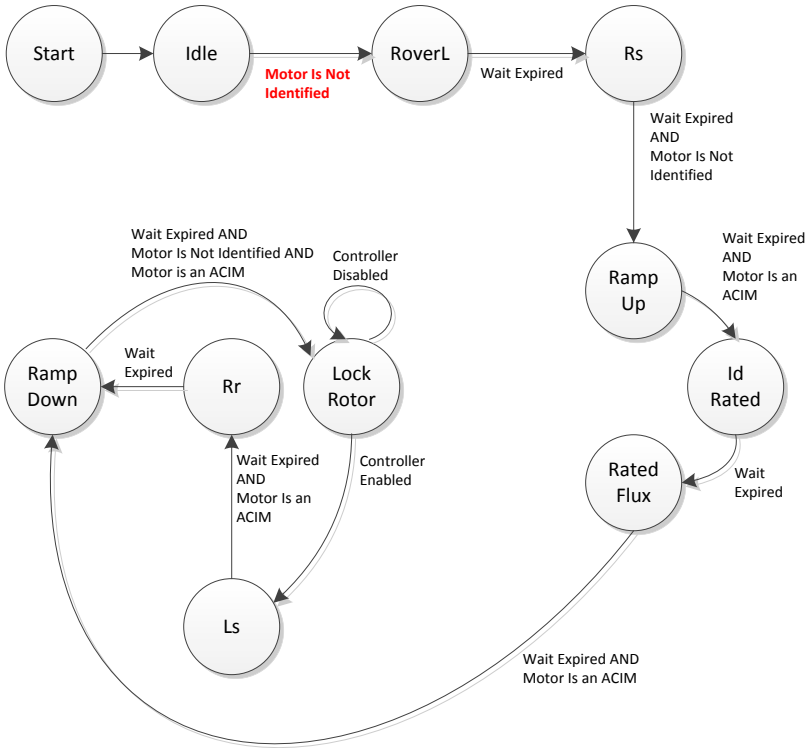


Figure 6-43. ACIM Ramp Down EST State after Completing Rr

Figure 6-44 of the ACIM current waveform after full identification, note the current is removed from the motor with a ramp. For more information about this state, see Section 6.5.8.

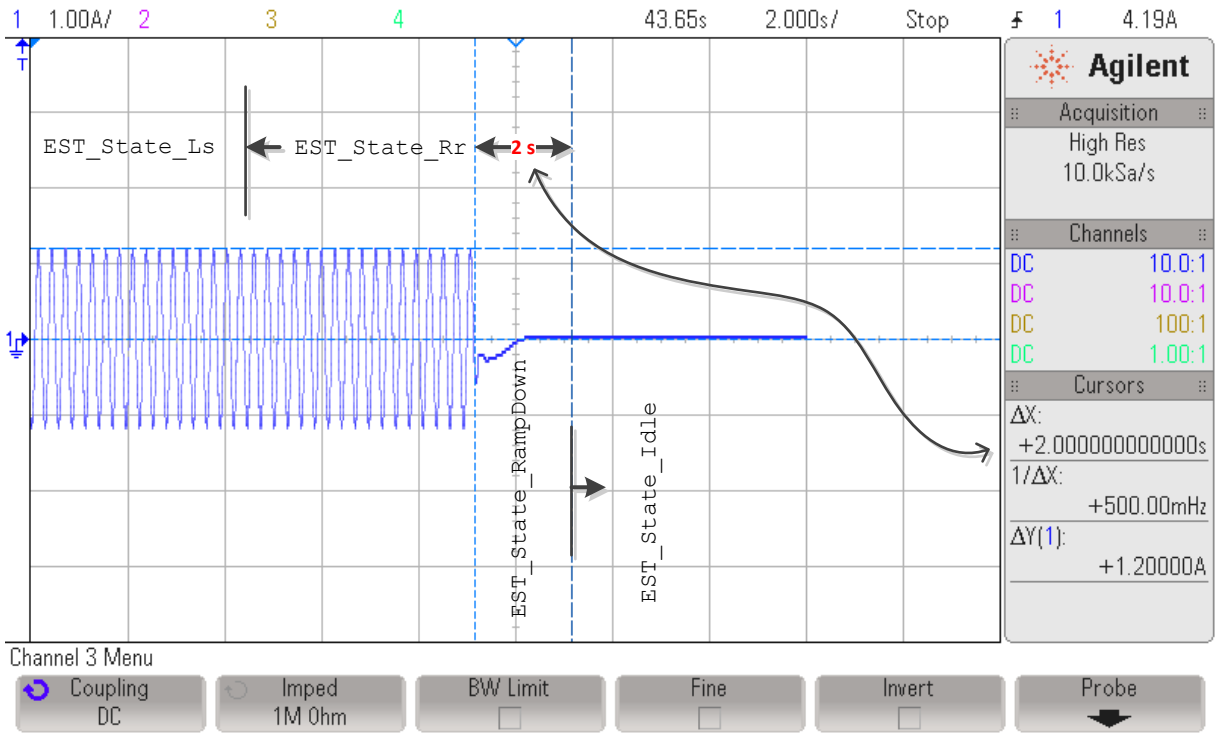


Figure 6-44. ACIM Current During Rr and RampDown

Copyright © 2013–2019, Texas Instruments Incorporated

6.6.13 CTRL_State_OnLine and EST_State_MotorIdentified

When the estimator is in the Motor Identified state, the full motor identification of the ACIM motor is complete (Figure 6-45). For more information about this state, see Section 6.5.9.

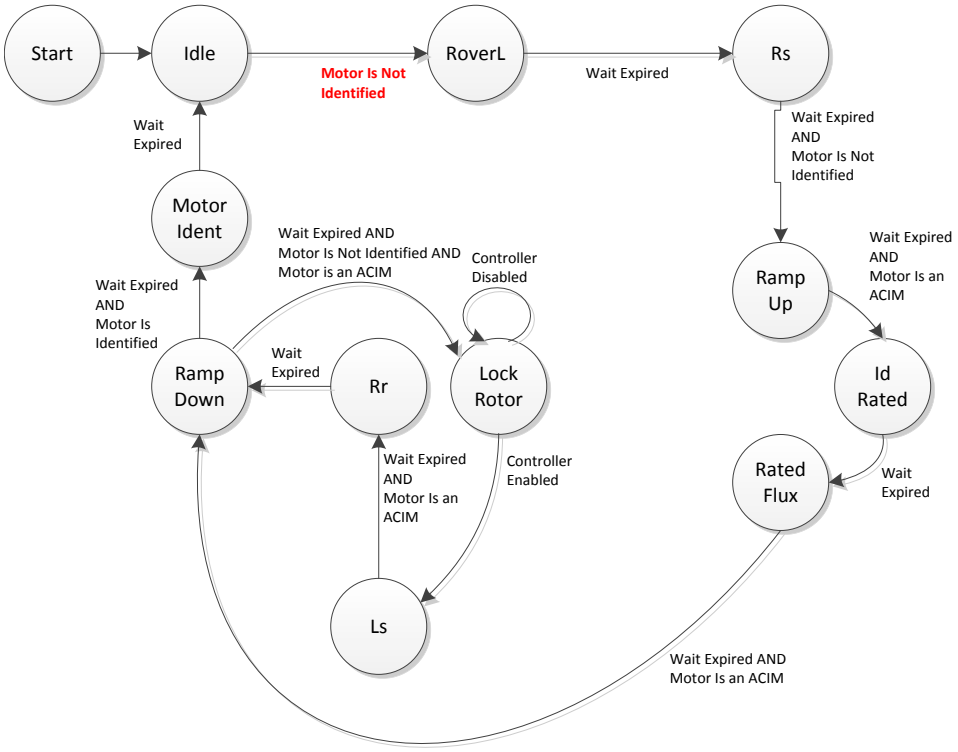


Figure 6-45. Complete ACIM Motor ID Process in EST State Diagram

The entire process of ACIM motor identification is also shown in Figure 6-46, where one phase current is plotted.

Copyright © 2013–2019, Texas Instruments Incorporated

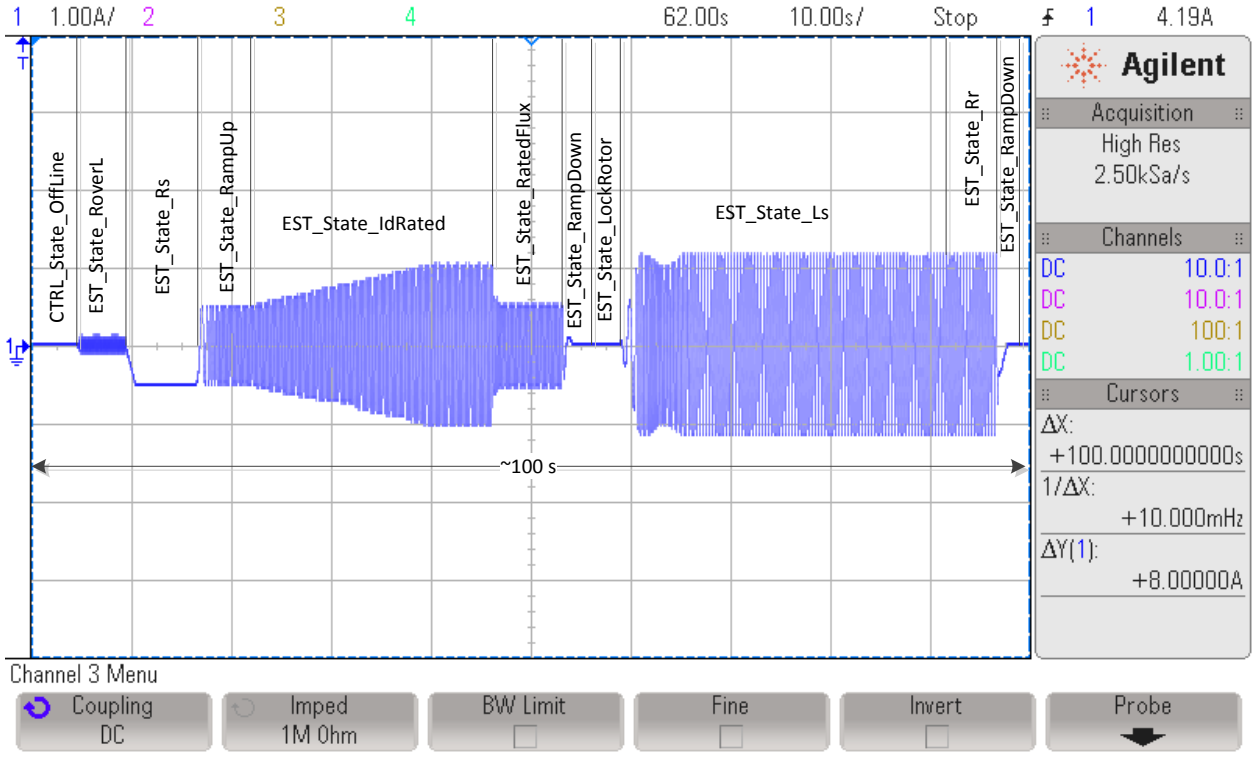


Figure 6-46. Phase Current of Entire ACIM Motor ID Process

6.6.14 CTRL_State_Idle and EST_State_Idle

After the motor is fully identified, both state machines are set to Idle.

6.7 Recalibration of PMSM and ACIM Motor Identification

Recalibration is part of the motor identification because it identifies and tunes two parameters: the offsets and the stator resistance of the motor.

6.7.1 Recalibration of PMSM and ACIM Motors After Full Identification

This section covers the recalibration of PMSM and ACIM motors. Motor recalibration is used to fine tune or recalibrate hardware offsets and stator resistance. In comparison with a full calibration of PMSM and ACIM motors, recalibration only covers three states of the estimator state machine as shown in Equation 49. The recalibration for PMSM and ACIM motors are identical. Recalibration for board Offsets and Stator Resistance can be individually enabled or disabled as explained in next sections of this document.

Copyright © 2013–2019, Texas Instruments Incorporated

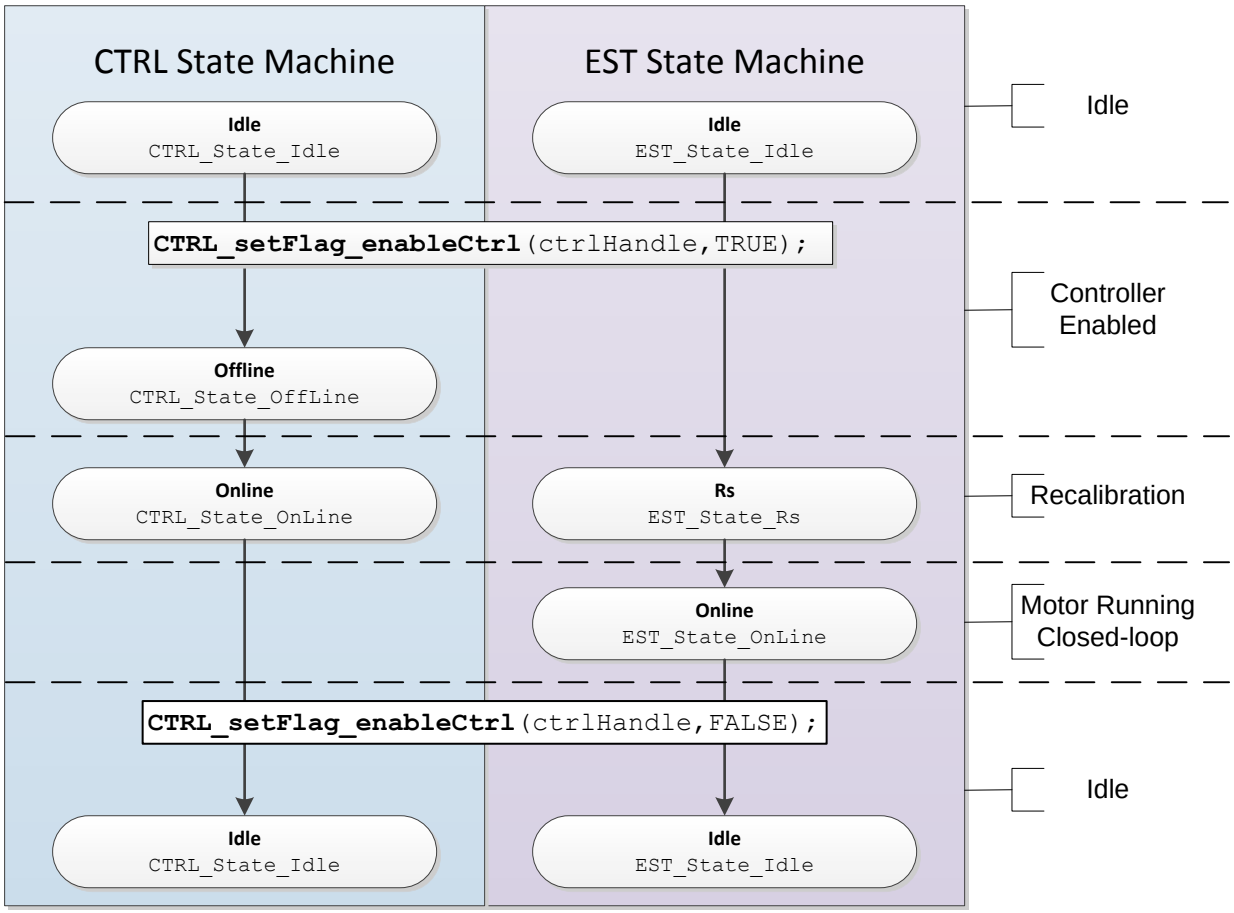


Figure 6-47. PMSM and ACIM Recalibration - CTRL and EST Sequence of States

The motor runs in closed loop when both state machines are Online, and will stay Online until the controller is disabled as shown in Figure 6-48.

Motor recalibration is executed in two cases, when the motor has been through a full identification process, and when the motor parameters have been provided through a header file, user.h. In both scenarios the states executed are identical and will be explained in detail next.

Prior to enabling the controller, the code knows that a motor recalibration will be done when these two conditions are true, when the motor has been identified and no parameters are used from user.h:

```
if( (EST_isMotorIdentified(obj->estHandle) == TRUE) &&
(CTRL_getFlag_enableUserMotorParams(ctrlHandle) == FALSE))
```


Copyright © 2013–2019, Texas Instruments Incorporated

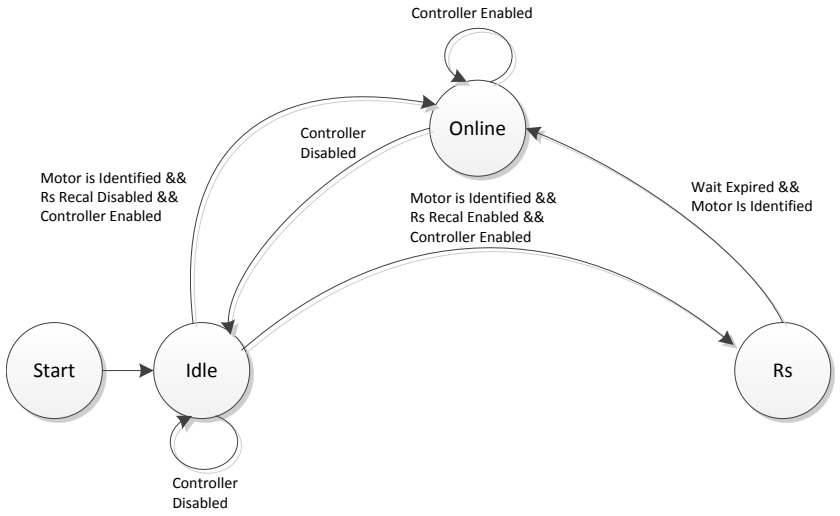


Figure 6-48. Motor Recalibration EST States

6.7.1.1 Start-Up Time Consideration

Even though recalibration takes time to execute, it is recommended to always enable both Offset Recalibration and Rs Recalibration to make sure the parameters are more accurate before running the motor, especially if the motor has been in idle for a long time. If the motor has not been run by the code for long periods of time, it is likely that ambient temperature affects the stator resistance of the motor, so Rs recalibration is recommended. Board offset recalibration is not as critical since it is hardware dependent, however, longevity of passive components and temperature variations could potentially affect the hardware offsets, so it is recommended that these offsets are tuned periodically.

6.7.1.2 CTRL_State_Idle and EST_State_Idle

Before the controller is enabled, both state machines, the controller and the estimator, are in the idle state, denoted by CTRL_State_Idle and EST_State_Idle. This is also known as the inactive state of both state machines.

6.7.1.3 CTRL_State_OffLine and EST_State_Idle

The controller is taken out of idle state by enabling it, by using the following function:

```
CTRL_s etFlag_enable Ctrl(ctrlHandle, TRUE);
```

Once the controller is enabled and with the motor already identified, the very first task performed by the controller state machine is the Offset recalibration. This only occurs if the offset calculations are enabled. To check if the offset recalibration flag is enabled, users can use the following code example:

```
if(CTRL_getFlag_enableOffset(ctrlHandle) == TRUE)
```

The motor identified flag is internally checked and can also be checked by the user with the following code example:

Copyright © 2013–2019, Texas Instruments Incorporated

```
if(EST_isMotorIdentified(obj->estHandle) == TRUE)
```

Offset recalibration is enabled by default, although the following code example can be used to enable it before the controller is enabled:

```
CTRL_s etFlag_enableOffset(ctrlHandle, TRUE);
```

And the following code example is used to disable offset recalibration.

```
CTRL_s etFlag_enableOffset(ctrlHandle, FALSE);
```

This state (CTRL_State_OffLine and EST_State_Idle) as explained in Section 6.2 is denoted by the state of the controller state machine named: CTRL_State_OffLine. The estimator state stays in the idle state (EST_State_Idle) during the controller offline state. Offset recalibration can be bypassed but offset calibration is a requirement during full motor identification. It cannot be bypassed when doing full identification of the motor.

For details of offset calibration as part of the full motor identification process, see Section 6.5.2. As seen in Figure 6-48, the "RoverL" state is not part of the recalibration process.

Figure 6-49 shows the corresponding states of the the phase current oscilloscope plots.

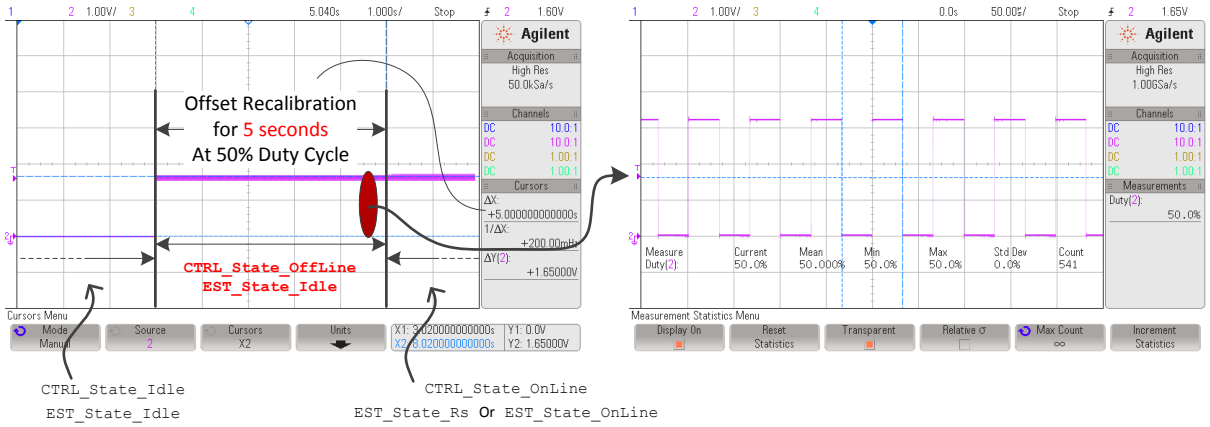


Figure 6-49. Phase Current during Offset Recalibration

6.7.1.4 CTRL_State_OnLine and EST_State_Rs

The Rs state is reached exiting the EST Idle state with the motor identified and Rs recalibration enabled (see Figure 6-48). Note that when the controller is Offline (motor is at standstill) and Offset Recalibration is enabled, offset recalibration will occur before entering the Rs state. Once the estimator state machine is in Rs state (EST_State_Rs), stator resistance recalibration is started.

Rs recalibration when the motor is at stand still is also known as Offline Stator Resistance Recalibration. A second Rs recalibration when the motor is spinning is covered in Chapter 15.

Even though the enable Rs flag is checked internally by the state machine, users can use the following code example to check that flag:

```
if(EST_getFlag_enableRsRecalc(obj->estHandle) == TRUE)
```


Copyright © 2013–2019, Texas Instruments Incorporated

Although Rs recalibration is enabled by default, the following code example can be used to enable it before enabling the controller:

```
EST_ setFlag_enableRsRecalc(obj->estHandle, TRUE);
```

And the following code example can be used to disable Rs recalibration before enabling the controller:

```
EST_ setFlag_enableRsRecalc(obj->estHandle, FALSE);
```

6.7.1.4.1 Managing Time Required for Rs Recalibration

Once the estimator is in Rs state (EST_State_Rs), the stator resistance recalibration will start. The EST_State_Rs contains three states which are executed during full motor identification:

- EST_Rs_State_RampUp
- EST_Rs_State_Coarse
- EST_Rs_State_Fine

During stator resistance recalibration only two states are executed: EST_Rs_State_RampUp and EST_Rs_State_Fine hence the recalibration time will be shorter than the full calibration during motor identification. In order to calculate the EST_State_Rs execution time during motor recalibration, consider the following two wait times configured in the user.c file:

```
pUserParams->RsWaitTime[EST_Rs_State_RampUp] = (uint_least32_t)(1.0*USER_EST_FREQ_Hz);
pUserParams->RsWaitTime[EST_Rs_State_Fine]   = (uint_least32_t)(3.0*USER_EST_FREQ_Hz);
```

Figure 6-50 shows the stator resistance recalibration. It shows the time it takes, as well as the states before and after Rs recalibration.

Copyright © 2013–2019, Texas Instruments Incorporated

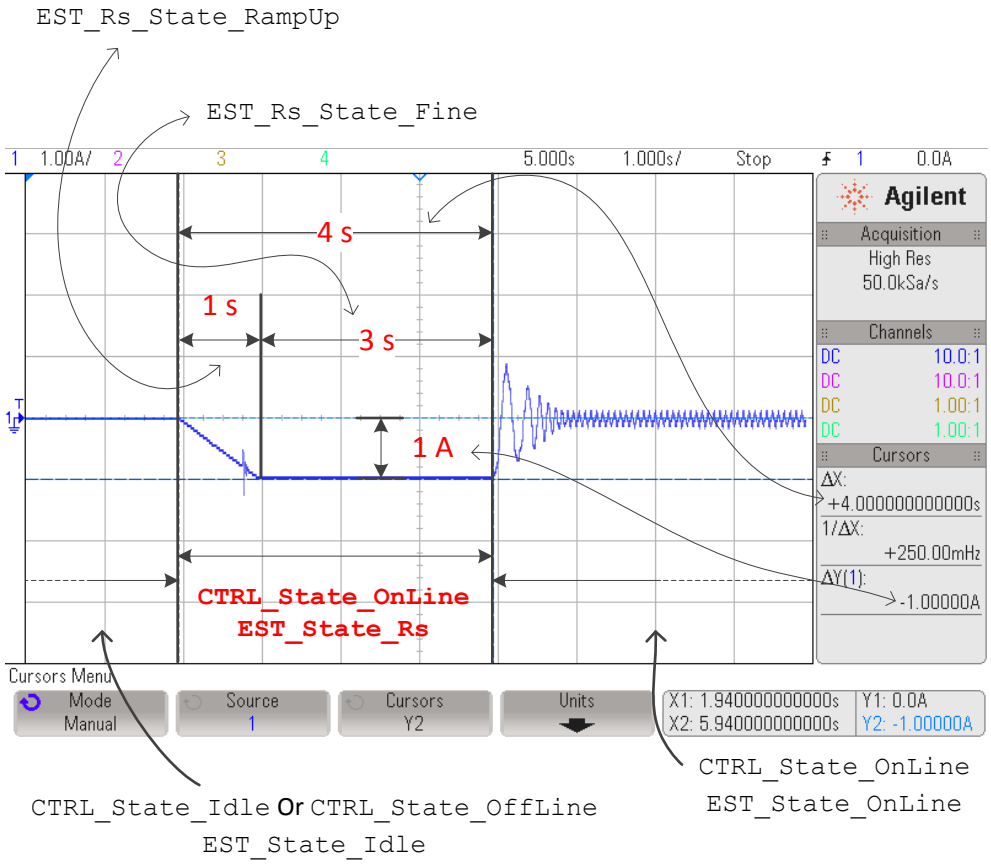


Figure 6-50. Phase Current Showing Rs Recalibration Timing

6.7.1.4.2 Software configuration for Rs recalibration

Configuration steps for a successful stator resistance recalibration are the same steps used during the Rs measurement of motor identification, for details, see Section 6.5.4. The same software configuration is used for both Rs recalibration (after motor is identified) and Rs calibration (during motor identification).

6.7.1.5 CTRL_State_OnLine and EST_State_OnLine

After the offset recalibration (if enabled) the controller state is set to Online, and after Rs recalibration (if enabled) the estimator state is set to Online. When both the controller and estimator state machines are in the Online state, the motor is running in closed loop, using the estimated angle to run the field oriented control (FOC) functional blocks, as well as the estimated speed for the speed controller.

Copyright © 2013–2019, Texas Instruments Incorporated

6.7.1.5.1 Transitioning to Online State from CTRL Online and EST Rs

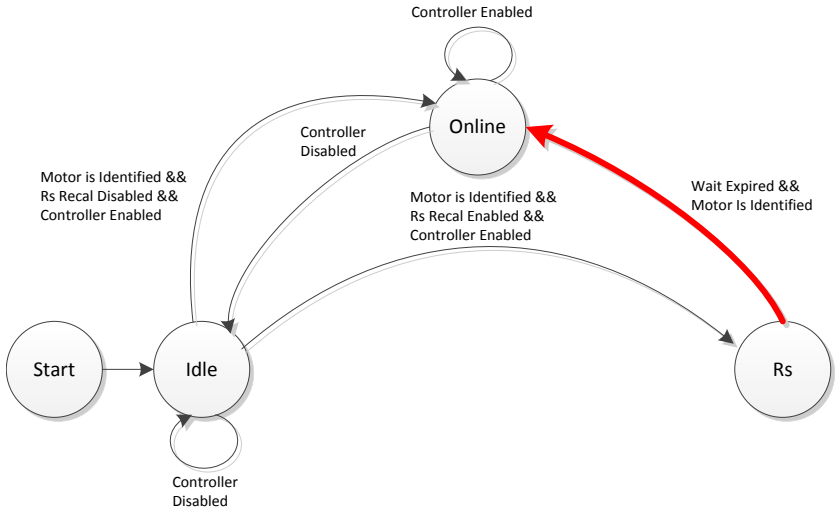


Figure 6-51. Transitioning to Online from EST Rs

Figure 6-52 shows the current of one phase with the transition from Rs recalibration to the online state.

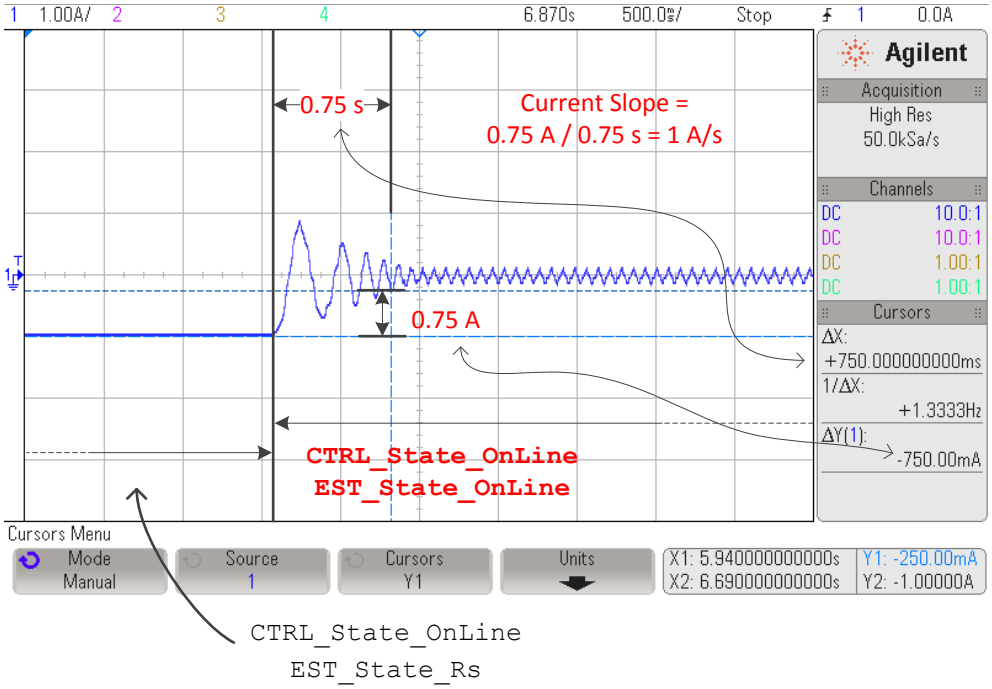


Figure 6-52. Phase Current Transitioning from EST Rs to Online

As can be seen in the plot, the phase current slopes from the current injected to recalibrate Rs to the current needed by the load. From the plot this takes 0.75 s.

In the case of no mechanical load on the motor's shaft, we can clearly see the current slope to be the current injected for Rs recalibration per second. For example, if 1 A was used to recalibrate the stator resistance, it will take 1 second for the controller to remove that current from the D-axis (ID) all the way to zero. The current remaining in the motor phases will be the IQ current, which will depend on the motor's mechanical load.

Copyright © 2013–2019, Texas Instruments Incorporated

6.7.1.5.2 Transitioning to Online state from (CTRL Idle or CTRL Offline) and EST Idle

Another transition into the online state is when the resistance recalibration is disabled. In this case, no previous current is injected into the motor. The previous state is either offline (if the offsets recalibration is enabled) or idle, as shown in Figure 6-53 and Figure 6-54.

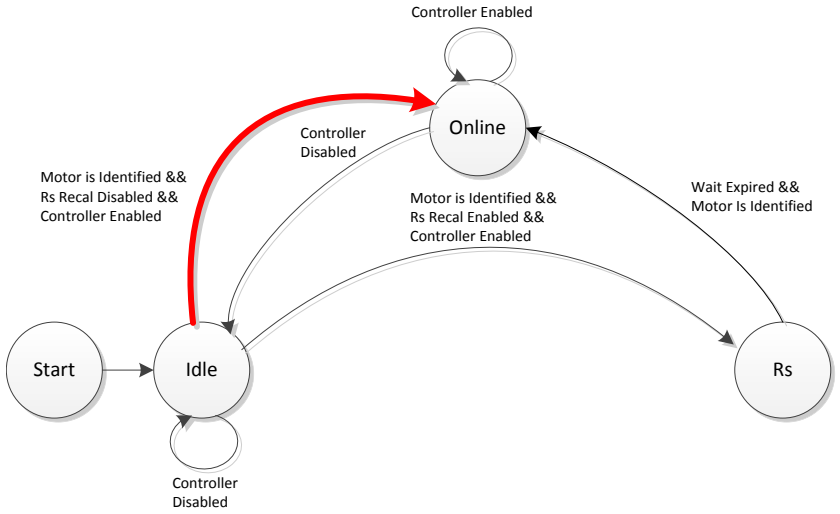


Figure 6-53. Transitioning to Online from EST Idle

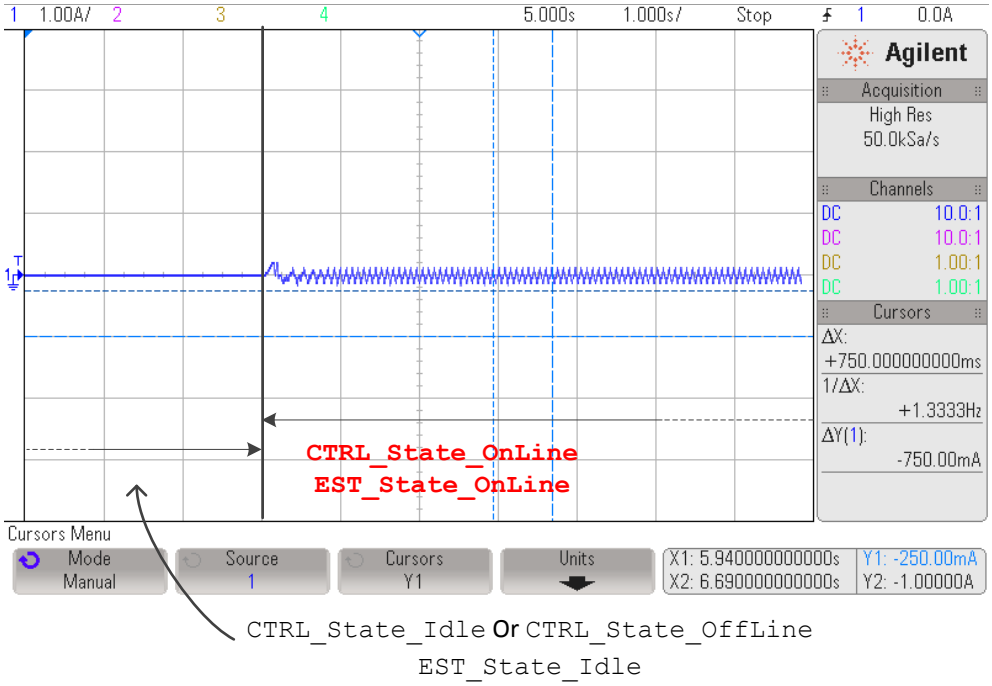


Figure 6-54. Phase Current Transitioning from EST Idle to Online

6.7.1.6 CTRL_State_Idle and EST_State_Idle

The state will remain in Online until the controller is disabled by calling the following function:

```
CTRL_s etFlag_enable Ctrl(ctrlHandle, FALSE);
```


Copyright © 2013–2019, Texas Instruments Incorporated

Calling the above function will disable the controller and will set both state machines for the controller and the estimator to the idle state.

Figure 6-55 shows current and output voltage for each state. The first state is the Offsets Recalibration state and the second is Rs Recalibration. The third stage is the online state when the commanded speed or torque is followed in closed-loop.

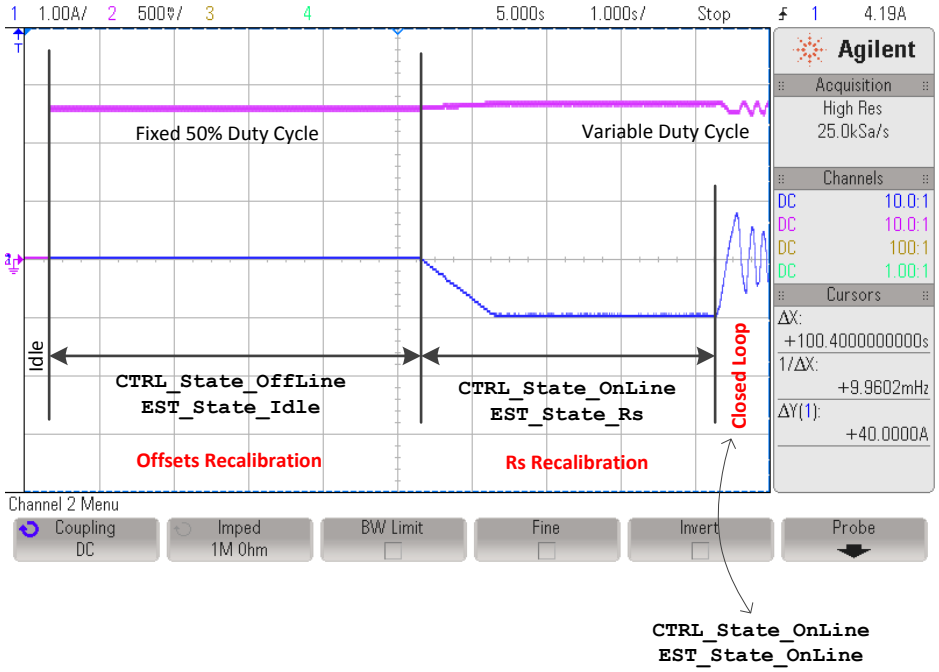


Figure 6-55. Timing of Complete Recalibration

6.7.2 Recalibration of PMSM and ACIM Motors after Using Parameters from user.h

This section covers the recalibration of PMSM and ACIM motors when motor parameters have been provided through a header file. The exact state machine used in Section 6.7.1 applies to this section.

Prior to enabling the controller, the code knows that a motor recalibration will be done using motor parameters from user.h when this condition is true:

```
if(CTRL_getFlag_enableUserMotorParams(ctrlHandle) == TRUE)
```

The same state, state transition conditions and functionality as described in Section 6.7.1 for Recalibration after Full Identification are also used for this case, recalibration using parameters from user.h.

6.8 Setting PMSM Motor Parameters in user.h

The parameters provided in user.h for PMSM motors are listed below:

```
#if (USER_MOTOR == User_PMSM)
#define USER_MOTOR_TYPE           MOTOR_Type_Pm
#define USER_MOTOR_NUM_POLE_PAIRS (4)
#define USER_MOTOR_Rs             (2.83)
#define USER_MOTOR_Ls_d           (0.0115)
#define USER_MOTOR_Ls_q           (0.0135)
#define USER_MOTOR_RATED_FLUX     (0.502)
#define USER_MOTOR_MAX_CURRENT    (4.0)
```


Copyright © 2013–2019, Texas Instruments Incorporated

Table 6-6 summarizes all the parameters that are required in user.h header file when PMSM motor identification is bypassed.

Table 6-6. PMSM Motor Parameters in user.h

PMSM Motor Parameter in user.h	PMSM Motor Parameter and Units	PMSM Motor Model Symbol
USER_Motor_Rs	Stator Resistance (Ω)	R_s
USER_Motor_Ls_d	Stator Direct Inductance (H)	L_{sd}
USER_Motor_Ls_q	Stator Quadrature Inductance (H)	L_{sq}
USER_Motor_RATED_FLUX	Rated Flux (V/Hz)	ψ

The following PMSM model can be referenced when pulling motor parameters from a motor's datasheet:

$$V_{sd} = R_s i_{sd} - \omega_m L_{sq} i_{sq} + L_{sd} \frac{di_{sd}}{dt}$$
$$V_{sq} = R_s i_{sq} + \omega_m L_{sd} i_{sd} + L_{sq} \frac{di_{sq}}{dt} + \omega_m \psi \quad (16)$$

Where:

R_s: Stator resistance

L_{sd}: D-axis stator inductance

L_{sq}: Q-axis stator inductance

ψ: Rotor flux

v_{sd}: D-axis voltage component

v_{sq}: Q-axis voltage component

i_{sd}: D-axis current component

i_{sq}: Q-axis current component

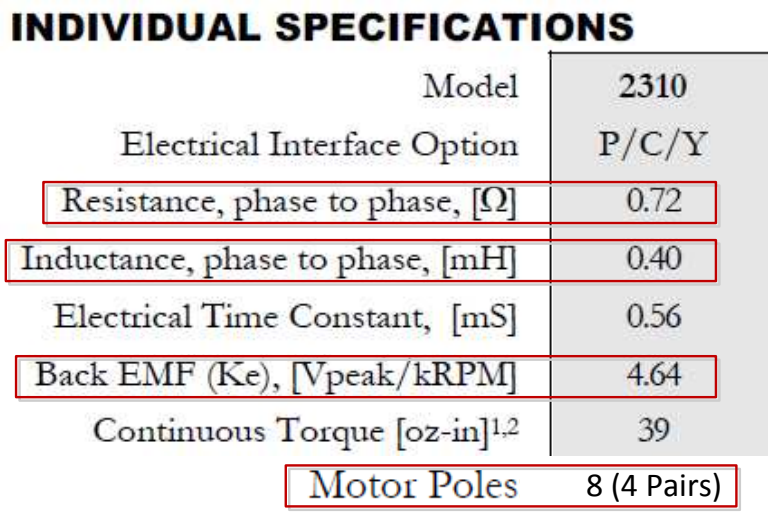
ω_m: Angular frequency of the magnetic field

The following section will cover each of these parameters, and how to get them from a typical motor manufacturer's datasheet.

6.8.1 Getting Parameters from a PMSM Datasheet

Figure 6-56 corresponds to a PMSM motor datasheet used as an example. The motor's part number is: 2310P, from company: Teknic, Inc. (www.teknic.com)

Copyright © 2013–2019, Texas Instruments Incorporated



INDIVIDUAL SPECIFICATIONS

Model	2310
Electrical Interface Option	P/C/Y
Resistance, phase to phase, [Ω]	0.72
Inductance, phase to phase, [mH]	0.40
Electrical Time Constant, [mS]	0.56
Back EMF (Ke), [Vpeak/kRPM]	4.64
Continuous Torque [oz-in][1,2]	39
Motor Poles	8 (4 Pairs)

Figure 6-56. Example PMSM Motor Datasheet

6.8.1.1 Number of Pole Pairs

The number of pole pairs is used to calculate speeds in revolutions per minute (RPM) and for some flux calculations, as shown in the rated flux calculation example. We simply use the number of pole pairs from the motor's datasheet into user.h as follows:

```
#define USER_MOTOR_NUM_POLE_PAIRS (4)
```

6.8.1.2 Stator Resistance (Rs)

The stator resistance from phase to phase shown in the previous motor's datasheet is 0.72Ω. What we need in user.h file is the phase to neutral resistance in a Y connected motor. In this case, the operation is a simple divide by 2 to convert from phase to phase resistance to phase to neutral resistance since the motor is known to be connected in Y configuration. The operation from line to line (Y connected motor) to line to neutral is as follows:

$$R_s^{user.h} = R_s^{phase\,to\,phase} \times \frac{1\Omega^{phase\,to\,phase}}{2\Omega^{phase\,to\,neutral}} = 0.72\Omega \times 0.5 = 0.36\Omega \quad (17)$$

The resulting value is then written in user.h as follows:

```
#define USER_MOTOR_Rs (0.36)
```

In the motor, if delta connected as opposed to Y connected, then a conversion from delta to Y needs to be done to set the resistance value. For example, if the delta Rs (delta) value is known to be 3 Ohms, the Rs (Y) value would be Rs (Y) = Rs (delta) / 3 = 1 Ohms.

6.8.1.3 Stator Inductance (Ls_d and Ls_q)

In the case of a non-salient PMSM motor, Ls_d and Ls_q are equal. In this example, a phase-to-phase stator inductance is shown with a value of 0.40 mH. We need to convert that value to phase-to-neutral inductance in a Y connected motor, following the same procedure as used with the previous parameter we simply divide by 2 as follows:

Copyright © 2013–2019, Texas Instruments Incorporated

$$L_{s_d}^{user.h} = L_{s_q}^{user.h} = L_s^{phase\,to\,phase} \times \frac{1H^{phase\,to\,phase}}{2H^{phase\,to\,neutral}} = 0.40mH \times 0.5 = 0.20mH \quad (18)$$

The resulting value is then written in user.h as follows:

```
#define USER_MOTOR_Ls_d (0.0002)
#define USER_MOTOR_Ls_q (0.0002)
```

In the case of different Ls_d and Ls_q, simply set the corresponding value to the correct definition in user.h. In the motor, if delta connected as opposed to Y connected, then a conversion from delta to Y needs to be done to set the resistance value. For example, if the delta Ls_d (delta) value is known to be 0.3 mH, the Ls_d (Y) value would be Ls_d (Y) = Ls_d (delta) / 3 = 0.1 mH.

6.8.1.4 Rated Flux (ψ)

The last parameter required from the motor's datasheet is the rated flux. This particular one can be calculated from the provided parameter, Back EMF constant, or Ke, which is provided in Vpeak/kRPM. The following unit conversion is needed in order to set the value correctly the header file:

$$\psi_s^{user.h} = K_e^{\frac{Vpeak}{kRPM}} \times \frac{60\,s}{1\,min} \times \frac{1k\,Rev}{1000\,Rev} \times \frac{1\,Rev}{PolePairs\,Cycles} \times \frac{1V_{line\,to\,neutral}}{\sqrt{3}\,V_{line\,to\,line}}$$

$$\psi_s^{user.h} = 4.64 \times 60 \times \frac{1}{1000} \times \frac{1}{4} \times \frac{1}{\sqrt{3}} = 0.0402\,V/Hz \quad (19)$$

The resulting value is then written in user.h as follows:

```
#define USER_MOTOR_RATED_FLUX (0.0402)
```

It is also common that the rated flux of the motor is not provided by the motor manufacturer's datasheet, and an alternative way to measure the rated flux is by measuring the phase-to-phase voltage when the motor to be measured is spun by another motor. In other words, we run the motor in generator mode, and connect an oscilloscope on the motor phases to see the voltage production at a given speed. The speed to run the motor in generator mode to measure flux should be high enough to overcome any cogging torque. For example, considering the same motor, we run it with another motor connected to its shaft, and plot the phase to phase voltage. In this example we run the motor at about 1000 RPM, but can be any speed as long as the motor runs at a constant speed. Figure 6-57 shows the result.

Copyright © 2013–2019, Texas Instruments Incorporated

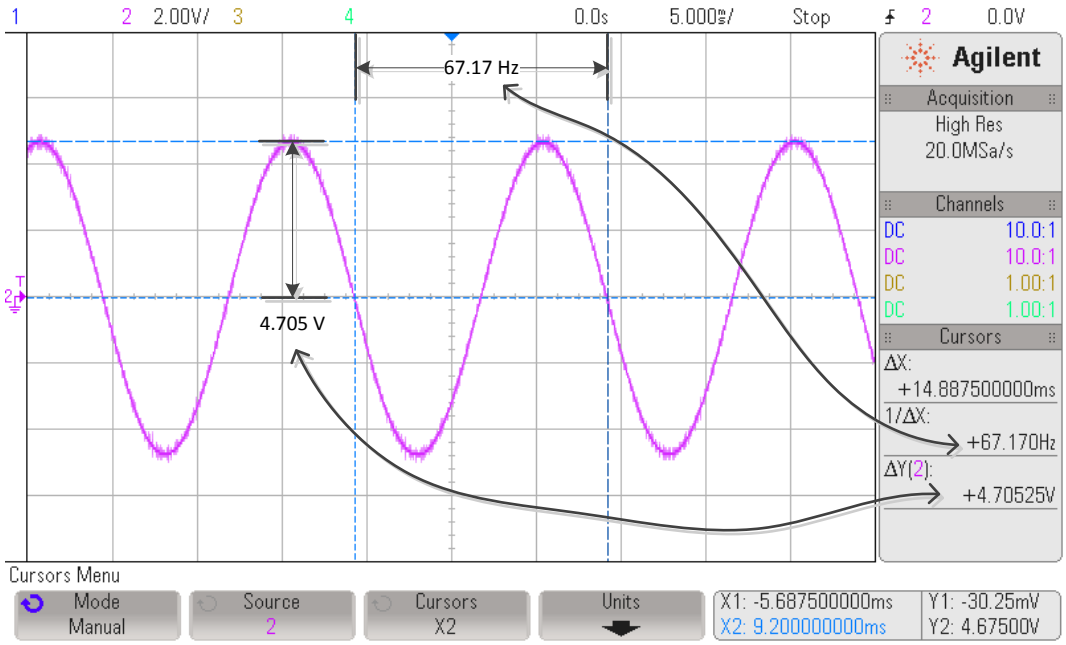


Figure 6-57. Determining Motor Flux from Phase Voltage of Motor in Generator Mode

Considering 4 pole pairs, and the fact that Ke is in Vpeak/kRPM, the measured Ke with the oscilloscope is:

$$K_e^{\frac{Vpeak}{kRPM}} = \frac{4.705\,V}{67.17\,Hz} \times \frac{1\,min}{60\,s} \times \frac{1000\ Rev}{1k\,Rev} \times \frac{PolePairsCycles}{1\,Rev} = 4.67\,Vpeak/kRPM \quad (20)$$

And to calculate what we configure in user.h is as follows, based on the measured value:

$$\psi_s^{user.h} = 4.67 \times 60 \times \frac{1}{1000} \times \frac{1}{4} \times \frac{1}{\sqrt{3}} = 0.0404\,V/H \quad (21)$$

This can also be calculated straight from the oscilloscope measurement as follows:

$$\psi_s^{user.h} = \frac{4.705\,V}{67.17\,Hz} \times \frac{1}{\sqrt{3}} = 0.0404\,V/Hz \quad (22)$$

```
#define USER_MOTOR_RATED_FLUX (0.0404)
```

6.9 Troubleshooting Motor Identification

6.9.1 General Checklist

[Coming in next release of user's guide]

Copyright © 2013–2019, Texas Instruments Incorporated

6.9.2 Troubleshooting PMSM Motor Identification

6.9.2.1 Identifying PMSM Motors When Load Cannot be Detached

InstaSPIN-FOC requires a few parameters from the motor to run with best performance. The majority of the time this is only done once, and requires the motor to be removed from any mechanical load so it is able to spin freely. A few steps during the identification process require motor rotation in open loop. That is the reason why we require no mechanical load in the motor since it is easy to stall a motor when running in open loop control. When motor can be removed from mechanical load, there is no issue with InstaSPIN-FOC's identification process, and at the end, motor parameters are available in a watch window for future use.

On the other hand, some applications are, by definition, attached to a mechanical load. Examples of these applications are compressors, some direct drive washing machines, geared motors with sealed enclosures, etc. For those applications, the user requires to run motor identification differently in order to extract motor parameters with no or minimum motor rotation.

6.9.2.2 Can Motor Rotate with the Attached Load?

The first step is to make sure that the motor doesn't rotate with load. In some cases, the open loop tests can in fact rotate the motor even with some load. If the motor stalls at any point during the identification process, proceed to next step.

6.9.2.3 Run First Three Steps of Identification

During motor identification, there are three initial steps that don't require rotation of the shaft. The first one calculates hardware offsets. The second one injects a high frequency current sine wave to identify what we call high frequency resistance (Rhf) and high frequency inductance (Lhf). The third step is to identify the stator resistance (Rs) by injecting a DC current. During this step, take note of the high frequency inductance (Lhf) and stator resistance (Rs). The two required variables in this step are available by using the following function calls from the library:

```
// get Lhf
gLhf = CTRL_getLhf(ctrlHandle);
// get the stator resistance
gRs = EST_getRs_Ohm(obj->estHandle);
```

6.9.2.4 Run Using Motor Parameters from user.h

Even though there are ways to calculate the flux of the machine without motor identification (see Section 6.8.1.4), this process describes a completely unknown motor. So far we have two parameters we can use, a rough estimate of inductance (Lhf) and an accurate stator resistance (Rs). We are missing one parameter from PMSM motors, the flux of the machine. For this step of the identification process, we will use the two known parameters and an arbitrary value of the flux. Keep in mind that once the motor runs in closed loop, the estimated flux will converge to the actual flux of the machine, so we only need to get the first guess close enough to make it converge to the real value.

Take a look at specific defines from user.h.

```
#define USER_MOTOR_NUM_POLE_PAIRS (3)
```


Copyright © 2013–2019, Texas Instruments Incorporated

The number of poles is only important to get the correct RPM reading from the library, but all the library cares about is electrical Hz, and this is not affected by the number of pole pairs. Take your best guess on this parameter. Once the motor is running, this can be changed with the correct number of poles, by commanding a speed reference of 60 RPM, and making sure the motor rotates one revolution in one second.

```
#define USER_MOTOR_Rs (0.8)
```

This parameter should be the one obtained in step 2, from variable gRs.

```
#define USER_MOTOR_Ls_d (0.01)
```

```
#define USER_MOTOR_Ls_q (0.01)
```

For the inductance, we need to use the high frequency inductance obtained in step 2. This is a rough approximation of the inductance of the motor. Having a different inductance compared to the real one limits the performance during high dynamics. For example, if motor needs to run with torque steps, or speed steps, not having the right inductance can be an issue, but having full torque operation with slow dynamics can be done using high frequency inductance instead of the real inductance. So for this step, copy the value obtained from gLhf in both Ls_d and Ls_q.

```
#define USER_MOTOR_RATED_FLUX (0.5)
```

Flux is estimated by the libraries, so user needs a value which is close enough to the real one in order to have the real flux of the machine based on the estimator output. Using an arbitrary number can cause some saturation of the estimated flux, since this value is limited internally. This can be easily done by looking at the estimated flux using the following function:

```
// get the flux
gFlux_VpHz = EST_getFlux_VpHz(obj->estHandle);
```

If gFlux_VpHz is clamped to a lower than original value set in user.h, decrease the value in half, and if it is clamped to a higher than original value, increase it by doubling the value. Once the motor runs in closed loop with a value on the gFlux_VpHz that varies slightly, then take note of this value and put it back into the USER_MOTOR_RATED_FLUX.

6.9.2.5 Troubleshooting Current Controller Stability during RoverL Identification

[TBD]

Copyright © 2013–2019, Texas Instruments Incorporated

6.9.2.6 Troubleshooting Motor Shaft Stopping During Ramp-Up

The motor shaft needs to start moving at all times during the ramp-up process. This means that in case of motors with some load on the shaft, as well as motors with high cogging torque, the current used to ramp up the motor might have to be increased, as high as needed to keep the shaft moving. Start with 10% of the rated current of the motor, and increase by increments of 10% of the rated current until the motor shaft is in continuous motion during the entire ramp-up process.

```
#define USER_MOTOR_RES_EST_CURRENT (1.0) //increase in 10% steps as needed
```

6.9.2.7 Troubleshooting Motor Shaft for Smoother Ramp

If the time required to ramp-up is extended, users should also consider setting a slower ramp-up acceleration to allow a smoother ramp. For example, for high inertia loads such as direct drive washing machines, the ramp up time can be extended to allow a smoother ramp up, as can be seen in Figure 6-58.

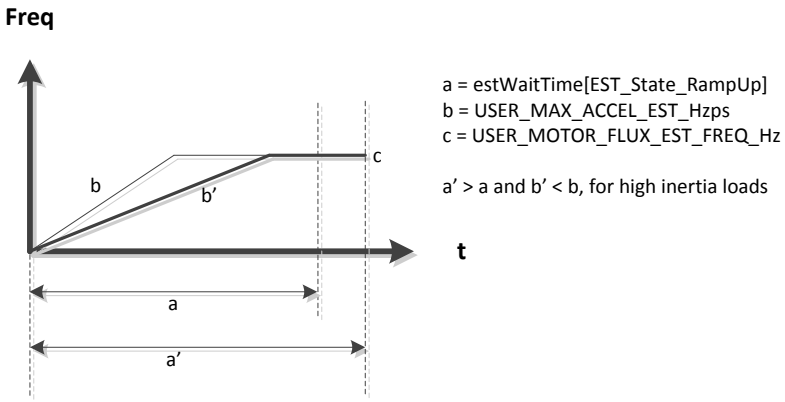


Figure 6-58. RampUp Timing with change in Acceleration

Figure 6-59, taken during the ramp up start, shows the acceleration in Hz/s.

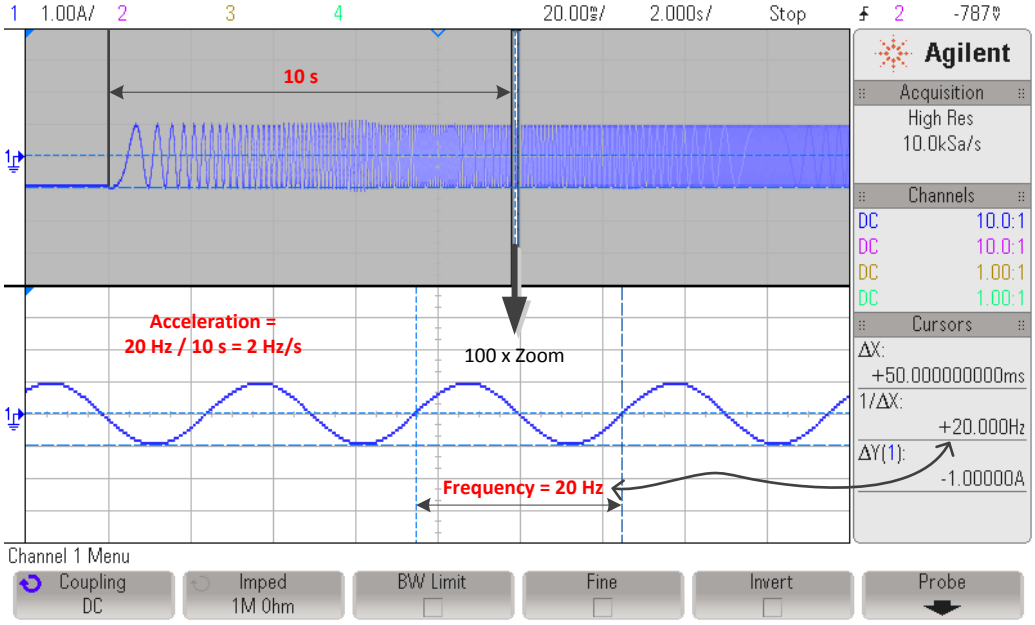


Figure 6-59. PMSM RampUp Acceleration

Copyright © 2013–2019, Texas Instruments Incorporated

6.9.2.8 Troubleshooting Flux Measurement

The monitored value of flux is considered unstable if any of the following is observed:

- saturating and not varying
- any negative value
- more than 10% variation, when the estimator state machine moves on to the next state, EST_State_Ls

If it is unstable, it is recommended to increase the time where the fine tuning of the rated flux is done (pUserParams->FluxWaitTime[*EST_Flux_State_Fine*]) in user.c.

If the motor stops spinning at any point during the identification of the flux, halt the identification process and retry it with a higher frequency by changing the following value set in user.h:

```
// During Motor ID, maximum commanded speed in Hz
#define USER_MOTOR_FLUX_EST_FREQ_Hz (20.0) (40.0)
```

The default frequency is set to 20 Hz and this will work for most PMSM motors. Increase with increments of 10 Hz until the motor does not stop spinning during flux identification.

6.9.2.9 Troubleshooting Ls Identification

It is important that the motor is spinning during the entire inductance identification process. If at any point during the identification of the inductance the motor stops, increase the frequency used for Flux and inductance estimation, for example, if 20 Hz was used, try 40 Hz. Also, the Flux estimation frequency must be increased when the motor is known to have a low inductance. In this context, low inductance is considered when a single digit µH motor is used. This frequency is used for both the identification of the flux, and the identification of the inductance.

```
// During Motor ID, maximum commanded speed in Hz
#define USER_MOTOR_FLUX_EST_FREQ_Hz (20.0) (40.0)
```

If a shorted identification time is needed per user's requirements, the inductance can be monitored in order to know when the inductance identification stabilizes. A stable identified inductance would be when the watch window value does not vary more than 5% as a general guideline. User can measure the time from the start of the EST_State_Ls state up to when the identification of the inductance shows a stable number using the following code example:

```
// get the stator inductance in the direct coordinate direction
gMotorVars.gLsd_H = EST_getLs_d_H(obj->estHandle);

// get the stator inductance in the quadrature coordinate direction
gMotorVars.gLsq_H = EST_getLs_q_H(obj->estHandle);
```

Once the time for a stable Ls is known, this time can be then entered into the following time so that the overall motor identification process is shorter: pUserParams->LsWaitTime[*EST_Ls_State_Fine*].

6.9.2.9.1 Identifying Low Inductance PMSM Motors

There are several considerations to take into account for low-inductance PMSM motors. Some considerations are regarding the hardware used for motor identification and some other considerations are related to the configuration of the software.

Copyright © 2013–2019, Texas Instruments Incorporated

6.9.2.9.1.1 Hardware Considerations

When identifying low-inductance PMSM motors, it is recommended to have the voltage divider for the voltage feedback as low as possible. For example if the voltage used to run the motor is 24 V, then the voltage resistor divider should be 26 V or so. This allows the maximum number of bits of the ADC converter when measuring the voltage feedback.

Once the hardware has been changed, update the USER_ADC_FULL_SCALE_VOLTAGE_V definition in user.h.

Usually, the low-inductance motors are high-speed motors, so the flux is small. The user needs to update the USER_IQ_FULL_SCALE_VOLTAGE_V to a value that lets the identification process identify the flux of the motor. The following equation can be used to determine a value for the definition of USER_IQ_FULL_SCALE_VOLTAGE_V:

Minimum Flux that can be identified (V/Hz) = USER_IQ_FULL_SCALE_VOLTAGE_V/ USER_EST_FREQ_Hz/0.7 (23)

6.9.2.9.1.2 Software Considerations

As far as the software configuration is concerned, it is recommended to identify R/L constant with a higher frequency. For majority of the low inductance motors tested, a 300 Hz frequency for R/L is enough. Updates to R/L frequency need to be updated in:

```
#define USER_R_OVER_L_EST_FREQ_Hz (300)
```

The second consideration is to have a higher frequency to identify the inductance of the motor. This is part of the motor parameters. The following example is used to identify a motor with a few tens of µH.

```
#define USER_MOTOR_FLUX_EST_FREQ_Hz (60.0)
```

The third consideration is to call a function that overwrites a limitation on the current control loops. This function needs to be called out of the ISR. The name of the function is: CTRL_recalcKpKi().

The fourth consideration is to call a function that calculates what the initial estimated inductance should be for a particular motor based on the R/L information. This function also needs to be called out of the ISR, and the name of this function is: CTRL_calcMax_Ls_qFmt().

The fifth consideration is to call a function that takes the calculated inductance from CTRL_calcMax_Ls_qFmt() function call and initializes the estimated inductance when inductance identification is performed. The new function call needs to be called at the end of the ISR and the name of this function is: CTRL_resetLs_qFmt().

6.9.2.9.2 Identifying Inductance of Salient PMSM Motors

There are several considerations to take into account for low-inductance PMSM motors. Some considerations are regarding the hardware used for motor identification, and some other considerations are related to the configuration of the software.

6.9.2.9.2.1 Hardware Considerations

When identifying low-inductance PMSM motors, it is recommended to have the voltage divider for the voltage feedback as low as possible. For example, if the voltage used to run the motor is 24 V, then the voltage resistor divider should be 26 V, or so. This allows the maximum number of bits of the ADC converter when measuring the voltage feedback.

Once the hardware has been changed, update USER_ADC_FULL_SCALE_VOLTAGE_V definition in user.h.

Usually the low-inductance motors are high-speed motors, so the flux is small. The user needs to update the USER_IQ_FULL_SCALE_VOLTAGE_V to a value that lets the identification process identify the flux of the motor. The following equation can be used to come up with a value for the definition of USER_IQ_FULL_SCALE_VOLTAGE_V:

Minimum Flux that can be identified (V/Hz) = USER_IQ_FULL_SCALE_VOLTAGE_V/ USER_EST_FREQ_Hz/0.7

Copyright © 2013–2019, Texas Instruments Incorporated

6.9.2.9.2.2 Software Considerations

As far as the software configuration is concerned, it is recommended to identify R/L constant with a higher frequency. For the majority of the low-inductance motors tested, a 300-Hz frequency for R/L is enough. Updates to R/L frequency need to be updated in:

```
#define USER_R_OVER_L_EST_FREQ_Hz (300)
```

The second consideration is to have a higher frequency to identify the inductance of the motor. This is part of the motor parameters. The following example is used to identify a motor with a few tens of µH.

```
#define USER_MOTOR_FLUX_EST_FREQ_Hz     (60.0)
```

The third consideration is to call a function that overwrites a limitation on the current control loops. This function needs to be called out of the ISR. The name of the function is: CTRL_recalcKpKi().

The fourth consideration is to call a function that calculates what the initial estimated inductance should be for a particular motor based on the R/L information. This function also needs to be called out of the ISR, and the name of this function is: CTRL_calcMax_Ls_qFmt().

The fifth consideration is to call a function that takes the calculated inductance from CTRL_calcMax_Ls_qFmt() function call, and initializes the estimated inductance when inductance identification is performed. The new function call needs to be called at the end of the ISR, and the name of this function is: CTRL_resetLs_qFmt().

6.9.2.9.3 Identifying Inductance of Salient PMSM Motors

[TBD]

6.9.2.10 Identifying High-Speed PMSM Motors

Adjusting resulting current controller gains for high-speeds. [TBD]

6.9.2.11 Identifying High-Cogging Torque PMSM Motors

For a discussion on this topic, see Section 6.9.2.6.

6.9.3 Troubleshooting ACIM Motor Identification

6.9.3.1 Identifying Low-Voltage and High-Current ACIM Motors

[TBD]

6.9.3.2 Identifying Low-Inductance ACIM Motors

[TBD]

6.9.3.3 Troubleshooting Flux Measurement

Troubleshooting the flux measurement is the same for both motors. For more information, see Section 6.5.3. The default frequency is set to 20 Hz and will also work for most ACIM motors. As with PMSM motors, increase with increments of 10 Hz until the motor does not stop spinning during flux identification.

6.9.3.4 Troubleshooting Locked Rotor Test

Is it very important to lock the rotor completely, avoiding any motion of the shaft. This is the only way the internal variables will produce accurate motor parameters in subsequent states of the estimator state machine.

Copyright © 2013–2019, Texas Instruments Incorporated

Inertia Identification

In classical mechanics, moment of inertia, also called mass moment of inertia, or rotational 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 ridigly coupled with that motor.

There is a common misunderstanding that inertia is equivalent to load. Load usually presents as load inertia and load torque, where 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. Example 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. Load inertia of a system should be estimated with the load torque eliminated or minimized.

Figure 7-1 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.

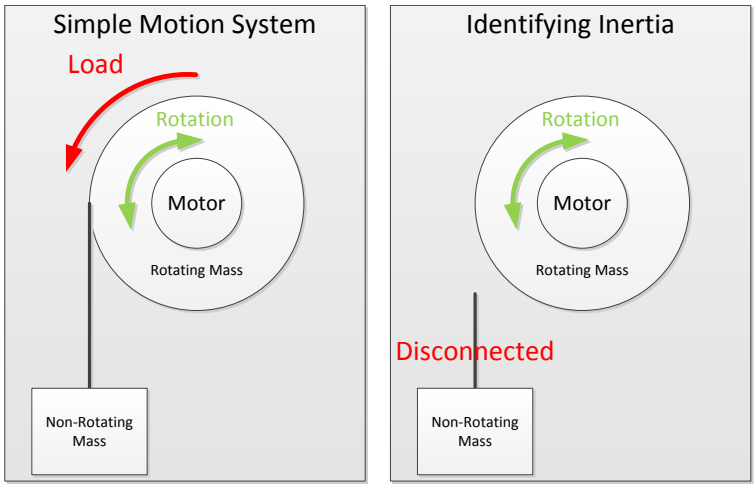


Figure 7-1. Example of Identifying Inertia in a Simple Motion System

Copyright © 2013–2019, Texas Instruments Incorporated

Topic Page

7.1 InstaSPIN-MOTION Inertia Identification 306
7.2 Inertia Identification Process Overview 307
7.3 Software Configuration for SpinTAC Velocity Identify 310
7.4 Troubleshooting Inertia Identification 312
7.5 Difficult Applications for Inertia Identification 313

Copyright © 2013–2019, Texas Instruments Incorporated

7.1 InstaSPIN-MOTION Inertia Identification

Inertia is an important piece of information needed to precisely control the mechanical system. InstaSPIN-MOTION provides a robust inertia identification feature via SpinTAC Velocity Identify that obtains an accurate estimation of inertia, while accounting for the influence of friction within the sensorless application. Currently, SpinTAC Velocity Identify does not actively consider the load torque. In order to get the appropriate value, the load torque, such as gravity for crane-type applications or compressed fluid in compressor applications, needs to be removed or minimized.

SpinTAC Velocity Identify estimates the inertia in the units A / [krpm/s]. This is different from the traditional unit for inertia of Kg * m^2. The unit A / [krpm/s] represents the amount of torque required to accelerate the system. It is proportional to the SI unit Kg * m^2. The relationship is based on the amount of torque that the motor can produce. The SpinTAC controller needs to know how much torque is required to accelerate the system and thus uses this non-traditional unit for inertia.

SpinTAC Velocity Identify produces a very accurate inertia result. Figure 7-2 is a plot of the inertia identification result of the same motor for 100 tests of the inertia identification process. As you can see the inertia identification is extremely repeatable. The maximum and minimum values for these trials are within 0.5% of the average value for 100 tests. Once SpinTAC Velocity Identify is correctly estimating your system inertia it will produce a result with similar levels of repeatability.

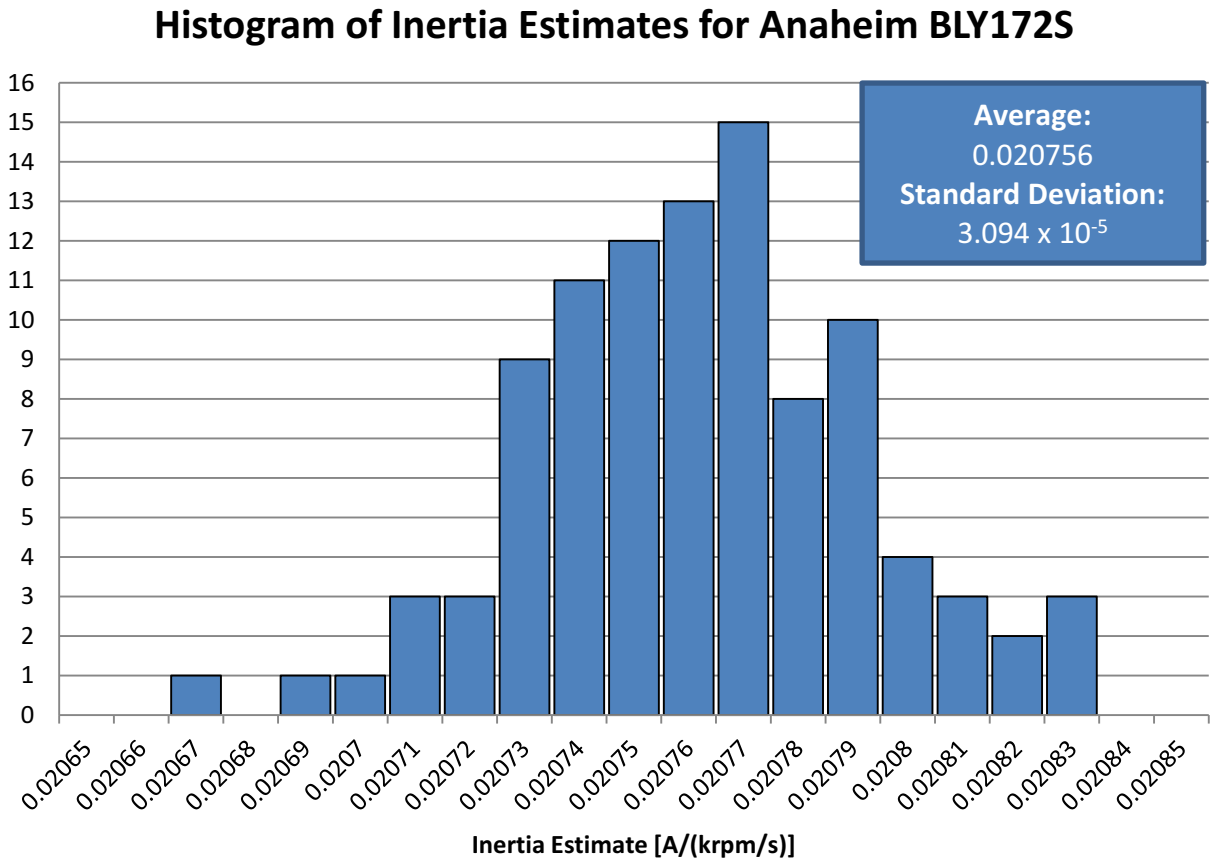


Figure 7-2. Histogram of 100 Inertia Identification Trials

The estimated inertia is an input to both SpinTAC speed and position controllers. However, the SpinTAC controller is extremely robust and can tolerate a wide range of inertia variation. This feature is valuable in applications where the inertia of the system changes over time.

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 7-3 compares the performance of the SpinTAC speed controller with a range of wrong inertia setting. This was tested by applying a torque disturbance to a motor system. The inertia value provided to the SpinTAC speed controller was set to different values to highlight the range of inertia error that can be tolerated by the controller. This shows that the SpinTAC speed controller can tolerate an inertia mismatch of up to eight times. The best performance is realized when the inertia value is match with the application, but if the inertia of the system changes the SpinTAC speed controller remain stable.

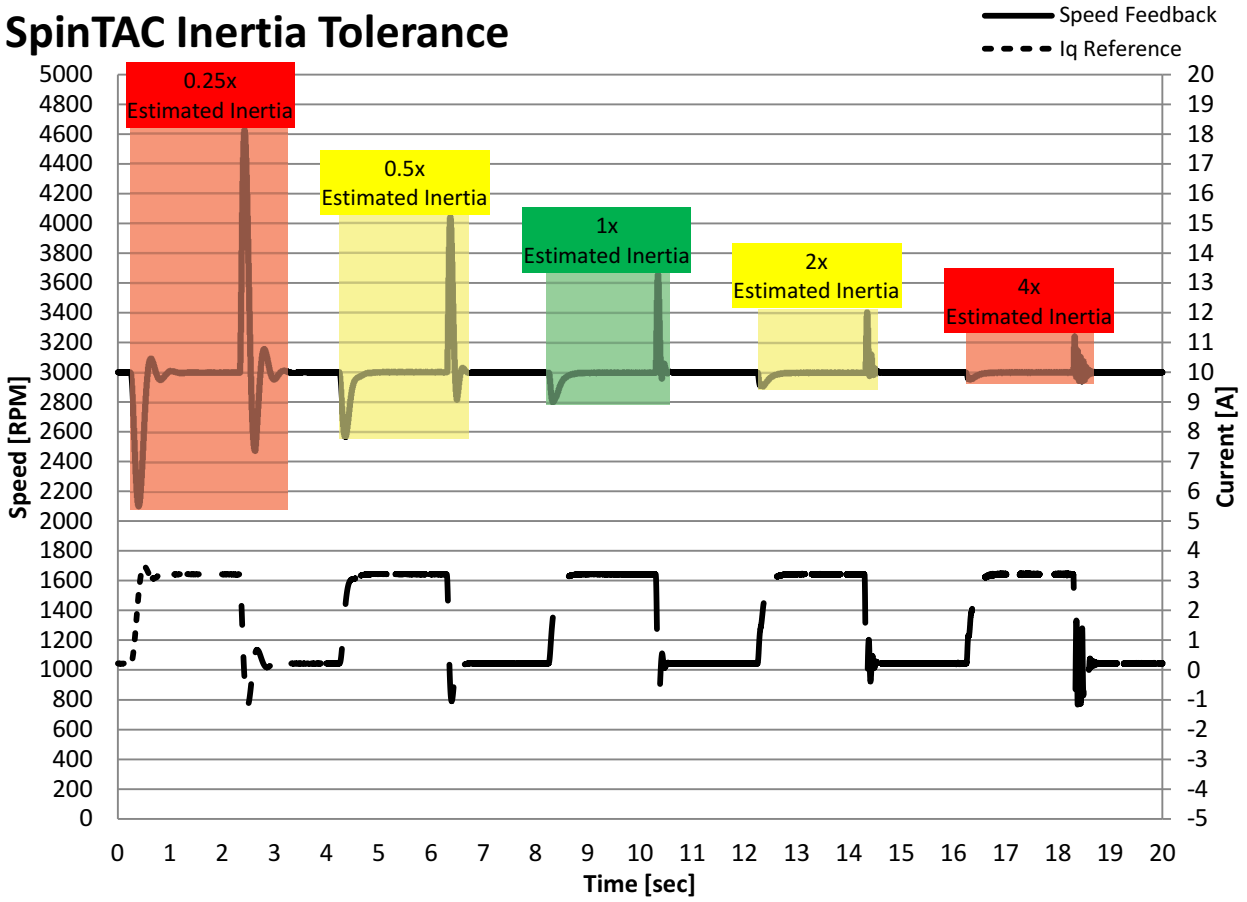


Figure 7-3. SpinTAC Speed Controller Inertia Tolerance

SpinTAC Velocity Identify provides a method to quickly and easily estimate the system inertia. It applies a continuous torque profile to the motor and uses the speed feedback to calculate the motor inertia. This is an open loop test that is designed to run as part of the development process. 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.

During inertia identification, the motor spins in a positive direction and will then spin briefly in a negative direction. If this cannot be accomplished in your system there are special considerations that will need to be taken into account. The description of these considerations in discussed in Section 7.5.

7.2 Inertia Identification Process Overview

The SpinTAC Inertia Identification process is very quick. It needs to accelerate and decelerate the motor in order to build an estimate of the system inertia. Prior to the inertia identification process a couple of conditions need to be satisfied.

- 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 must be disabled.

Copyright © 2013–2019, Texas Instruments Incorporated

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 7-4 is a flowchart that shows the steps required prior to enabling SpinTAC Velocity Identify.

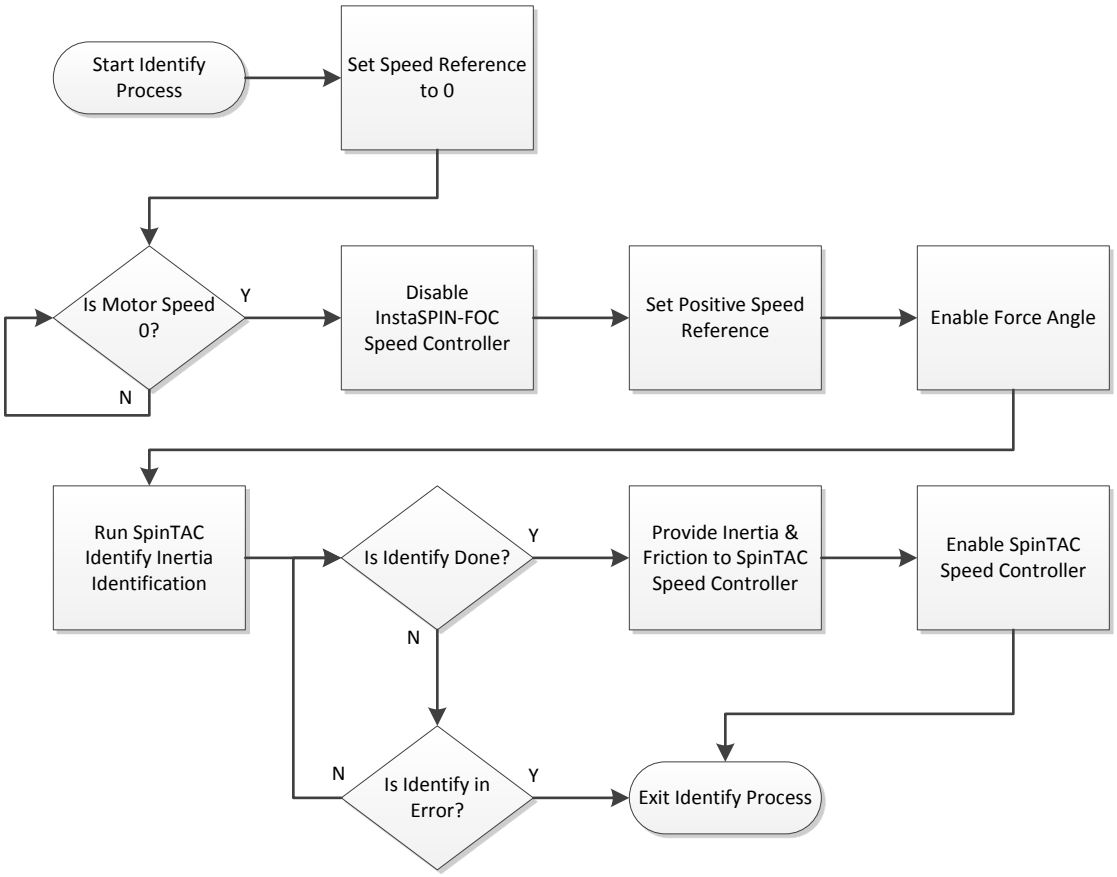


Figure 7-4. Flowchart for SpinTAC Velocity Identify Process

Figure 7-5 is a plot of the continuous torque curve that is applied to the motor. Both positive and negative torque is applied during the SpinTAC Velocity Identify process. Torque is initially applied to the motor in order for the motor rotor to be properly aligned prior to the inertia identification process.

Copyright © 2013–2019, Texas Instruments Incorporated

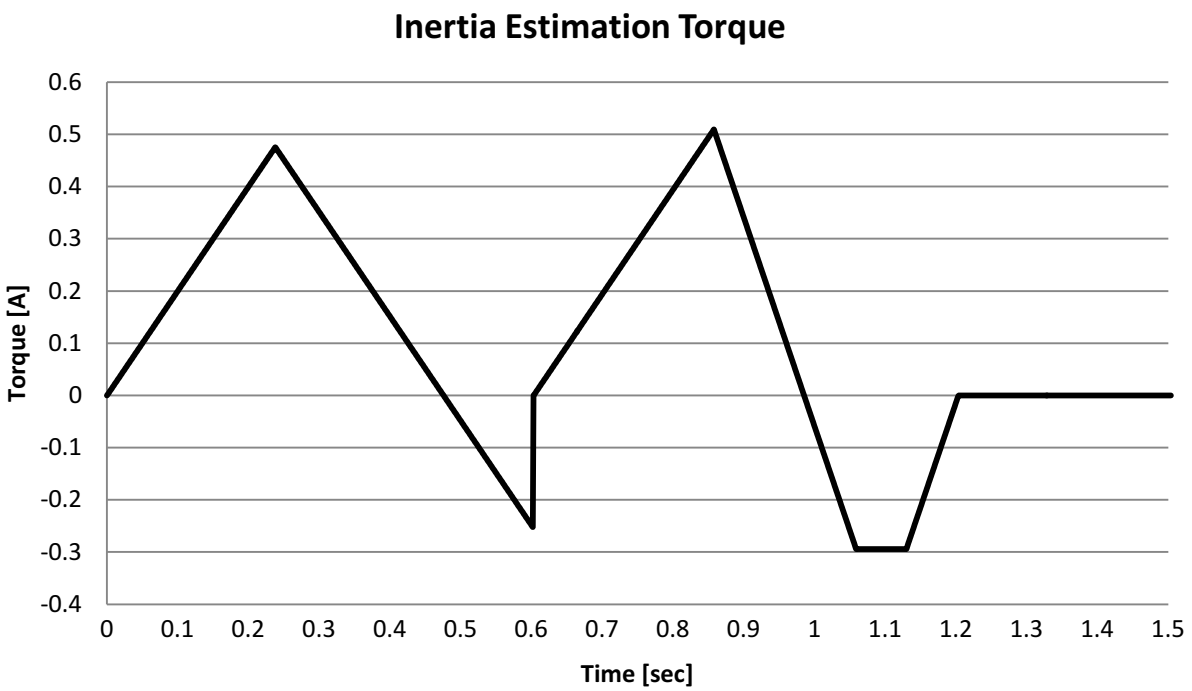


Figure 7-5. SpinTAC Velocity Identify Torque Reference

This results in the motor spinning as described in Figure 14-35. It is important that the motor is spinning continuously during the inertia identification process. If the motor stops during the inertia identification trial, the configuration parameters should be adjusted and the inertia identification process should be repeated. For more information about how to correct for common configuration errors during inertia identification, see Section 7.4.

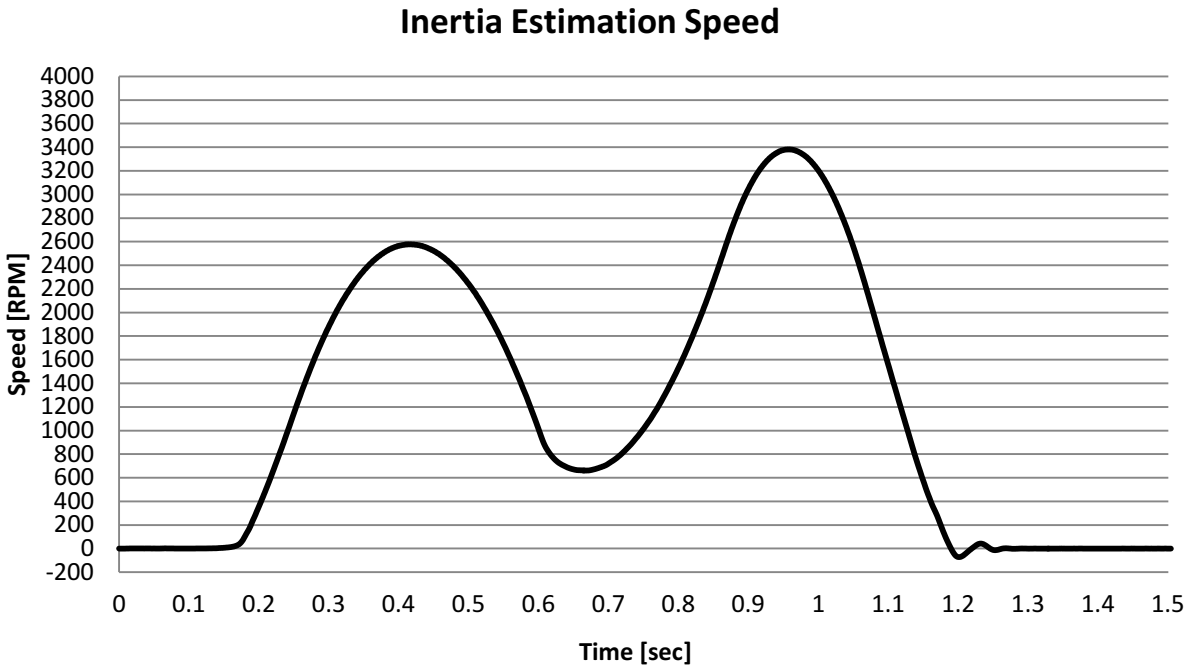


Figure 7-6. SpinTAC Velocity Identify Speed Feedback

This represents the typical case. For more information about how to identify the system inertia for motors with high cogging force or large friction, see Section 7.5.

Copyright © 2013–2019, Texas Instruments Incorporated

7.3 Software Configuration for SpinTAC Velocity Identify

Configuring SpinTAC Velocity Identify requires four steps. Lab 5c — InstaSPIN-MOTION Inertia Identification — is an example project that implements the steps required to use SpinTAC Velocity Identify to estimate the system inertia. The header file spintac_velocity.h, included in MotorWare, allows you to quickly include the SpinTAC components in your project.

7.3.1 Include the Header File

This should be done with the rest of the module header file includes. In the Lab 5c example project, this file is included in the spintac_velocity.h header file. For your project, this step can be completed by including spintac_velocity.h.

```
#include "sw/modules/spintac/src/32b/spintac_vel_id.h"
```

7.3.2 Declare the Global Variables

This should be done with the global variable declarations in the main source file. In the Lab 5c project, this structure is included in the ST_Obj structure that is declared as part of the spintac_velocity.h header file.

```
ST_Obj st_obj;      // The SpinTAC Object
ST_Handle stHandle; // The SpinTAC    Handle
```

If you do not wish to use the ST_Obj structure that is declared in the spintac_velocity.h header file, use the example below.

```
ST_VelId_t stVelId;            // The SpinTAC Velocity Identify object
ST_VELID_Handle stVelIdHandle; // The SpinTAC Inertia Identify handle
```

7.3.3 Initialize the Configuration Variables

This should be done in the main function of the project ahead of the forever loop. This will load all of the default values into SpinTAC Velocity Identify. This step can be completed by running the functions ST_init and ST_setupVelId that are declared in the spintac_velocity.h header file. If you do not wish to use these two functions, the code example below can be used to configure the SpinTAC Velocity Identify component. This configuration of SpinTAC Velocity Identify represents the typical configuration that should work for most motors.

```
// Initialize the SpinTAC Velocity Identify Component
stVelIdHandle = STVELID_init(&stVelId, sizeof(stVelId));

// Setup SpinTAC Velocity Identify Component
// Sample time [s]
STVELID_setSampleTime_sec(stVelIdHandle, _IQ(ST_SPEED_SAMPLE_TIME));
// System speed limit [pu/s], (0, 1]
STVELID_setVelocityMax(stVelIdHandle, _IQ(1.0));
// System maximum (0,1] & minimum [-1,0) control signal [PU]
STVELID_setOutputMaximums(stVelIdHandle, maxCurrent_PU, -maxCurrent_PU);
// Goal Speed of the inertia identification process [pu/s], (0, 1]
STVELID_setVelocityPositive(stVelIdHandle, _IQmpy(_IQ(0.4), _IQ(1.0)));
// System control signal high (0, OutMax] & low [OutMin, 0) limit [PU]
STVELID_setOutputLimits(stVelIdHandle, maxCurrent_PU, -maxCurrent_PU);
// Low pass filter constant to smooth the speed feedback signal [tick], [1, 100]
STVELID_setLowPassFilterTime_tick(stVelIdHandle, 1);
// Configure the time out for inertia identification process [s], [100*T, 10.0]
STVELID_setTimeOut_sec(stVelIdHandle, _IQ(10.0));// Rate at which torque is applied to the
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
motor [s], [T, 25.0]
STVELID_setTorqueRampTime_sec(stVelIdHandle, _IQ(5.0));
// Initially ST_VelId is not in reset
STVELID_setReset(stVelIdHandle, false);
// Initially ST_VelId is not enabled
STVELID_setEnable(stVelIdHandle, false);
```

7.3.4 Call SpinTAC Velocity Identify

This should be done in the main ISR. This function needs to be called at the proper decimation rate for this component. The decimation rate is established by ST_ISR_TICKS_PER_SPINTAC_TICK declared in the spintac_velocity.h header file; for more information, see Section 4.7.1.4. Before calling the SpinTAC Velocity Identify function the speed feedback must be updated. It is also important to notice that this example implements the flowchart from Figure 7-4 in order to make sure the system is ready to identify inertia.

```
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle; // Get pointer to CTRL object
_iq speedFeedback = EST_getFm_pu(obj->estHandle); // Get the mechanical speed in pu/s
_iq iqReference = 0;
if(gMotorVars.SpinTAC.VelIdRun != false) {
  // if beginning the SpinTAC Velocity Identify process
  // set the speed reference to zero
  gMotorVars.SpeedRef_krpm = 0;
  // wait until the actual speed is zero
  if((_IQabs(speedFeedback) < _IQ(ST_MIN_ID_SPEED_PU))
     && (STVELID_getEnable(stVelIdHandle) == false)) {
     gMotorVars.Flag_enableForceAngle = true;
     EST_setFlag_enableForceAngle(obj->estHandle, gMotorVars.Flag_enableForceAngle);
     // set the GoalSpeed
     STVELID_setVelocityPositive(stVelIdHandle, gMotorVars.VelIdGoalSpeed);
     // set the Torque Ramp Time
     STVELID_setTorqueRampTime_sec(stVelIdHandle, gMotorVars.VelIdTorqueRampTime);
     // Enable SpinTAC Velocity Identify
     STVELID_setEnable(stVelIdHandle, true);
     // Set a positive speed reference to FAST to provide direction information
     gMotorVars.SpeedRef_krpm = _IQ(0.001);
     CTRL_setSpd_ref_krpm(ctrlHandle, gMotorVars.SpeedRef_krpm);
  }
}

// Run SpinTAC Velocity Identify
STVELID_setVelocityFeedback(stVelIdHandle, speedFeedback);
STVELID_run(stVelIdHandle);

if(STVELID_getDone(stVelIdHandle) != false) {
  // If inertia identification is successful
  // update the inertia setting of SpinTAC Velocity Controller
  // EXAMPLE:
  // STVELCTL_setInertia(stVelCtlHandle, STVELID_getInertiaEstimate(stVelIdHandle));
  gMotorVars.VelIdRun = false;
  // return the speed reference to zero
  gMotorVars.SpeedRef_krpm = _IQ(0.0);
  CTRL_setSpd_ref_krpm(ctrlHandle, gMotorVars.SpeedRef_krpm);
}
else if((STVELID_getErrorID(stVelIdHandle) != false)
        && (STVELID_getErrorID(stVelIdHandle) != ST_ID_INCOMPLETE_ERROR)) {
  // if not done & in error, wait until speed is less than 1RPM to exit
  if(_IQabs(speedFeedback) < _IQ(ST_MIN_ID_SPEED_PU)) {
    gMotorVars.VelIdRun = false;
    // return the speed reference to zero
    gMotorVars.SpeedRef_krpm = _IQ(0.0);
    CTRL_setSpd_ref_krpm(ctrlHandle, gMotorVars.SpeedRef_krpm);
  }
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
}

// Set the Iq reference that came out of SpinTAC Identify
iqReference = STVELID_getTorqueReference(stVelIdHandle);
CTRL_setIq_ref_pu(ctrlHandle, iqReference);
```

7.4 Troubleshooting Inertia Identification

SpinTAC Velocity Identify has been tested on a wide variety of motors. Non-typical motors can have difficulty with the inertia identification process and result in an error. This error is represented by a value in the ERR_ID field of the SpinTAC Velocity Identify global structure. Common errors and correcting for them is discussed below.

7.4.1 ERR_ID

ERR_ID provides an error code for users. A list of errors defined for SpinTAC Velocity Identify and the solutions for these errors are shown in Table 7-1.

Table 7-1. SpinTAC Velocity Identify Error Code

ERR_ID	Problem	Solution
1	Invalid sample time value	Set cfg.T_sec within (0, 0.01]
2	Invalid system maximum velocity value	Set cfg.VelMax within (0, 1]
4	Invalid velocity loop control signal maximum value	Set cfg.OutMax within (0, 1]
5	Invalid velocity loop control signal minimum value	Set cfg.OutMin within [-1, 0)
22	Invalid velocity value	Set cfg.VelPos within (0, cfg.VelMax]
23	Invalid velocity loop control signal positive value	Set cfg.OutPos within (0, cfg.OutMax]
24	Invalid velocity loop control signal negative value	Set cfg.OutNeg within [cfg.OutMin, 0)
34	Invalid acceleration ramp time value	Set cfg.RampTime_sec within [cfg.T_sec, 25]
36	Invalid value for feedback type	Set cfg.Sensorless within {false, true}
1010	Invalid velocity feedback low pass filter time constant	Set cfg.LptTime_tick within [1, 100]
1011	Invalid time out value	Set cfg.TimeOut_sec within [1, 10]
2003	Invalid inertia estimate value	Adjust the configuration parameters and repeat
2004	Inertia identification process timed out	Adjust the configuration parameters and repeat
2005	Identification process is discarded by setting RES = 1 or ENB = 0	No action
2006	Motor stopped during identification process	Adjust the configuration parameters and repeat
4001	Invalid SpinTAC license	Use the chip with valid license
4003	Invalid ROM Version	Use a chip with a valid ROM version or use the SpinTAC library that is compatible with the current ROM version.

7.4.2 2003 Error

This error indicates that the estimated inertia value is incorrect. This is commonly caused by motors that have a large friction or a high cogging force. To correct for this error, decrease the RampTime_sec parameter in the configuration section of the global structure. Decreasing this parameter will increase the rate at which torque is applied to the motor during the inertia identification process.

If the identified inertia is a valid number and the friction coefficient is a very small negative number, this error might be caused by the precision of the calculation when the friction coefficient is very small. In such a case, the identified inertia may still be valid.

Copyright © 2013–2019, Texas Instruments Incorporated

7.4.3 2004 Error

This error represents that the inertia identification process timed out prior to completion. There are a couple of different causes for this event.

7.4.3.1 Motor Spins Continuously

If the motor spins continuously and the inertia identification process results in this error it means that the goal speed of the identification process is too high. This is commonly caused by the motor having a low rated speed. The way to correct this error is to decrease the VelPos parameter in the configuration section of the global structure. This parameter represents the goal speed of the identification process. If the VelPos value is set too low it can result in inaccurate inertia identification.

7.4.3.2 Motor Does Not Spin Initially

If the motor does not spin initially and the inertia identification process results in this error it means that the torque applied to the motor was too low. This is commonly caused by the motor requiring a large amount of current to begin motion. The way to correct for this error is to increase the OutPos parameter in the configuration section of the global structure. This parameter represents the maximum torque that is applied during the identification process.

7.4.4 2006 Error

This error indicates that the motor failed to spin continuously during the inertia identification process. It is important that the motor does not stop spinning until the inertia identification process is complete. The error is commonly caused by the RampTime_sec parameter being set too high. When the motor stops during the inertia identification process it can cause the estimated inertia value to be larger than the actual inertia value. Decreasing the RampTime_sec parameter will increase the rate at which torque is applied to the motor during the inertia identification process.

7.5 Difficult Applications for Inertia Identification

Some applications have features that make it difficult to identify the system inertia. The default configuration of SpinTAC Velocity Identify is designed to work with applications that use typical motors. Applications where the motor feature any of the following conditions, some changes need to be made to the SpinTAC Velocity Identify configuration.

- Large Cogging Force
- Large Friction
- Low-Rated Speed
- Large Back EMF
- Large Start-Up Current

7.5.1 Automotive Pumps (Large-Cogging Force / Large Friction)

Many automotive pumps feature a large amount of cogging force or have a very large friction. For these applications it is important to decrease the RampTime_sec parameter. This parameter is located in the configuration structure of the SpinTAC Velocity Identify global structure. The RampTime_sec value represents the amount of time in seconds it takes to ramp the Iq reference from 0 to 1.0 PU. Decreasing this value means that Iq reference will increase more quickly during inertia identification. This ensures that the motor decelerates properly. If the motor does not properly decelerate, the inertia identification process will produce a bad result.

Figure 7-7 is the speed feedback during the inertia identification process for an automotive pump with large friction. When RampTime_sec is set to 10.0 the inertia identification process does not complete successfully. It is important to notice that the larger RampTime_sec has a delayed start to the process and the motor stops in the middle of the test. Both of these conditions cause the inertia identification to fail with the ERR_ID set to 2003. When RampTime_sec is set to 3.0, the inertia identification does not have a delayed start, and the motor does not stop during the test. Both of these conditions need to be satisfied in order for the inertia identification to complete successfully.

Copyright © 2013–2019, Texas Instruments Incorporated

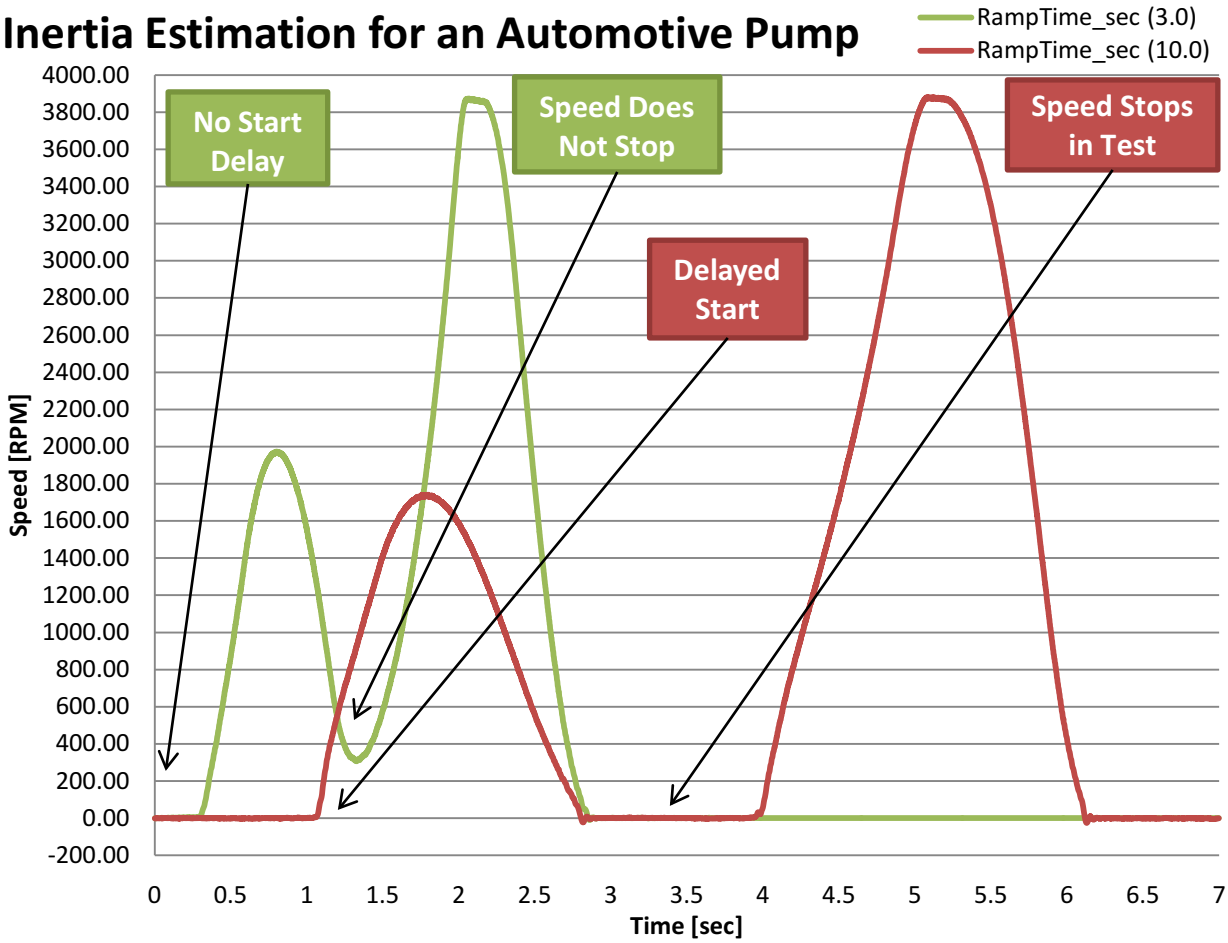


Figure 7-7. Speed Feedback for Inertia Identification for an Automotive Pump

7.5.2 Direct Drive Washing Machines (Low-Rated Speed and Large Back EMF)

Most direct drive washing machine use motors with a low rated speed. In these applications it is important to decrease the VelPos parameter. This parameter is located in the configuration structure of the SpinTAC Velocity Identify global structure. The VelPos value represents the goal speed in pu/s of the inertia identification process. If the goal speed is greater than twice the rated speed of the motor, inertia identification will fail because the motor will not be able to achieve enough speed. Decreasing VelPos means that the goal speed of the inertia identification process will be lower and will allow the process to succeed. Using field weakening to increase the speed of the motor during the inertia identification process is not recommended. Field weakening will impact the relationship between speed and torque.

Figure 7-8 is the speed feedback during the inertia identification process for a direct drive washing machine. You should notice that the motor spins at its rated speed of approximately 360 rpm for longer than 5 seconds. This means that the motor never reached the goal speed specified in the VelPos configuration parameter and SpinTAC Velocity Identify timed out and ended the inertia identification process.

Copyright © 2013–2019, Texas Instruments Incorporated

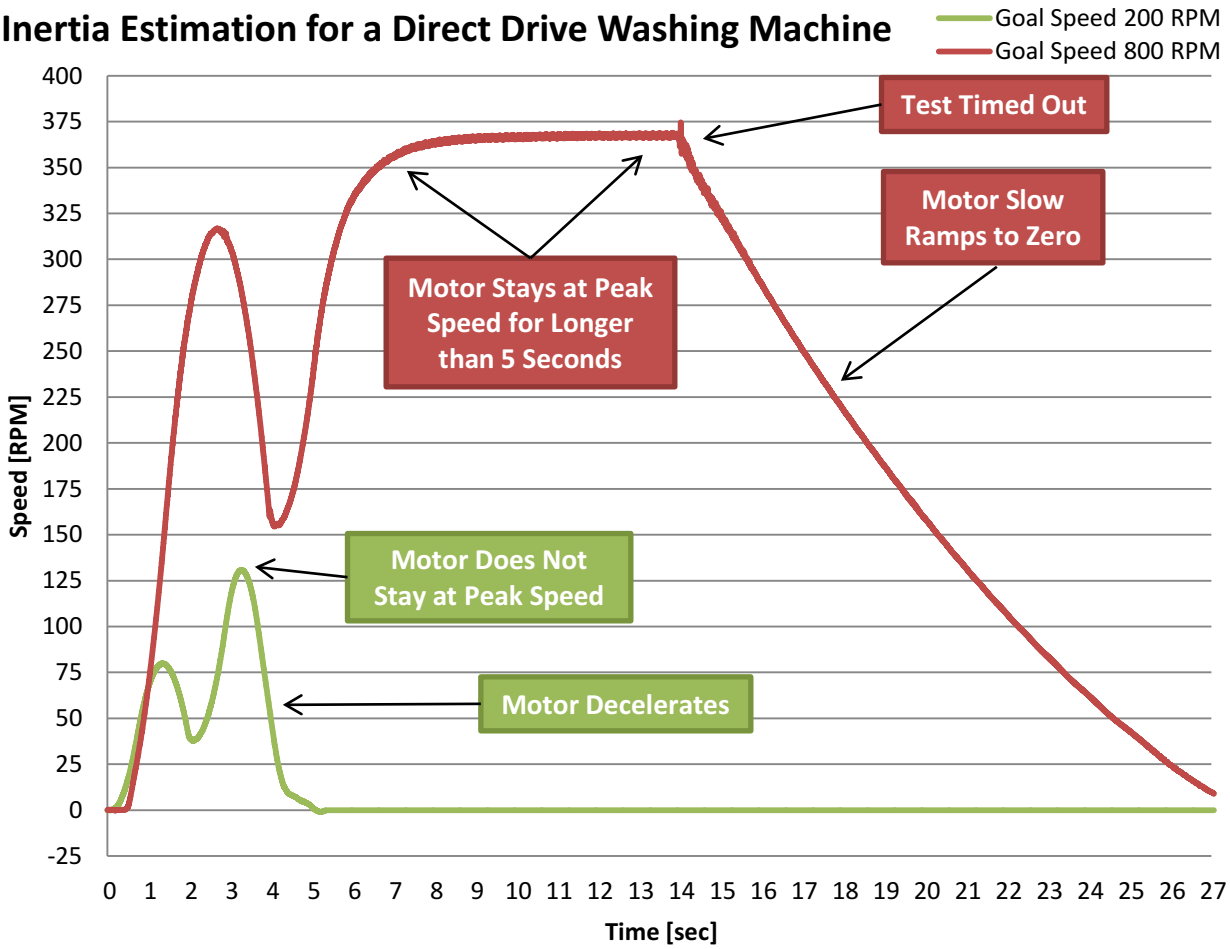


Figure 7-8. Speed Feedback for Inertia Identification for a Direct Drive Washing Machine

Direct drive washing machines also feature a large amount of back emf and rapid deceleration could cause an over-voltage condition on the DC bus. Reducing the Goal Speed will cause the motor to have less deceleration and will generate less voltage on the DC bus. The Goal Speed is set by the VelPos parameter.

Figure 7-9 is a plot that shows the voltage on the DC bus during the inertia identification process. The speed feedback is provided for reference. You should notice that the DC bus rises to 400 volts during the inertia identification process when the Goal Speed is set to 400 RPM. In order to eliminate this large rise in voltage, the VelPos configuration parameter is decreased. When the Goal Speed is set to 200 RPM, the DC bus stays below 350 volts during the entire inertia identification process.

Copyright © 2013–2019, Texas Instruments Incorporated

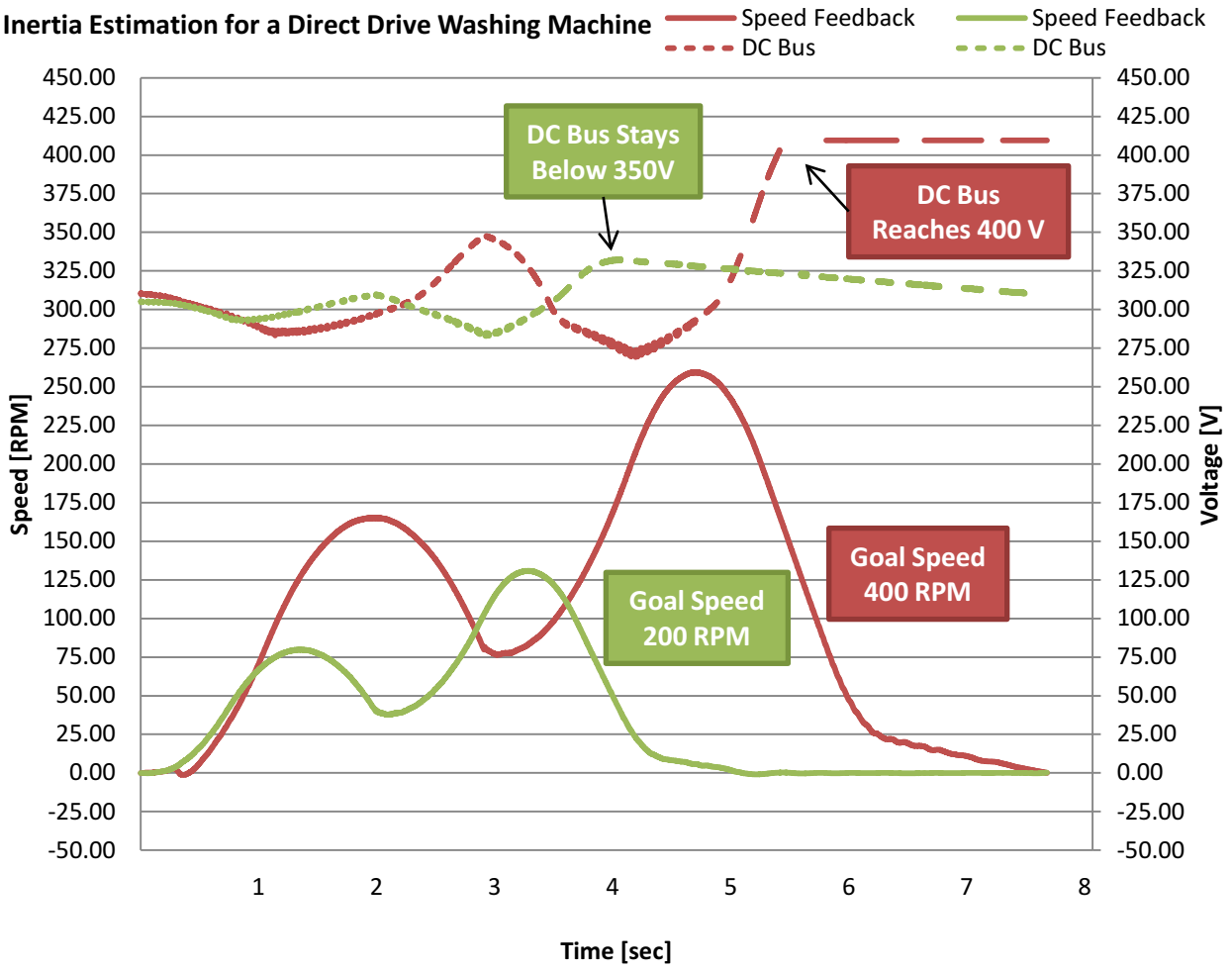


Figure 7-9. DC Bus Voltage for Inertia Identification for a Direct Drive Washing Machine

7.5.3 Compressors (Large Start-Up Current)

Compressors often cannot have the load torque completely eliminated. This causes the motor to require a large amount of Iq reference to begin spinning. For these applications it is important to increase the PosOut parameter. This parameter is located in the configuration structure of the SpinTAC Velocity Identify global structure. The PosOut value represents the amount of Iq reference, in PU, that will be applied as part of the inertia identification process. Increasing this value will apply more current to the motor during the process. It might also be important to decrease the RampTime_sec parameter. This will increase the rate at which the Iq reference is applied to the system.

Figure 7-10 is the speed feedback during the inertia identification process for a compressor. You should notice that the speed does not accelerate up to the goal speed very quickly. It takes a long time for it to build up to the goal speed. This indicates that the motor needs additional torque in order to reach the goal speed. The configuration parameter that needs to be adjusted is PosOut. This parameter should be increased in order to supply more torque to the motor during the inertia identification process. The inertia identification process will only use as much torque as is required to reach the goal speed. It is better to have the PosOut parameter be larger than what is required.

Copyright © 2013–2019, Texas Instruments Incorporated

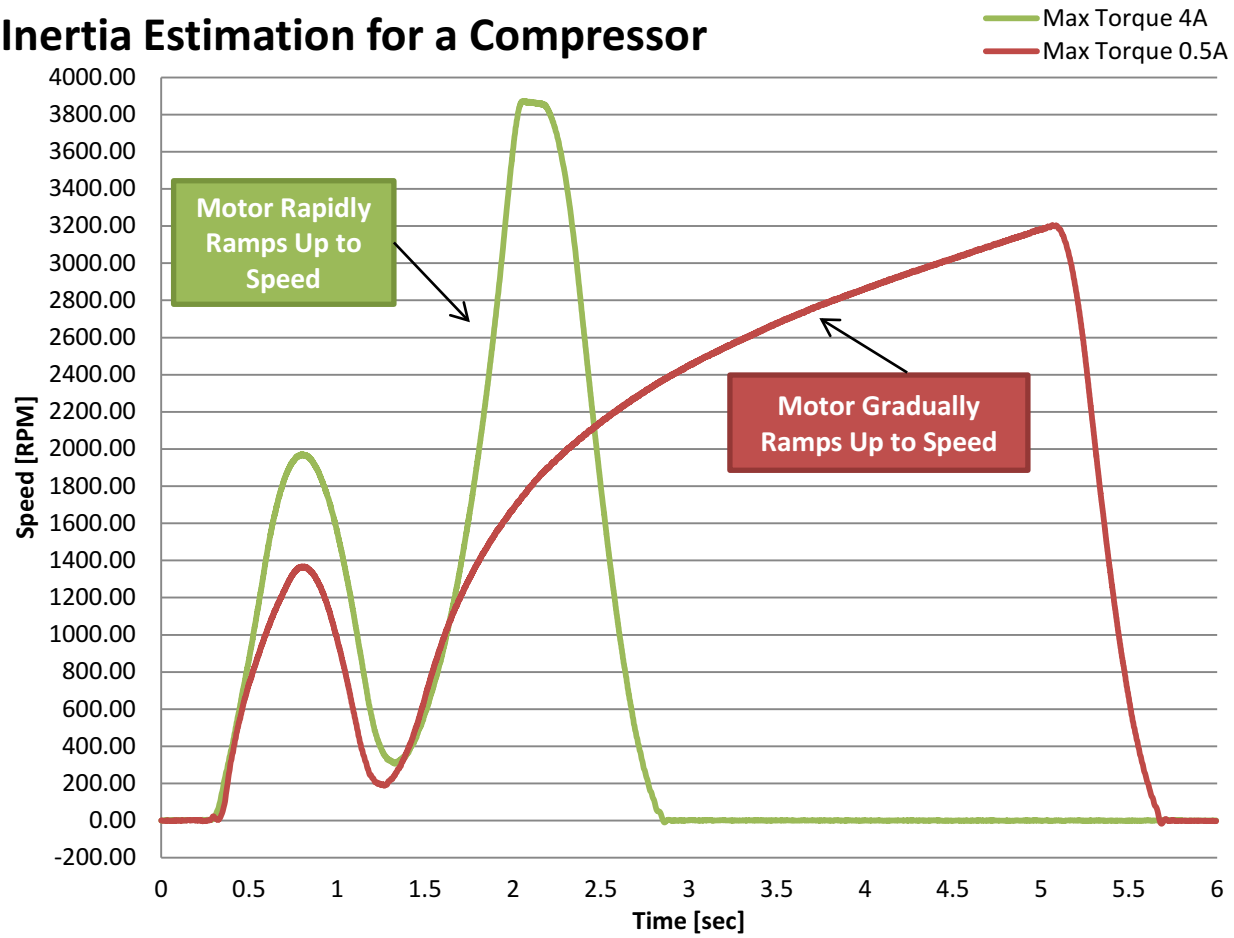


Figure 7-10. Speed Feedback for Inertia Identification for a Compressor

Many compressors cannot run in the negative direction. The inertia identification process applies negative Iq reference to the system in order to decelerate the motor. Even though negative Iq reference is being applied, the motor still does not end up rotating in the negative direction due to the inertia identification process ending before the motor would spin in the negative direction.

Copyright © 2013–2019, Texas Instruments Incorporated

MCU Considerations

In this section we will review the MCU-specific considerations required to successfully implement InstaSPIN-FOC and InstaSPIN-MOTION:

- List of devices that are enabled with InstaSPIN and describing their specific requirements
- Memory map considerations
- Clock rates

  This section provides details the microcontroller resources required by the InstaSPIN libraries. Two implementations of InstaSPIN are discussed in this document depending on how much of the functionality is run from secured ROM, or what functionality is run from user's memory, either RAM or FLASH:

  – All of InstaSPIN-FOC from ROM, also known as full implementation.
  – Only FAST™ from ROM, also known as minimum implementation.

  Also, a distinction must be made depending on where the code is placed and executed from in user's memory. Two categories are discussed:

  – Library executing from ROM and loading and executing user's code from RAM
  – Library executing from ROM and loading and executing user's code from FLASH
- Specifically for the library implementation and where the code is loaded and executed from, the following resources categories are discussed in this document:
  – CPU Utilization
  – Memory Allocation
  – Stack Utilization
- A common section at the end lists the Digital and Analog Pins Utilization, which is common to every mode of operation of InstaSPIN.

Topic **Page**

8.1 InstaSPIN-Enabled Devices **319**
8.2 ROM and User Memory Overview **320**
8.3 Details on CPU Load and Memory Footprint Measurements **324**
8.4 Memory Footprint **328**
8.5 CPU Load **333**
8.6 Digital and Analog Pins **353**

Copyright © 2013–2019, Texas Instruments Incorporated

8.1 InstaSPIN-Enabled Devices

The devices that currently include InstaSPIN-FOC or InstaSPIN-MOTION in ROM are shown in Table 8-1.

Table 8-1. InstaSPIN-Enabled Devices

Device	InstaSPIN-FOC in ROM	InstaSPIN-MOTION in ROM	InstaSPIN TRM	Device Datasheet	Device TRM
TMS320F28069M	V1.6	V2.1.8	SPRUHJ0	SPRS698	SPRUH18
TMS320F28069F	V1.6	not included	SPRUHI9	SPRS698	SPRUH18
TMS320F28068M	V1.6	V2.1.8	SPRUHJ0	SPRS698	SPRUH18
TMS320F28068F	V1.6	not included	SPRUHI9	SPRS698	SPRUH18
TMS320F28062F	V1.6	not included	SPRUHI9	SPRS698	SPRUH18
TMS320F28054M	V1.7	V2.1.8	SPRUHW1	SPRS797	SPRUHE5
TMS320F28054F	V1.7	not included	SPRUHW0	SPRS797	SPRUHE5
TMS320F28052M	V1.7	V2.1.8	SPRUHW1	SPRS797	SPRUHE5
TMS320F28052F	V1.7	not included	SPRUHW0	SPRS797	SPRUHE5
TMS320F28027F	V1.7	not included	SPRUHP4	SPRS523	SPRUFN3
TMS320F28026F	V1.7	not included	SPRUHP4	SPRS523	SPRUFN3

The devices have remained exactly the same except that InstaSPIN technology has been added to a specific region of ROM. For detailed information on the device, see the device-specific data sheets and errata for complete details on the device that you are using.

8.1.1 softwareUpdate1p6() - Function is Required in User Code

The function softwareUpdate1p6() is a work-around for a bug in InstaSPIN-FOC v1.6 to correct how inductance is converted from Henries to per unit value when using the inductance from user.h. This function needs to be called whenever motor parameters are loaded from user.h when using InstaSPIN-FOC v1.6.

The following fixes are in this patch:

- Added a maximum per inductance value. Thus, we wanted the per unit inductance values scaled with respect to this maximum value. This would impact the Q format of the inductance value as well.
- Set the current controller gain values (Id/Iq current controllers) based on these new per unit inductance values.

Below is the source code to the patch, it is used in every InstaSPIN-FOC v1.6 and InstaSPIN-MOTION lab example.

```
void softwareUpdate1p6(CTRL_Handle handle)
{
 CTRL_Obj *obj = (CTRL_Obj *)handle;

 float_t fullScaleInductance = EST_getFullScaleInductance(obj->estHandle);
 float_t Ls_coarse_max = _IQ30toF(EST_getLs_coarse_max_pu(obj->estHandle));
 int_least8_t lShift = ceil(log(obj-
>motorParams.Ls_d/(Ls_coarse_max*fullScaleInductance))/log(2.0));
 uint_least8_t Ls_qFmt = 30 - lShift;
 float_t L_max = fullScaleInductance * pow(2.0,lShift);
 _iq Ls_d_pu = _IQ30(obj->motorParams.Ls_d / L_max);
 _iq Ls_q_pu = _IQ30(obj->motorParams.Ls_q / L_max);

 float_t RoverL = obj->motorParams.Rs/obj->motorParams.Ls_d;
 float_t fullScaleCurrent = EST_getFullScaleCurrent(obj->estHandle);
 float_t fullScaleVoltage = EST_getFullScaleVoltage(obj->estHandle);
 float_t ctrlPeriod_sec = CTRL_getCtrlPeriod_sec(ctrlHandle);
 _iq Kp = _IQ((0.25*obj->motorParams.Ls_d*fullScaleCurrent)/(ctrlPeriod_sec*fullScaleVoltage));
 _iq Ki = _IQ(RoverL*ctrlPeriod_sec);
 _iq Kd = _IQ(0.0);
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
 // store the results
 EST_setLs_d_pu(obj->estHandle,Ls_d_pu);
 EST_setLs_q_pu(obj->estHandle,Ls_q_pu);
 EST_setLs_qFmt(obj->estHandle,Ls_qFmt);

// set the Id controller gains
 PID_setKi(obj->pidHandle_Id,Ki);
 CTRL_setGains(ctrlHandle,CTRL_Type_PID_Id,Kp,Ki,Kd);

// set the Iq controller gains
 PID_setKi(obj->pidHandle_Iq,Ki);
 CTRL_setGains(ctrlHandle,CTRL_Type_PID_Iq,Kp,Ki,Kd);

 return;
} // end of softwareUpdate1p6() function
```

8.2 ROM and User Memory Overview

8.2.1 InstaSPIN-FOC Full Implementation in ROM

When application requirements allow to run all of the field oriented control (FOC) blocks from ROM, and no additional functionality is required (that is, a specialized current control algorithm, or Clarke transform, and so on), a full implementation is recommended. This implementation will make use of the entire library contents placed in ROM, and will execute the complete suite of functions and blocks, known as InstaSPIN-FOC. Full implementation not only includes the FAST algorithm, but it also contains the rest of the FOC blocks. The following block diagram shows how the full implementation contains several blocks allowing the entire FOC code to run from ROM, freeing up more memory resources, and taking advantage of the 0-wait state execution from ROM. The ROM is also execute-only ROM, providing an additional level of security, since the memory cannot be written or read, only executed.

For F2802xF devices, some of the functional blocks are loaded in user memory due to reduced ROM size.

Copyright © 2013–2019, Texas Instruments Incorporated

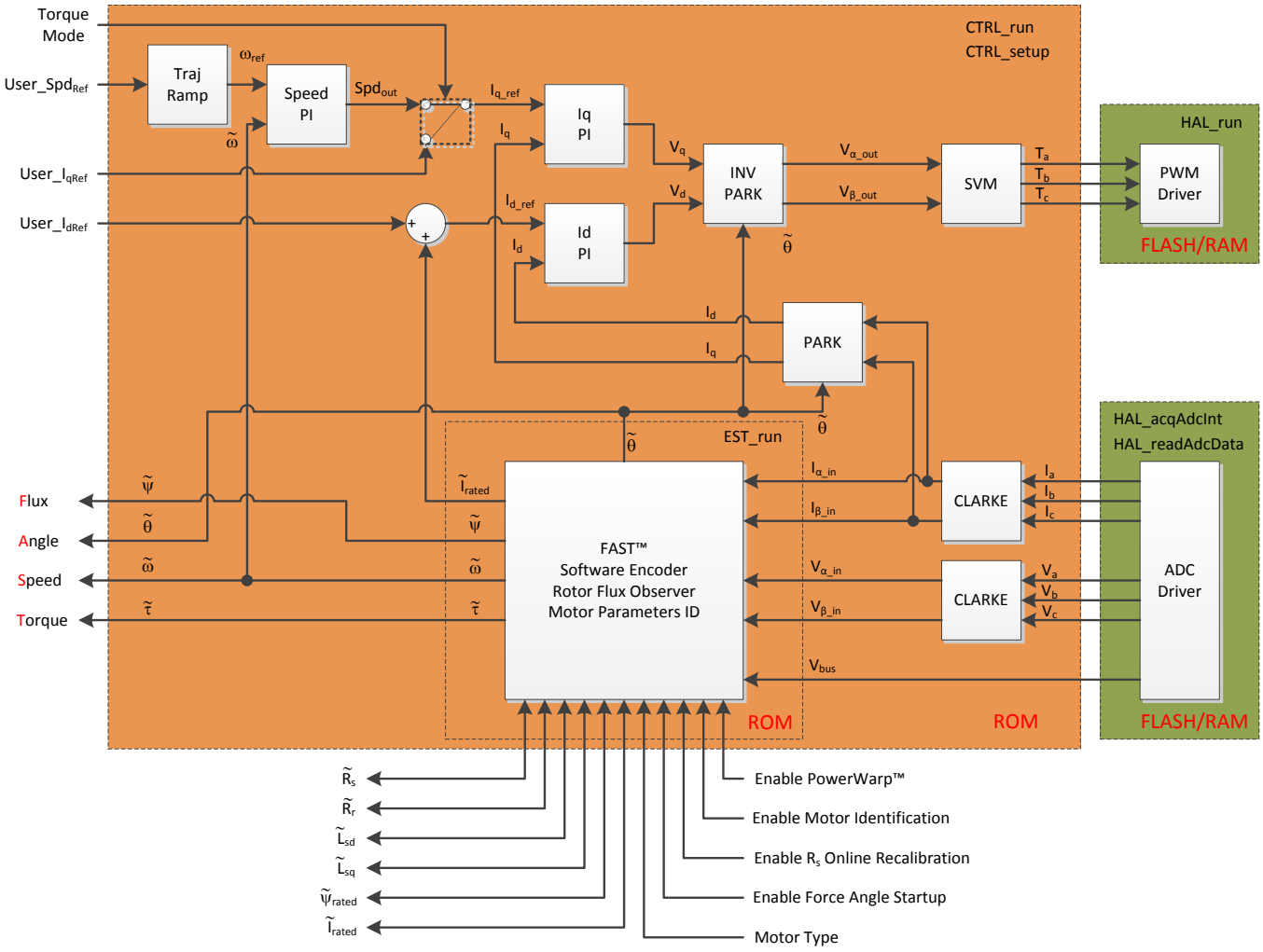


Figure 8-1. InstaSPIN-FOC Full Implementation in ROM

8.2.1.1 Executing from ROM and RAM

Even though the entire InstaSPIN library is executed from ROM, there are a few functions that need to be loaded and run from users' memory. These functions are the interface from the library to the hardware peripherals, as shown in Figure 8-1. All functions related to the driver object (HAL_Obj) interface to the hardware and need to be placed in user's memory. The performance data will depend on where these user's functions are implemented. This section shows performance data when all users' functions are placed and run out of RAM.

From a CPU performance standpoint, loading and executing users' functions from RAM presents the biggest advantage since RAM does not require wait states. On the other hand, loading users' functions to RAM consumes volatile memory space, so users would have to consider total available RAM for variables. The stack utilization and pins used by the library is independent of where the users' code resides, RAM or FLASH.

8.2.1.2 Executing from ROM and FLASH

Loading users' functions to FLASH helps on the RAM consumption aspect, although a portion of the available RAM is still needed for variables and stack. Another consideration when loading users' code from FLASH is the CPU execution time, since FLASH requires wait states.

Due to reduced ROM size for the F2802xF devices, it is impossible to execute InstaSPIN-FOC fully from ROM. For details on running the minimum implementation of InstaSPIN-FOC, see Section 8.2.2.

Copyright © 2013–2019, Texas Instruments Incorporated

Stack usage as well as pins used is the same as loading users' code to RAM. We are still listing those parameters here to provide a complete list of resources usage for a particular implementation.

8.2.2 InstaSPIN-FOC Minimum Implementation in ROM

Some applications require more control of what the field oriented control is doing. Applications with this requirement can use the minimum implementation of InstaSPIN, which consists in running only the FAST estimator from ROM with any of the other software blocks moveable to user memory. The estimator must remain in ROM since the source code is proprietary to TI.

Figure 8-2 highlights in different colors what runs from ROM and users' memory.

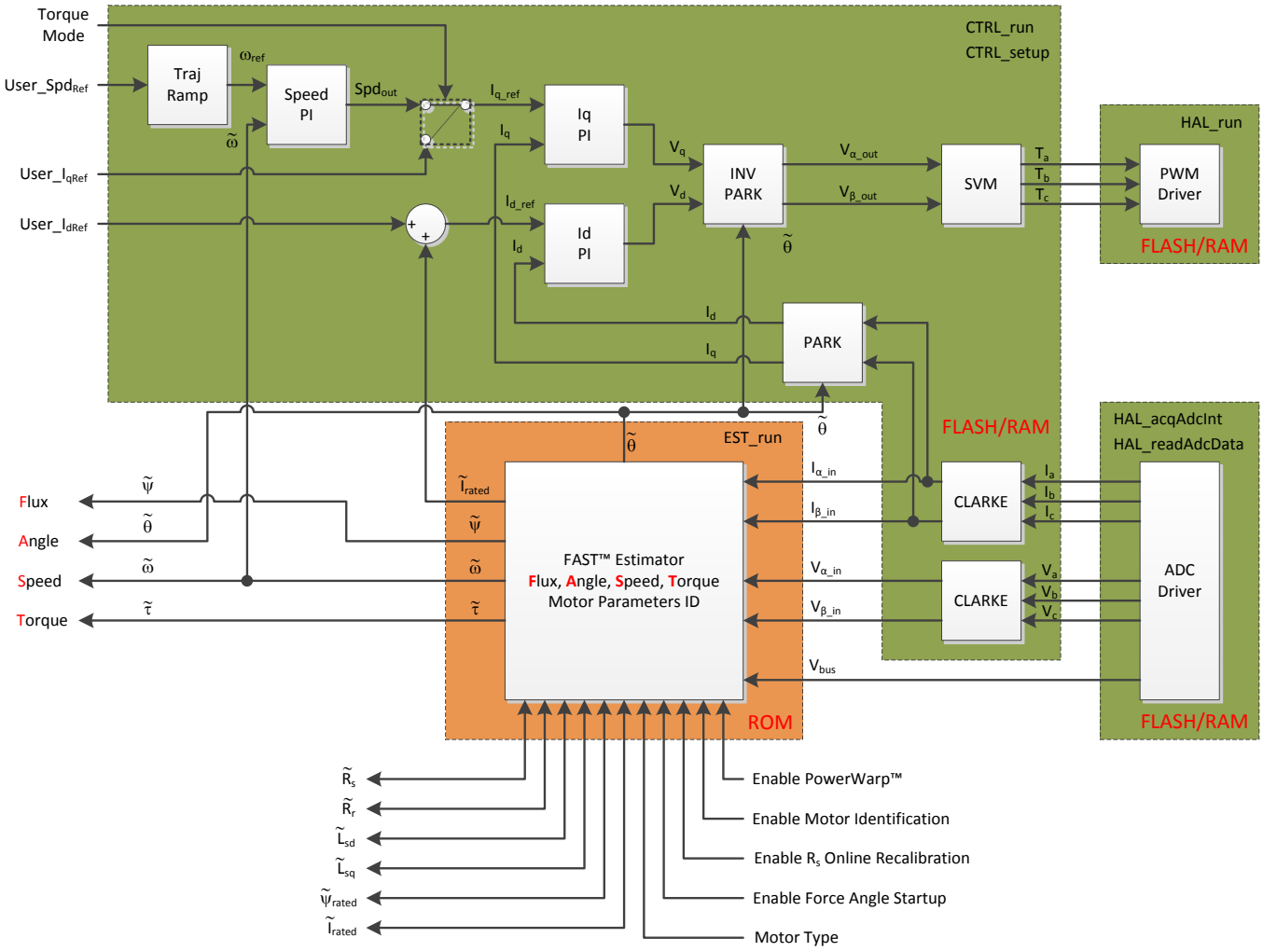


Figure 8-2. InstaSPIN-FOC Minimum Implementation in ROM

Notice that only the function that runs the estimator (Est_run) is executed from ROM. Everything else is executed from users' memory, either RAM or FLASH. In the following subsections, the performance of InstaSPIN is described when a minimum implementation is used.

8.2.2.1 Executing from ROM and RAM

When the users' functions are loaded and executed from RAM, code executes faster than from FLASH, with the penalty of using a portion of the available RAM for code. If a particular application requires maximum execution speed, and the available RAM satisfies non volatile memory requirements, this is the best option.

Copyright © 2013–2019, Texas Instruments Incorporated

8.2.2.2 Executing from ROM and FLASH

As mentioned in previous sections, loading users' functions to FLASH helps on the RAM consumption aspect. However, the disadvantage of FLASH execution is the speed, since FLASH requires wait states to operate properly; hence, affecting execution speed.

8.2.3 InstaSPIN-MOTION in ROM

InstaSPIN-MOTION, currently available on F2806xM devices, is designed in a modular structure. Customers can determine which functions will be included in Flash memory when their system is deployed.

InstaSPIN-MOTION Control, Identify, and Move components are available in ROM InstaSPIN-MOTION Plan and Public Library are available in RAM.

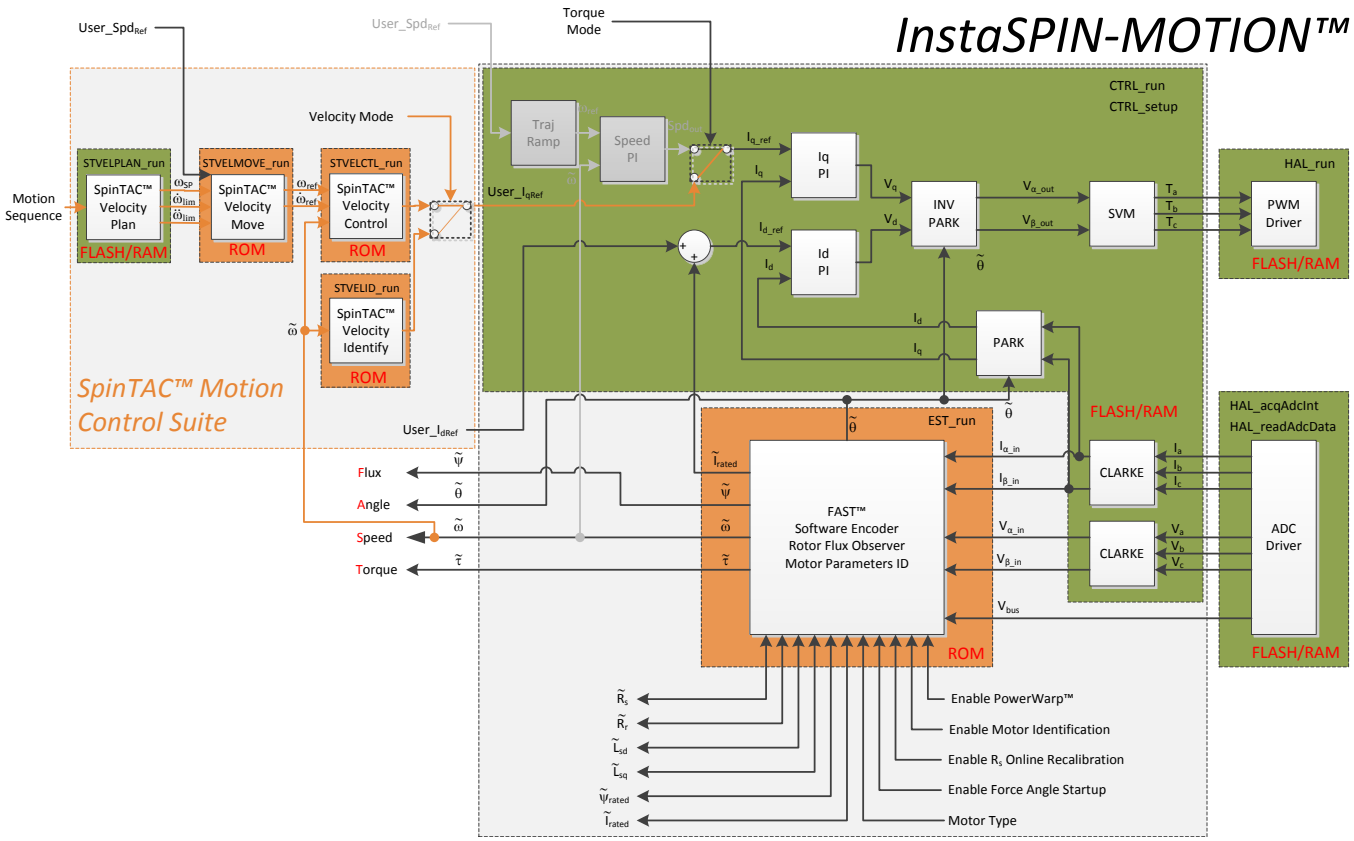


Figure 8-3. InstaSPIN-MOTION in ROM

8.2.3.1 Executing InstaSPIN-MOTION in the Main Interrupt

The InstaSPIN-MOTION components are individually executed in the main interrupt service routine. Each InstaSPIN-MOTION component should be called at a fixed decimation from the main ISR. It is recommended that the InstaSPIN-MOTION components be called at a rate at least 10 times slower than the PWM interrupt ISR or main ISR.

The InstaSPIN-MOTION core functions, including Control, Identify and Move, can only be executed from ROM. The user-included library and Plan can be executed from RAM or flash.

8.2.3.2 Executing Library from RAM

Loading and executing the public library from RAM presents a CPU performance advantage since RAM does not require wait states. However, for applications that require large amounts of RAM, executing the library from flash could be a better option.

Copyright © 2013–2019, Texas Instruments Incorporated

8.2.3.3 Executing Library from Flash

Loading and executing the public library from flash helps to decrease RAM consumption, although some RAM will be used by variables and stack. An important consideration is the flash wait states.

8.3 Details on CPU Load and Memory Footprint Measurements

8.3.1 CPU Utilization Measurement Details

In order to measure the CPU cycles as accurate as possible, one of the three available CPU timers was used. The timer was clocked as fast as possible in order to provide the maximum number of counts per execution. So the input clock of the timer was set to the same clock as the CPU with no prescaler. The following code example shows how the timer's count is reloaded and then read after the function of interest is executed:

```
// reload the CPU timer
HAL_reloadTimer0(halHandle);

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

// get the CPU timer count
timercount = HAL_getCountTimer0(halHandle);
```

Even though the functions that reload and read the timer count are as efficient as possible, there is an overflow of about 5 CPU cycles which have to be considered when using the data provided in the following sections.

The CPU utilization tables have a minimum column (Min) calculated by running hundreds of thousands of interrupts, and comparing each interrupt cycle time against a minimum, and if it is smaller, the minimum is updated. The same approach was followed to calculate the maximum number of cycles, or the Max column. The Average column was calculated by an accumulative number of cycles, and also counting the number of interrupts used for the accumulation, and then dividing the two numbers. Similarly to the Min and Max column, the Average is calculated over hundreds of thousands interrupts to generate a stable average.

The CPU utilization tables list a few optional configurations, changing three main things:

- The Interrupt vs. Controller (ISR vs CTRL) decimation rate, or tick rate
- The Controller vs. Estimator (CTRL vs EST) decimation rate, or tick rate
- The Rs Online recalibration feature

For the first two, related to the decimation rates, Figure 8-4 shows the entire software execution clock tree of InstaSPIN. This diagram shows how the clocks are divided all the way from the CPU clock to the estimator. We are only changing the highlighted tick rates, since these two are the main contributors of the CPU usage. Changing the speed controller, current controller or trajectory generation tick rates does not change the CPU usage significantly, so those are kept constant throughout the CPU utilization measurements.

Copyright © 2013–2019, Texas Instruments Incorporated

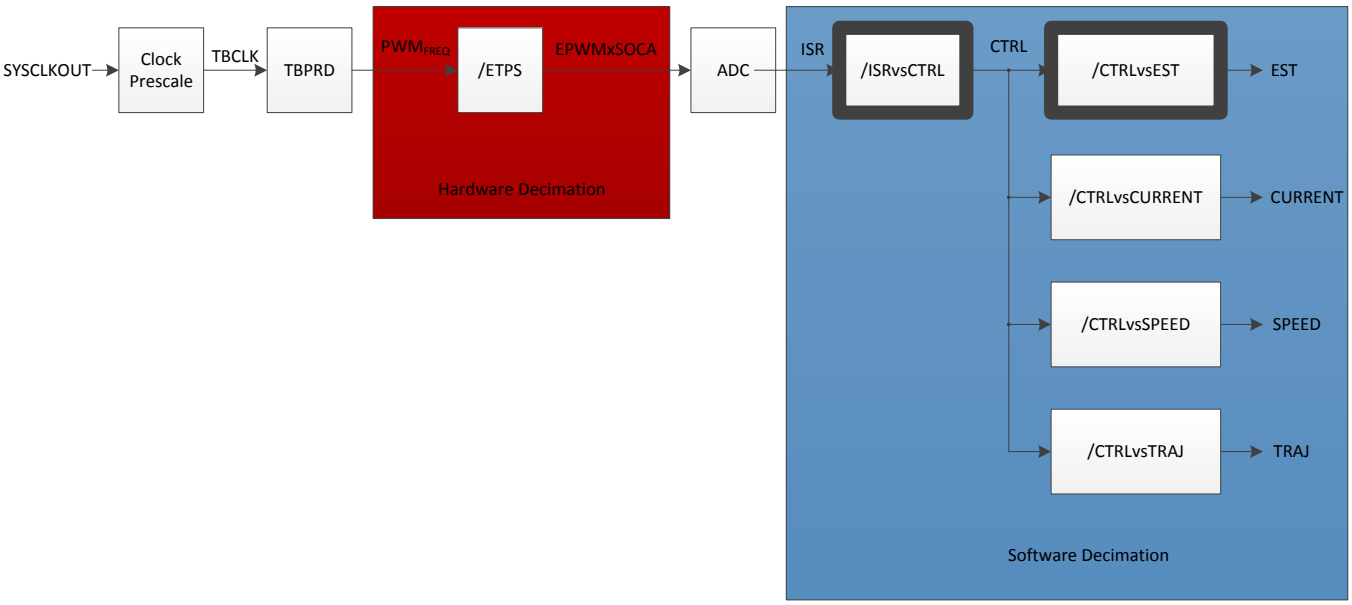


Figure 8-4. InstaSPIN Software Execution Clock Tree

For more information about InstaSPIN software execution clock tree, see the Section 9.1.

The third parameter that is enabled and disabled for the CPU utilization measurement is the Rs Online recalibration feature.

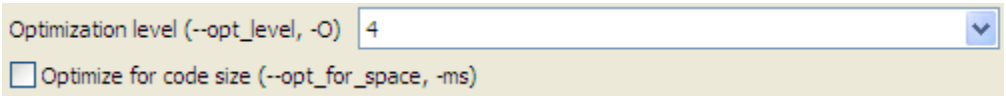
This is also considered since it impacts the CPU utilization considerably. To learn more about Rs Online recalibration, see Chapter 15.

8.3.2 Memory Allocation Measurement Details

Memory allocation depends on several factors. Here are some factors that affect how the memory is allocated as well as the configuration used for each item:

- Compiler Version: 6.1.0

- Optimization Settings: Level 4



- Additional User's Code: None. Minimal code was used to interface InstaSPIN libraries.

The entire command line showing the options of the compiler is also here:

-v28 -ml -mt -O4 -g

In order to have a minimum set of variables to interface InstaSPIN libraries, here is a list of must have global variables in your code:

```
CTRL_Handle ctrlHandle;

HAL_Handle halHandle;

USER_Params gUserParams;

HAL_PwmData_t gPwmData;
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
HAL_AdcData_t gAdcData;
```

For a complete description of these variables and the data type, see the labs. Some other variables might be useful to control the flow of the software, such as flags to enable or disable the system, as well as other global variables to display motor parameters. Those variables are not included in the project built to measure memory allocation, since they are not needed for the functionality of the libraries.

There are five different sections in the memory allocation tables:

- Library Interface (.ebss). This section of the table indicates the variables, in uninitialized memory area, or .ebss, used to interface the InstaSPIN libraries wherever they are, in user's memory or ROM.
- Library (.ebss). This one refers to the variables used for the InstaSPIN library itself, and the memory consumption for this area doesn't change.
- Code (.text). This section refers to the actual code being executed in user's memory. This code area will change depending on where the code is loaded (RAM or FLASH). It would also change if the compiler optimization settings are changed. Also, if the user is running most of the code from ROM, this code section will be minimized.
- IQmath (.text). The memory allocated for the code related to IQmath depends on the location of the InstaSPIN libraries. For a full implementation of InstaSPIN, where the majority of the code is in ROM, the IQmath code is minimized, since the ROM code itself has its own math code, and does not use user's memory for IQmath operations. For a minimum implementation of InstaSPIN, where a more math intensive code is operated from user's memory, some additional IQmath functions are needed in user's memory so the memory allocated to IQmath code increases for a minimum implementation of InstaSPIN.
- Max Stack Used (.ebss). This memory area is explained in a later section of this document.

8.3.3 IQ Math Built in ROM

The libraries in ROM were built with IQmath library version 1.5c. All code executable from ROM uses functions implemented in ROM itself, so the ROM code does not rely on externally added IQ math functionality.

However, code executed from user's memory with IQ math operations need an IQ math library to be added. This externally added IQ math library can be any released library version, not 1.5c necessarily. Users can mix their own version of IQ math library, and still execute code from ROM which uses IQ math library version 1.5c. All example code includes a full IQ math library in the CCStudio project.

8.3.4 Stack Utilization Measurement Details

The stack utilization was measured by the following procedure:

- Device is reset.
- The entire memory section where the stack is placed by the linker is initialized with known values, (0x5555AAAA, 0x12345678, 0x0BADF00D, 0xCAFEBABE, 0xFEEDFACE, and so on).
- Run the code for a few minutes, exercising all branches.
- Analyze the memory area where the stack is allocated and look for the last value changed, before the initialized values are present.
- Calculate the memory area modified.

Although the stack utilization method does not guarantee an absolute number of words needed for the stack, it gives a good idea of the stack area needed. However, it is recommended to have a stack section bigger than the minimum requirement to provide more robustness to the entire project. For additional details on this topic, see the *Online Stack Overflow Detection on the TMS320C28x DSP* application report (SPRA820).

The number on the tables listed in the following sections represent the maximum stack utilized, not the stack area reserved by the build options. As mentioned in this section, it is recommended to have a greater stack area reserved to avoid potential stack overflow conditions, especially when adding more code, other interrupts, or simply more variables.

Copyright © 2013–2019, Texas Instruments Incorporated

8.3.5 InstaSPIN Main Interrupt

The InstaSPIN library is executed at a fixed frequency from a single interrupt service routine. By default this main ISR is triggered by an end of conversion interrupt from the ADC. This conversion is first started by the PWM module at a fixed rate. Once in the ISR, mainISR() for this example, a series of function calls are needed in order to get data from the ADC and to call the functions in ROM. The following code is an example of this:

```
interrupt void mainISR(void)
{

  // 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);


  // run the driver -- set the pwm compare values
  HAL_writePwmData(halHandle,&gPwmData);


  // setup the controller
  CTRL_setup(ctrlHandle);


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

In order to describe the performance of InstaSPIN we will consider a top-level approach first, including these five function calls from the main ISR (see Figure 8-5).

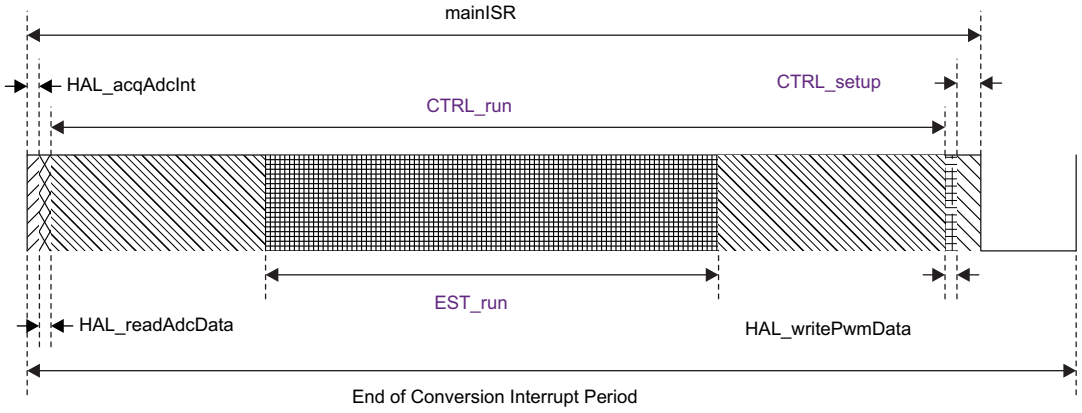


Figure 8-5. Function Calls from the Main ISR

8.3.6 Clock Rate

InstaSPIN-FOC and InstaSPIN-MOTION are real-time control systems and thus its performance is directly linked with the CPU clock rate of the processor it is executing on. The CPU clock rate can be reduced and the performance of InstaSPIN-FOC and InstaSPIN-MOTION can be tested to see if it meets the application requirements. Aspects of CPU loading are covered in Section 8.5 and the software clock tree of InstSPIN-FOC and InstaSPIN-MOTION is covered in Chapter 9.

Copyright © 2013–2019, Texas Instruments Incorporated

8.4 Memory Footprint

InstaSPIN-FOC is stored in a region of the device ROM that is execute-only (EXE-only) memory such that it is not readable by software or IDE.

Table 8-2. Allocated Memory for InstaSPIN-FOC Library

Features	2806xF	2806xM	2805xF	2805xM	2802xF
FAST	Yes	Yes	Yes	Yes	Yes
SpinTAC	No	Yes	No	Yes	No
Maximum Number of Motors that can be controlled	2	2	2	2	1
Relocalable Controller Structure	No	No	Yes	Yes	Yes
FAST Version	1.6	1.6	1.7	1.7	1.7
Public Library needs to be added to project	No	No	No	No	Yes
ROM Library Start [address, hex]	3F 8000	3F 8000	3F 8808	3F 8808	3F C000
Library Required RAM [size, hex, words]	800	800	800	800	200
Library Start RAM [address, hex]	01 3800	01 3800	00 8000	00 8000	00 0600

8.4.1 Device Memory Map

8.4.1.1 F2806xF and F2806xM Devices

For the F2806xF and F2806xM devices, InstaSPIN-FOC v1.6 and SpinTAC v2.1.8 is stored in address range of 0x3F8000 to 0x3FBFFF and the last part of L8-RAM is reserved for InstaSPIN variables, address range 0x013800 to 0x013FFF. Note that the InstaSPIN-FOC and InstaSPIN-MOTION variable range is fixed and must not be used. The rest of L8-RAM is available to the user (0x012000 to 0x0137FF); see Figure 8-6.

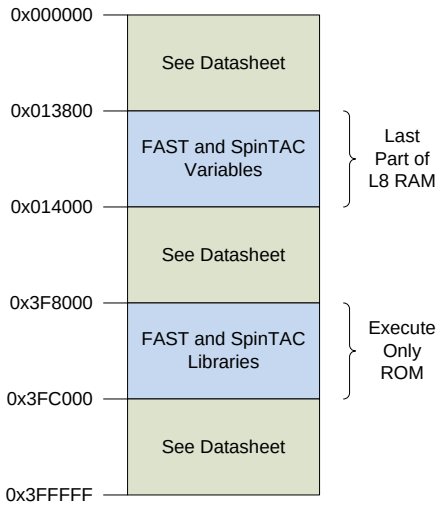


Figure 8-6. F2806xF and F2806xM Allocated Memory for InstaSPIN-FOC and SpinTAC Library

In addition to InstaSPIN-FOC v1.6 and SpinTAC v2.1.8 stored in ROM, several tables in ROM have moved to new addresses. If you are porting existing code that references for example IQmath tables in ROM, your linker command file will require an update of the addresses as shown in Table 8-3.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-3. ROM Table Addresses

Starting Address in ROM	F2806x	F2806xF and F2806xM
FPUTABLES	0X03FD860	0X03FD590
IQTABLES	0X03FDF00	0X03FDC30
IQTABLES2	0X03FEA50	0X03FE780
IQTABLES3	0X03FEADC	0x03FE80C

8.4.1.2 F2805xF and F2805xM Devices

For the 2805xF and 2805xM devices, InstaSPIN-FOC v1.7 and SpinTAC v2.1.8 is stored in address range of 0x3F8808 to 0x3FC52F and L0-RAM is reserved for InstaSPIN variables, address range 0x008000 to 0x0087FF; see Figure 8-7.

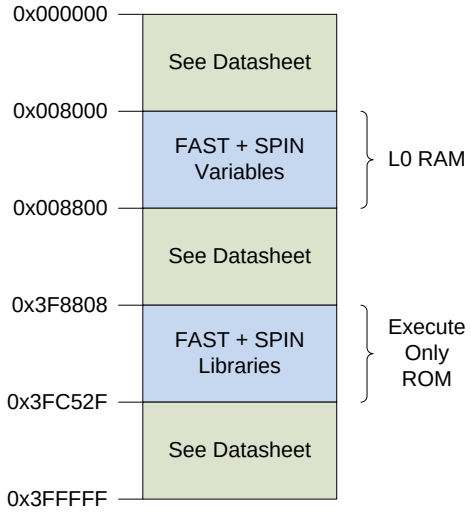


Figure 8-7. F2805xF and F2805xM Allocated Memory for InstaSPIN-FOC and SpinTAC Library

8.4.1.3 F2802xF Devices

For the 2802xF devices, InstaSPIN-FOC v1.7 is stored in address range of 0x3FC000 to 0x3FDFFF and the last part of M1-RAM is reserved for InstaSPIN variables, address range 0x000600 to 0x0007FF. Note that the InstaSPIN-FOC variable range is fixed and must not be used. The rest of M1-RAM is available to the user (0x000400 to 0x0005FF); see Figure 8-8

Copyright © 2013–2019, Texas Instruments Incorporated

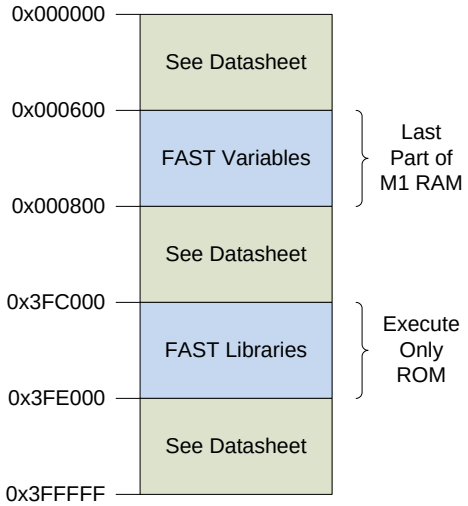


Figure 8-8. F2802xF Allocated Memory for InstaSPIN-FOC Library

8.4.2 InstaSPIN Memory Footprint

8.4.2.1 F2806xF and F2805xF Devices

Table 8-4 summarizes the memory used for the four configurations discussed in this document. Note the code size increase as fewer functions in ROM are used.

Table 8-4. Total Memory Usage of InstaSPIN-FOC for F2806xF and F2805xF Devices

Code Configurations		Memory Sizes (16-bit Words)			Maximum Stack Used (16-bit Words)
ROM Code	User Code	RAM	Flash	Total	
Full Implementation	RAM	0x1870	0x0000	**0x1870**	0x0120
Full Implementation	FLASH	0x001E	0x186C	**0x188A**	0x0120
Min Implementation	RAM	0x1F31	0x0000	**0x1F31**	0x0120
Min Implementation	FLASH	0x001E	0x1F2D	**0x1F4B**	0x0120

8.4.2.2 F2802xF Devices

Table 8-5 summarizes the memory used for the only configuration available for F2802xF devices.

Table 8-5. Total Memory Usage of InstaSPIN-FOC for F2802xF Devices

Code Configurations		Memory Sizes (16-bit Words)			Maximum Stack Used (16-bit Words)
ROM Code	User Code	RAM	Flash	Total	
Min Implementation	FLASH	0x06B2	0x2DD8	**0x348A**	0x0120

8.4.2.3 F2806xM and F2805xM Devices

To calculate the memory usage for InstaSPIN-MOTION, add the InstaSPIN-FOC memory usage to the SpinTAC memory usage in Table 8-6. The different memory requirements of SpinTAC Velocity Plan and SpinTAC Position Plan represent how many configuration functions are used in the project. RAM size is taken from the linker section ".ebss" and FLASH size from ".text".

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-6. Code Size and RAM Usage for SpinTAC Components

Component	Code (.text) (16-Bit Words)	RAM (.ebss) (16-Bit Words)
Velocity Control	0X2E6	0x4C
Velocity Move	0x488	0x5C
Velocity Plan (Min)	0x666	0x4E
Velocity Plan (Max)	0x14BA	0x4E
Velocity Identify	0x392	0x3C
Position Converter	0x21C	0x4C
Position Control	0x416	0x62
Position Move	0x13A4	0xCC
Position Plan (Min)	0x7AE	0x60
Position Plan (Max)	0x16F4	0x60

Section 14.1.1.5 breaks down the maximum stack utilization of SpinTAC components when run individually. The stack consumption of InstaSPIN-FOC is included.

Table 8-7. Stack Utilization of SpinTAC Components + InstaSPIN-FOC

Configuration (InstaSPIN-FOC is running in all cases)	Maximum Stack Used (16-bit Words)
Velocity Control	0x0120
Velocity Move	0x0120
Velocity Plan + Move + Control	0x0120
Velocity Identify	0x0120
Position Converter	0x0120
Position Control	0x0120
Position Move	0x0120
Position Plan + Move + Control	0x0120

8.4.3 Memory Wait-States

For additional wait-state options, refer to the device-specific data sheet for the device that you are using.

8.4.3.1 F2806xF/M Devices

The wait states shown in Table 8-8 were set for the CPU execution time measurements when executing code from FLASH at 90 MHz for the F2806xF/M devices.

Table 8-8. CPU Execution Time Wait States (F2806xF and F2806xM Devices)

Page Wait State	Random Wait State	OTP Wait State
3	3	5

8.4.3.2 F2805xF/M Devices

The wait states shown in Table 8-9 were set for the CPU execution time measurements when executing code from FLASH at 60 MHz for the F2805xF/M devices.

Table 8-9. CPU Execution Time Wait States (F2805xF and F2805xM Devices)

Page Wait State	Random Wait State	OTP Wait State
2	2	3

Copyright © 2013–2019, Texas Instruments Incorporated

8.4.3.3 F2802xF Devices

The wait states shown in Table 8-10 were set for the CPU execution time measurements when executing code from FLASH at 60 MHz for the F2802xF devices.

Table 8-10. CPU Execution Time Wait States (F2802xF Devices)

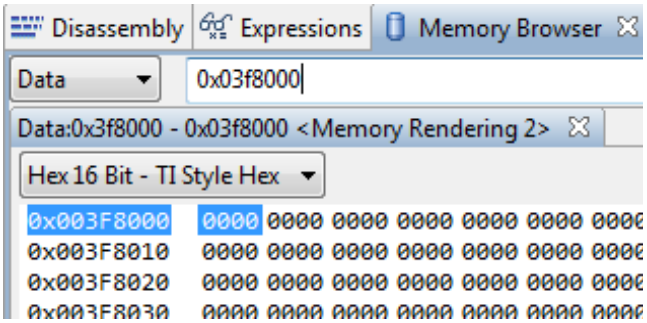
Page Wait State	Random Wait State	OTP Wait State
2	2	3

8.4.4 Flash Configuration Required Even for RAM-Only Execution

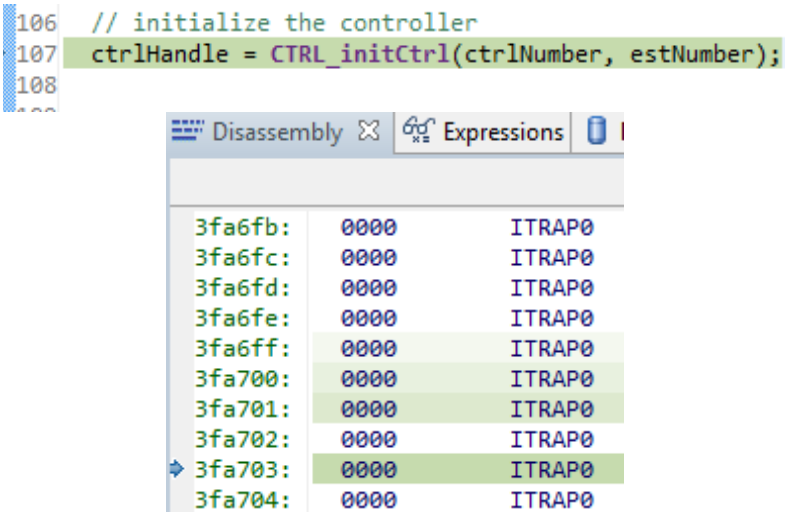
InstaSPIN-FOC and InstaSPIN-MOTION execute from ROM, but also access the OTP on the F2806xF and F2806xM and the OTP is a flash-based technology which requires the flash memory to be configured. Therefore, when you are running the labs from this guide you will notice that the file Flash.c is included in the project. The function FLASH_init() is called from HAL_init() which is called from main() in the lab. MotorWare provides drivers for all the peripherals on the device, including flash.

8.4.5 Debug (IDE) of EXE-Only Memory

Even though InstaSPIN-FOC and InstaSPIN-MOTION are stored in EXE-only ROM the debug experience is much the same except that you will not be able to see the contents of memory. Viewing memory from CCStudio using the Memory Browser, contents of EXE-only memory will be all 0s.



If you single-step through EXE-only memory with the Disassembly window open, it will display the opcode for reading all 0s (ITRAP0). Below is an example of this, single-stepping into the function CTRL_initCtrl() from Lab02A.



Copyright © 2013–2019, Texas Instruments Incorporated

8.5 CPU Load

8.5.1 F2806xF Devices

8.5.1.1 CPU Cycles

The following tables summarize all of the performance data per function, when users' code is loaded and executed from FLASH, on a minimum implementation of InstaSPIN library. Note that the number of cycles does not change significantly between the different implementations since the FAST estimator block remains in ROM for each of these configurations. This estimator block consumes the most cycles of all the InstaSPIN-FOC blocks. For more details on managing execution time in the ISR, see Section 9.1.

8.5.1.1.1 User Code in RAM

8.5.1.1.1.1 Full Implementation

Table 8-11. Full Implementation Memory Usage Executing in RAM

Section	Memory Usage (16-bit Words)	
	RAM	Flash
Library Interface (.ebss)	0x018C	×
Library (.ebss)	0x0800	×
Code (.text)	0x1870	×
IQmath (.text)	0x0014	×

Table 8-12 summarizes all of the performance data per function, when users' code is loaded and executed from RAM, on a full implementation of InstaSPIN library.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-12. Full Implementation Executing in RAM

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
HAL_acqAdcInt	23	23	23	×	✓	×
HAL_readAdcData	106	106	106	×	✓	×
Ctrl_run						
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	2345	2355	2425			
CTRL vs EST = 2	1154	1760	2425			
CTRL vs EST = 3	1154	1562	2425			
ISR vs CTRL = 2, CTRL vs EST = 1	58	1207	2425			
CTRL vs EST = 2	58	909	2425			
CTRL vs EST = 3	58	810	2425			
ISR vs CTRL = 3, CTRL vs EST = 1	58	824	2425			
CTRL vs EST = 2	58	626	2425			
CTRL vs EST = 3	58	560	2425	✓	×	×
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	2807	2821	2894			
CTRL vs EST = 2	1154	1993	2894			
CTRL vs EST = 3	1154	1717	2894			
ISR vs CTRL = 2, CTRL vs EST = 1	58	1439	2894			
CTRL vs EST = 2	58	1025	2894			
CTRL vs EST = 3	58	887	2894			
ISR vs CTRL = 3, CTRL vs EST = 1	58	979	2894			
CTRL vs EST = 2	58	702	2894			
CTRL vs EST = 3	58	610	2894			
HAL_writePwmData	62	62	62	×	✓	×
CTRL_setup	37	51	178	✓	×	×

8.5.1.1.1.2 Minimum Implementation

Table 8-13. Minimum Implementation Memory Usage Executing in RAM

Section	Memory Usage (16-bit Words)	
	RAM	Flash
Library Interface (.ebss)	0x018C	×
Library (.ebss)	0x0800	×
Code (.text)	0x1F31	×
IQmath (.text)	0x0064	×

Table 8-14 summarizes all of the performance data per function, when users' code is loaded and executed from RAM, on a minimum implementation of InstaSPIN library. Notice that CTRL_run is executed from both ROM and RAM. That is because CTRL_run has some function calls to the estimator. For instance, the EST_run function call is executed from CTRL_run, so that will be executed from ROM. Similarly, CTRL_setup has some code that calls some InstaSPIN state machine code, which needs to be executed from ROM because it contains some interaction with the FAST estimator. The difference in Code from the full implementation running from RAM is an additional Offset object added as well as the entire FOC code inlined in the code.

Table 8-14. Minimum Implementation Executing in RAM

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
HAL_acqAdcInt	23	23	23	×	✓	×

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-14. Minimum Implementation Executing in RAM (continued)

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
HAL_readAdcData	106	106	106	×	✓	×
Ctrl_run						
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	2361	2372	2454			
CTRL vs EST = 2	1171	1777	2454			
CTRL vs EST = 3	1171	1579	2454			
ISR vs CTRL = 2, CTRL vs EST = 1	59	1215	2454			
CTRL vs EST = 2	59	918	2454			
CTRL vs EST = 3	59	819	2454			
ISR vs CTRL = 3, CTRL vs EST = 1	59	830	2454			
CTRL vs EST = 2	59	631	2454			
CTRL vs EST = 3	59	565	2454	✓	×	×
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	2825	2840	2925			
CTRL vs EST = 2	1171	2012	2925			
CTRL vs EST = 3	1171	1736	2925			
ISR vs CTRL = 2, CTRL vs EST = 1	59	1450	2925			
CTRL vs EST = 2	59	1035	2925			
CTRL vs EST = 3	59	897	2925			
ISR vs CTRL = 3, CTRL vs EST = 1	59	986	2925			
CTRL vs EST = 2	59	710	2925			
CTRL vs EST = 3	59	618	2925			
HAL_writePwmData	62	62	62	×	✓	×
CTRL_setup	36	50	178	✓	✓	×

8.5.1.1.2 User Code in FLASH

8.5.1.1.2.1 Full Implementation

Table 8-15. Full Implementation Memory Usage Executing in FLASH

Section	Memory Usage (16-bit Words)	
	RAM	Flash
Library Interface (.ebss)	0x018C	×
Library (.ebss)	0x0800	×
Code (.text)	0x001E	0x186C
IQmath (.text)	×	0x0014

Table 8-16 shows the resource utilization when a full implementation of InstaSPIN is done, as well as users' code is loaded to FLASH. The Code section now adds a couple of functions that initialize the FLASH memory, which need to be run from RAM (loaded under ram functions). That is the reason of the new code from RAM, and not all of it from FLASH. There is a memCopy function added to the code when running from FLASH, which increases the code section as well.

Table 8-16. Full Implementation Executing in FLASH

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
HAL_acqAdcInt	25	25	25	×	×	✓

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-16. Full Implementation Executing in FLASH (continued)

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
HAL_readAdcData	108	108	108	×	×	✓
Ctrl_run						
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	2345	2355	2425			
CTRL vs EST = 2	1154	1760	2425			
CTRL vs EST = 3	1154	1562	2425			
ISR vs CTRL = 2, CTRL vs EST = 1	58	1207	2425			
CTRL vs EST = 2	58	909	2425			
CTRL vs EST = 3	58	810	2425			
ISR vs CTRL = 3, CTRL vs EST = 1	58	824	2425			
CTRL vs EST = 2	58	626	2425			
CTRL vs EST = 3	58	560	2425	✓	×	×
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	2807	2821	2894			
CTRL vs EST = 2	1154	1993	2894			
CTRL vs EST = 3	1154	1717	2894			
ISR vs CTRL = 2, CTRL vs EST = 1	58	1439	2894			
CTRL vs EST = 2	58	1025	2894			
CTRL vs EST = 3	58	887	2894			
ISR vs CTRL = 3, CTRL vs EST = 1	58	979	2894			
CTRL vs EST = 2	58	702	2894			
CTRL vs EST = 3	58	610	2894			
HAL_writePwmData	64	64	64	×	×	✓
CTRL_setup	37	51	178	✓	×	×

8.5.1.1.2.2 Minimum Implementation

Table 8-17. Minimum Implementation Memory Usage Executing in FLASH

Section	Memory Usage (16-bit Words)	
	RAM	Flash
Library Interface (.ebss)	0x018C	×
Library (.ebss)	0x0800	×
Code (.text)	0x001E	0x1F2D
IQmath (.text)	×	0x0064

Table 8-18 shows the resource utilization when a minimum implementation of InstaSPIN is done and the users' code is loaded to FLASH.

Table 8-18. Minimum Implementation Executing in FLASH

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
HAL_acqAdcInt	25	25	25	×	×	✓
HAL_readAdcData	108	108	108	×	×	✓

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-18. Minimum Implementation Executing in FLASH (continued)

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
Ctrl_run						
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	2447	2459	2544			
CTRL vs EST = 2	1257	1864	2544			
CTRL vs EST = 3	1257	1665	2544			
ISR vs CTRL = 2, CTRL vs EST = 1	71	1265	2544			
CTRL vs EST = 2	71	967	2544			
CTRL vs EST = 3	71	868	2544			
ISR vs CTRL = 3, CTRL vs EST = 1	71	867	2544			
CTRL vs EST = 2	71	668	2544			
CTRL vs EST = 3	71	602	2544	✓	×	✓
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	2911	2927	3015			
CTRL vs EST = 2	1258	2098	3015			
CTRL vs EST = 3	1258	1822	3015			
ISR vs CTRL = 2, CTRL vs EST = 1	71	1499	3015			
CTRL vs EST = 2	71	1084	3015			
CTRL vs EST = 3	71	946	3015			
ISR vs CTRL = 3, CTRL vs EST = 1	71	1022	3015			
CTRL vs EST = 2	71	746	3015			
CTRL vs EST = 3	71	654	3015			
HAL_writePwmData	64	64	64	×	×	✓
CTRL_setup	46	60	188	✓	×	✓

8.5.1.2 CPU Load with PWM = 10 kHz

Table 8-19. Full Implementation Executing from ROM and FLASH

F2806xF CPU = 90 MHz Available MIPs = 90 MIPs PWM = 10 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	28.86	25.97	64.03
CTRL vs EST = 2	22.24	20.02	69.98
CTRL vs EST = 3	20.04	18.04	71.96
ISR vs CTRL = 2, CTRL vs EST = 1	16.1	14.49	75.51
CTRL vs EST = 2	12.79	11.51	78.49
CTRL vs EST = 3	11.69	10.52	79.48
ISR vs CTRL = 3, CTRL vs EST = 1	11.84	10.66	79.34
CTRL vs EST = 2	9.64	8.68	81.32
CTRL vs EST = 3	8.91	8.02	81.98
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	34.03	30.63	59.37
CTRL vs EST = 2	24.83	22.35	67.65
CTRL vs EST = 3	21.77	19.59	70.41
ISR vs CTRL = 2, CTRL vs EST = 1	18.68	16.81	73.19
CTRL vs EST = 2	14.08	12.67	77.33
CTRL vs EST = 3	12.54	11.29	78.71
ISR vs CTRL = 3, CTRL vs EST = 1	13.57	12.21	77.79
CTRL vs EST = 2	10.49	9.44	80.56

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-19. Full Implementation Executing from ROM and FLASH (continued)

F2806xF CPU = 90 MHz Available MIPs = 90 MIPs PWM = 10 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
CTRL vs EST = 3	9.47	8.52	81.48

Table 8-20. Minimum Implementation Executing from ROM and FLASH

F2806xF CPU = 90 MHz Available MIPs = 90 MIPs PWM = 10 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	30.01	27.01	62.99
CTRL vs EST = 2	23.4	21.06	68.94
CTRL vs EST = 3	21.19	19.07	70.93
ISR vs CTRL = 2, CTRL vs EST = 1	16.74	15.07	74.93
CTRL vs EST = 2	13.43	12.09	77.91
CTRL vs EST = 3	12.33	11.1	78.9
ISR vs CTRL = 3, CTRL vs EST = 1	12.32	11.09	78.91
CTRL vs EST = 2	10.11	9.1	80.9
CTRL vs EST = 3	9.38	8.44	81.56
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	35.21	31.69	58.31
CTRL vs EST = 2	26	23.4	66.6
CTRL vs EST = 3	22.93	20.64	69.36
ISR vs CTRL = 2, CTRL vs EST = 1	19.34	17.41	72.59
CTRL vs EST = 2	14.73	13.26	76.74
CTRL vs EST = 3	13.2	11.88	78.12
ISR vs CTRL = 3, CTRL vs EST = 1	14.04	12.64	77.36
CTRL vs EST = 2	10.98	9.88	80.12
CTRL vs EST = 3	9.96	8.96	81.04

8.5.1.3 CPU Load with PWM = 20 kHz

Table 8-21. Full Implementation Executing from ROM and FLASH

F2806xF CPU = 90 MHz Available MIPs = 90 MIPs PWM = 20 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	57.71	51.94	38.06
CTRL vs EST = 2	44.49	40.04	49.96
CTRL vs EST = 3	40.09	36.08	53.92
ISR vs CTRL = 2, CTRL vs EST = 1	32.2	28.98	61.02
CTRL vs EST = 2	25.58	23.02	66.98
CTRL vs EST = 3	23.38	21.04	68.96
ISR vs CTRL = 3, CTRL vs EST = 1	23.69	21.32	68.68
CTRL vs EST = 2	19.29	17.36	72.64
CTRL vs EST = 3	17.82	16.04	73.96
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	68.07	61.26	28.74
CTRL vs EST = 2	49.67	44.7	45.3
CTRL vs EST = 3	43.53	39.18	50.82
ISR vs CTRL = 2, CTRL vs EST = 1	37.36	33.62	56.38
CTRL vs EST = 2	28.16	25.34	64.66

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-21. Full Implementation Executing from ROM and FLASH (continued)

F2806xF CPU = 90 MHz Available MIPs = 90 MIPs PWM = 20 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
CTRL vs EST = 3	25.09	22.58	67.42
ISR vs CTRL = 3, CTRL vs EST = 1	27.13	24.42	65.58
CTRL vs EST = 2	20.98	18.88	71.12
CTRL vs EST = 3	18.93	17.04	72.96

Table 8-22. Minimum Implementation Executing from ROM and FLASH

F2806xF CPU = 90 MHz Available MIPs = 90 MIPs PWM = 20 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	60.02	54.02	35.98
CTRL vs EST = 2	46.8	42.12	47.88
CTRL vs EST = 3	42.38	38.14	51.86
ISR vs CTRL = 2, CTRL vs EST = 1	33.49	30.14	59.86
CTRL vs EST = 2	26.87	24.18	65.82
CTRL vs EST = 3	24.67	22.2	67.8
ISR vs CTRL = 3, CTRL vs EST = 1	24.64	22.18	67.82
CTRL vs EST = 2	20.22	18.2	71.8
CTRL vs EST = 3	18.76	16.88	73.12
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	70.42	63.38	26.62
CTRL vs EST = 2	52	46.8	43.2
CTRL vs EST = 3	45.87	41.28	48.72
ISR vs CTRL = 2, CTRL vs EST = 1	38.69	34.82	55.18
CTRL vs EST = 2	29.47	26.52	63.48
CTRL vs EST = 3	26.4	23.76	66.24
ISR vs CTRL = 3, CTRL vs EST = 1	28.09	25.28	64.72
CTRL vs EST = 2	21.96	19.76	70.24
CTRL vs EST = 3	19.91	17.92	72.08

8.5.1.4 CPU Load Examples

From the tables discussed in the previous section, CPU usage can be computed by adding up the cycles and calculating a percentage of use.

8.5.1.4.1 Example 1

Consider the following scenario:

- CPU Clock = 90 MHz
- Available MIPS = 90 MIPS
- PWM Frequency = 10 kHz
- InstaSPIN Implementation:
  - Full implementation, libraries in ROM and user's code in RAM (Section 8.2.1)
  - Rs Online Disabled
  - ISR vs CTRL = 1
  - CTRL vs EST = 1

The percentage of CPU used by the interrupt is calculated, where:

Maximum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup

Copyright © 2013–2019, Texas Instruments Incorporated

Maximum % of CPU Used by InstaSPIN =
100% * ((Maximum Cycles) / 90 MHz) * PWM Frequency =
100% * ((23 + 106 + 2425 + 62 + 178) / 90 MHz) * 10 kHz =
31.04%

Average % of CPU Used by InstaSPIN =
100% * ((Average Cycles) / 90 MHz) * PWM Frequency =
100% * ((23 + 106 + 2355 + 62 + 51) / 90 MHz) * 10 kHz =
28.86%

Another useful calculation is the number of MIPS used by the application. This can be calculated as follows:

Average MIPS used by InstaSPIN =
(Average % of CPU Used by InstaSPIN/100%) * Available MIPS =
(28.86% / 100%) * 90 MIPS =
25.97 MIPS

And then, we can calculate the average available MIPS for other tasks:

Average user's available MIPS =
Total Available MIPS – Average MIPS used by InstaSPIN =
90 MIPS – 25.97 MIPS =
64.03 MIPS Available for other tasks

8.5.1.4.2 Example 2

Typically Rs Online needs to be enabled and often the PWM frequency needs to be increased. To free-up CPU bandwidth, ISR vs CTRL is set to 2.

Consider the following conditions:

- CPU Clock = 90 MHz
- Available MIPS = 90 MIPS
- PWM Frequency = 20 kHz
- InstaSPIN Implementation:
  - Minimum implementation, libraries in ROM and user's code in RAM (Section 8.2.2)
  - Rs Online Enabled
  - ISR vs CTRL = 2
  - CTRL vs EST = 1

First, we need to calculate the average MIPS used by InstaSPIN under the given conditions:

Average % of CPU Used by InstaSPIN =
100% * ((Average Cycles) / 90 MHz) * PWM Frequency =
100% * ((23 + 106 + 1450 + 62 + 50) / 90 MHz) * 20 kHz =
37.58%

Second, the number of average MIPS used by InstaSPIN under the given conditions:

Average MIPS used by InstaSPIN =
(Average % of CPU Used by InstaSPIN/100%) * Available MIPS =
(37.58% / 100 %) * 90 MIPS =
33.82 MIPS

And then, we can calculate the average available MIPS for other tasks:

Average user's Available MIPS =
Total Available MIPS - Average MIPS used by InstaSPIN =
90 MIPS – 33.82 =
56.18 MIPS Available for other tasks

Copyright © 2013–2019, Texas Instruments Incorporated

8.5.2 F2806xM Devices

8.5.2.1 CPU Cycles

InstaSPIN-MOTION combines the functionality of InstaSPIN-FOC with the SpinTAC Motion Control Suite from LineStream Technologies. CPU usage can be computed by adding the InstaSPIN-FOC cycles and the SpinTAC cycles presented in the following sections, and calculating a percentage of use. Examples are provided in subsequent sections.

8.5.2.1.1 RAM Execution - SpinTAC Library and User Code

Table 8-23 summarizes all of the performance data per function, when the SpinTAC library is loaded and executed from RAM. Note that each function makes calls into the ROM memory to run core SpinTAC functions. The typical cases for each component are highlighted in **bold**.

Table 8-23. CPU Cycle Utilization for SpinTAC with Library Executing in RAM on F2806xM Devices

Function Name		CPU Cycles			Executed From		
		Min	Avg	Max	ROM	RAM	FLASH
STVELCTL_run (Velocity Control)							
	RES = 0, ENB = 0	158	158	158	✓	✓	×
	RES = 0, ENB = 1	**573**	**573**	**573**			
	First call after ENB = 1	1010	1010	1010			
		786	786	786			
	Change Inertia parameter	786	786	786			
	RES = 1, ENB = 1	289	289	289			
STVELMOVE_run (Velocity Move)							
	RES = 0, ENB = 0	202	202	202	✓	✓	×
stcurve	**RES = 0, ENB = 1**	**675**	**704**	**1346**			
scurve	**RES = 0, ENB = 1**	**638**	**669**	**1312**			
trap	**RES = 0, ENB = 1**	**509**	**576**	**1039**			
	RES = 1, ENB = 1	421	421	421			
STVELPLAN_run (Velocity Plan)					✓	✓	×
	RES = 1, ENB = 0	159	159	159			
	RES = 0, ENB = 1	**169**	**169**	**169**			
	First call after ENB = 1	285	285	285			
	STAY FSM State	194	194	194			
	Condition FSM State Calculation must be done for each State	374 (fixed) + 274 * Number of Transitions + 334 * Number of EXIT Actions					
	Transition FSM State Calculation must be done for each State	229 (fixed) + 378 * Number of ENTER Actions					
STVELPLAN_runTick (Velocity Plan)		58	78	78			
STVELID_run (Velocity Identify)					✓	✓	×
	RES = 0, ENB = 0	142	142	142			
	RES = 0, ENB = 1	**332**	**341**	**658**			
	First call after ENB = 1	1063	1063	1063			
	RES = 1, ENB = 1	249	249	249			
STPOSCONV_run (Position Converter)					✓	✓	×
	RES = 0, ENB = 0	110	110	110			
	RES = 0, ENB = 1	**322**	**341**	**343**			
	First call after ENB = 1	1060	1060	1060			
	RES = 1, ENB = 1	118	118	118			

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-23. CPU Cycle Utilization for SpinTAC with Library Executing in RAM on F2806xM Devices (continued)

Function Name		CPU Cycles			Executed From		
		Min	Avg	Max	ROM	RAM	FLASH
STPOSCTL_run (Position Control)							
	RES = 0, ENB =0	166	166	166			
	RES = 0, ENB = 1	**1120**	**1125**	**1140**			
	First call after ENB = 1	1903	1903	1903	✓	✓	×
	Change Bandwidth parameter	1611	1611	1611			
	Change Inertia parameter	1611	1611	1611			
	RES = 1, ENB = 1	385	385	385			
STPOSMOVE_run (Position Move)							
	RES = 0, ENB = 0	406	406	406			
stcurve	**RES = 0, ENB = 1**	**616**	**1383**	**2733**			
	First call after ENB = 1	1270	1377	2368			
scurve	**RES = 0, ENB = 1**	**616**	**1333**	**2561**	✓	✓	×
	First call after ENB = 1	1219	1337	2324			
trap	**RES = 0, ENB = 1**	**616**	**1253**	**2501**			
	First call after ENB = 1	1319	1608	2049			
	RES = 1, ENB = 1	877	877	877			
STPOSPLAN_run (Position Plan)							
	RES = 1, ENB = 0	166	166	166			
	RES = 0, ENB = 1	201	201	201			
	First call after ENB = 1	325	325	325			
	STAY FSM State	209	209	209			
	Condition FSM State Calculation must be done for each State	436 (fixed) + 276 * Number of Transitions + 334 * Number of EXIT Actions			✓	✓	×
	Transition FSM State Calculation must be done for each State	245 (fixed) + 378 * Number of ENTER Actions					
STPOSPLAN_runTick (Position Plan)		58	78	78			

8.5.2.1.2 FLASH Execution - SpinTAC Library and User Code

Table 8-24 summarizes all of the performance data per function, when the SpinTAC library is loaded and executed from Flash. InstaSPIN-FOC is in Full Implementation.

Table 8-24. CPU Cycle Utilization for SpinTAC Library Executing in Flash on F2806xM Devices

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
STVELCTL_run (Velocity Control)						
RES = , ENB = 0	216	216	216			
RES = 0, ENB = 1	**672**	**672**	**672**			
First call after ENB = 1	1183	1183	1183	✓	×	✓
Change Bandwidth	925	925	925			
Change Inertia parameter	925	925	925			
RES = 1, ENB = 1	396	396	396			

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-24. CPU Cycle Utilization for SpinTAC Library Executing in Flash on F2806xM Devices (continued)

Function Name		CPU Cycles			Executed From		
		Min	Avg	Max	ROM	RAM	FLASH
STVELMOVE_run (Velocity Move)							
	RES = 0, ENB = 0	258	258	258			
stcurve	**RES = 0, ENB = 1**	**780**	**816**	**1625**			
scurve	**RES = 0, ENB = 1**	**743**	**781**	**1589**	✓	×	✓
trap	**RES = 0, ENB = 1**	**616**	**700**	**1309**			
	RES = 1, ENB = 1	554	554	554			
STVELPLAN_run (Velocity Plan)							
	RES = 1, ENB = 0	219	219	219			
	RES = 0, ENB = 1	**266**	**266**	**266**			
	First call after ENB = 1	393	393	393			
	STAY FSM State	280	280	280			
	Transition FSM State Calculation must be done for each State	488 (fixed) + 368 * Number of Transitions + 440 * Number of EXIT Actions			✓	×	✓
	Condition FSM State Calculation must be done for each State	337 (fixed) + 503 * Number of ENTER Actions					
STVELPLAN_runTick (Velocity Plan)		91	119	119			
STVELID_run (Velocity Identify)							
	RES = 1, ENB = 0	198	198	198			
	RES = 0, ENB = 1	**311**	**332**	**822**			
	First call after ENB = 1	1366	1366	1366	✓	×	✓
	RES = 1, ENB = 1	338	338	338			
STPOSCONV_run (Position Converter)							
	RES = 0, ENB = 0	145	145	145			
	RES = 0, ENB = 1	**443**	**448**	**450**			
	First call after ENB = 1	1372	1372	1372	✓	×	✓
	RES = 1, ENB = 1	170	170	170			
STPOSCTL_run (Position Control)							
	RES = 0, ENB =0	246	246	246			
	RES = 0, ENB = 1	**1311**	**1316**	**1326**			
	First call after ENB = 1	2236	2236	2236			
	Change Bandwidth parameter	1909	1909	1909	✓	×	✓
	Change Inertia parameter	1909	1909	1909			
	RES = 1, ENB = 1	509	509	509			
STPOSMOVE_run (Position Move)							
	RES = 0, ENB = 0	520	520	520			
stcurve	**RES = 0, ENB = 1**	**790**	**1611**	**3630**			
	First call after ENB = 1	1467	1588	2778			
scurve	**RES = 0, ENB = 1**	**790**	**1564**	**3205**	✓	×	✓
	First call after ENB = 1	1415	1551	2734			
trap	**RES = 0, ENB = 1**	**790**	**1501**	**3130**			
	First call after ENB = 1	1540	1903	2438			
	RES = 1, ENB = 1	1100	1100	1100			

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-24. CPU Cycle Utilization for SpinTAC Library Executing in Flash on F2806xM Devices (continued)

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
STPOSPLAN_run (Position Plan)						
RES = 1, ENB = 0	229	229	229			
RES = 0, ENB = 1	297	297	297			
First call after ENB = 1	450	450	450			
STAY FSM State	297	297	297			
Condition FSM State Calculation must be done for each State	548 (fixed) + 363 * Number of Transitions + 450 * Number of EXIT Actions			✓	×	✓
Transition FSM State Calculation must be done for each State	345 (fixed) + 508 * Number of ENTER Actions					
STPOSPLAN_runTick (Position Plan)	86	115	115	✓	×	✓

8.5.2.2 CPU Load Examples

8.5.2.2.1 Example 1

Consider the following scenario:

- CPU Clock = 90 MHz
- Available MIPS = 90 MIPS
- PWM Frequency = 10 kHz
- InstaSPIN Implementation:
  - InstaSPIN-FOC: Full implementation, libraries in ROM and user's code in RAM (Section 8.2.1)
  - SpinTAC Library: Velocity Control. Library in ROM and user library code in RAM.
  - Rs Online Disabled
  - ISR vs CTRL = 1
  - CTRL vs SPEED = 10

The percentage of CPU used by the interrupt is calculated, where:

Minimum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup

Maximum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup + STVELCTL_run

Minimum Cycles = 23 + 106 + 2355 + 62 + 51 = 2597 cycles

Maximum Cycles = 23 + 106 + 2355 + 62 + 51 + 573 = 3170 cycles

For each millisecond, the Minimum Cycles are used 9 times and the Maximum Cycles are used once.

Cycles in 1 millisecond = 2597 * 9 + 3170 * 1 = 26543 cycles

The CPU usage is now

100% * (26543 / 90 MHz) * (10 kHz / 10) = **29.49%**

Another useful calculation is the number of MIPS used by InstaSPIN. This can be calculated as follows:

Average MIPS used by InstaSPIN =

(Average % of CPU Used by InstaSPIN/100%) * Available MIPS =

(29.49 % / 100 %) * 90 MIPS =

26.54 MIPS

And then, we can calculate the average available MIPS for other tasks:

Average user's available MIPS =

Total Available MIPS – Average MIPS used by InstaSPIN(FOC + MOTION) =

90 MIPS – 26.54 MIPS =

Copyright © 2013–2019, Texas Instruments Incorporated

63.46 MIPS Available for other tasks

8.5.2.2.2 Example 2

For example, consider the following scenario:

- CPU Clock = 90 MHz
- Available MIPS = 90 MIPS
- PWM Frequency = 10 kHz
- InstaSPIN Implementation:
  - InstaSPIN-FOC: Full implementation, libraries in ROM and user's code in RAM (Section 8.2.1)
  - SpinTAC Library: Velocity Control + Velocity Move (stcurve). Library in ROM and user library code in RAM.
  - Rs Online Disabled
  - ISR vs CTRL = 1
  - CTRL vs SPEED = 10

The percentage of CPU used by the interrupt is calculated, where:

Minimum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup

Maximum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup + STVELCTL_run + STVELMOVE_run

Minimum Cycles = 23 + 106 + 2355 + 62 + 51 = 2597 cycles

Maximum Cycles = 23 + 106 + 2355 + 62 + 51 + 573 + 704 = 3874 cycles

For each millisecond, the Minimum Cycles are used 9 times and the Maximum Cycles are used once.

Cycles in 1 millisecond = 2597 * 9 + 3874 * 1 = 27247 cycles

The CPU usage is now

100% * (27247 / 90 MHz) * (10 kHz / 10) = **30.27%**

Another useful calculation is the number of MIPS used by the application. This can be calculated as follows:

Average MIPS used by InstaSPIN =

(Average % of CPU Used by InstaSPIN/100%) * Available MIPS =

(30.27 % / 100 %) * 90 MIPS =

27.24 MIPS

And then, we can calculate the average available MIPS for other tasks:

Average user's available MIPS =

Total Available MIPS – Average MIPS used by InstaSPIN(FOC + MOTION) =

90 MIPS – 27.24 MIPS =

62.76 MIPS Available for other tasks

8.5.2.2.3 Example 3

For this example, consider the following scenario:

- CPU Clock = 90 MHz
- Available MIPS = 90 MIPS
- PWM Frequency = 10 kHz
- InstaSPIN Implementation:
  - InstaSPIN-FOC: Full implementation, libraries in ROM and user's code in RAM (Section 8.2.1)
  - SpinTAC Library: Velocity Control + Velocity Move (stcurve) + Velocity Plan. Library in ROM and user library code in RAM.
  - Rs Online Disabled
  - ISR vs CTRL = 1
  - CTRL vs SPEED = 10

Copyright © 2013–2019, Texas Instruments Incorporated

SpinTAC Velocity Plan is used to generate a motion sequence between 4 states (see Figure 8-9). Note that in this example, both the Plan functions STVELPLAN_run and STVELPLAN_runTick are being run from the ISR. In a final implementation, STVELPLAN_run can be run from a slower ISR or background loop.

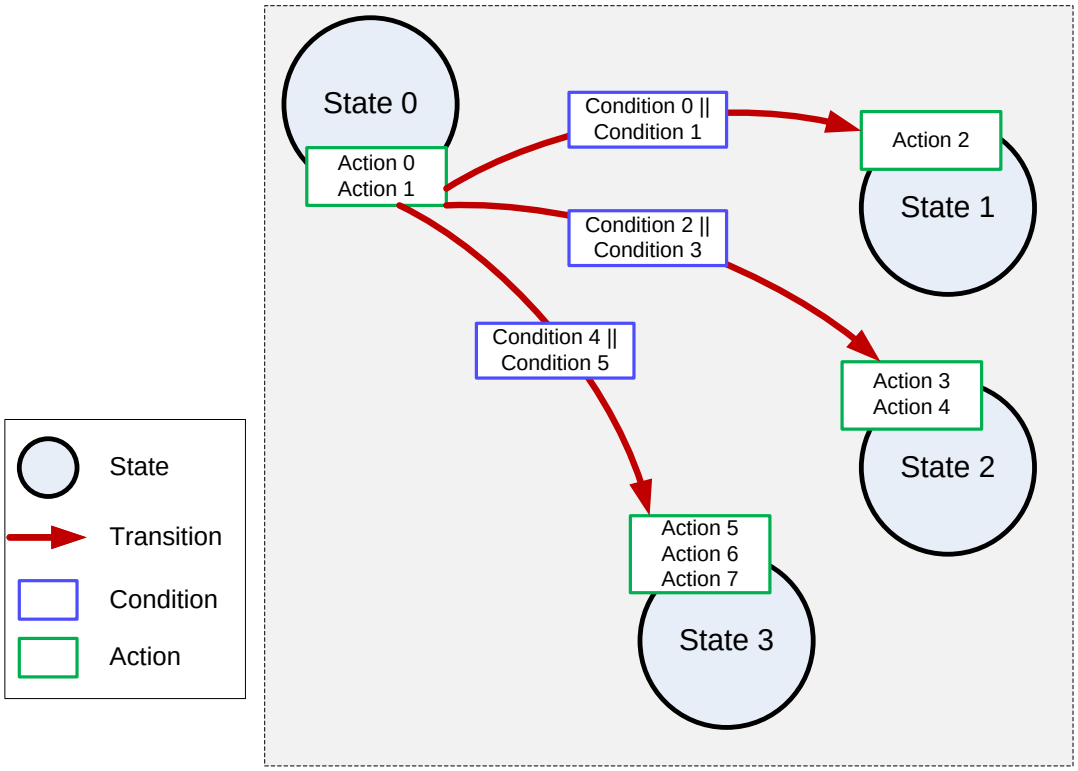


Figure 8-9. SpinTAC Velocity Plan Example

The maximum cycles for STVELPLAN_run are contributed either by the Condition FSM state **or** the Transition FSM state but never both at the same time.

1. Maximum cycles in Condition FSM state are determined by:
   - The number of transitions that originate from a given state.
   - The number of conditions being checked for a transition (worst case is two conditions)
   - The conditions compare a variable to a value or a variable to another variable (worst case)
   - The number of EXIT actions configured for that state.

   If the code is running in RAM, then Maximum cycles = 374 (fixed cycles for Condition FSM state) + (Number of Transitions * 274) + (Number of EXIT actions * 334).

   The example plan in Figure 8-9 configures three transitions leaving State 0. All transitions check conditions that compare variables with variables (worst case), and 2 EXIT actions are configured to this state. So the maximum cycles occurs when the last transition is taken = 374 + (3 * 274) + (2 * 334) = **1864**.
2. Maximum cycles in Transition FSM state are determined by the number of ENTER actions that are configured to the state being entered.

   If the code is running in RAM, then Maximum cycles = 229 (fixed cycles for Transition FSM state) + Number of ENTER actions * 378.

   The above example shows the State3 having 3 ENTER actions. This will cause the Transition FSM State to have maximum cycles = 229 + (3 * 378) = **1368**.

   Compare the two cases that can cause maximum cycles for STVELPLAN_run. In this instance, the maximum possible cycles are 1330, contributed by the Condition FSM state. Use that value as the worst case when calculating the percentage of CPU used by the interrupt.

Copyright © 2013–2019, Texas Instruments Incorporated

The percentage of CPU used by the interrupt is calculated, where:

Minimum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup

Maximum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup + STVELCTL_run + STVELMOVE_run + STVELPLAN_runTick + STVELPLAN_run

Minimum Cycles = 23 + 106 + 2355 + 62 + 51 = 2597 cycles

Maximum Cycles = 23 + 106 + 2355 + 62 + 51 + 573 + 704 + 78 + 1864 = 5738 cycles

For each millisecond, the Minimum Cycles are used 9 times and the Maximum Cycles are used once.

Cycles in 1 millisecond = 2597 * 9 + 5738 * 1 = 29111 cycles

The CPU usage is now

100% * (29111 / 90 MHz) * (10 kHz / 10) = **32.35%**

The above calculation is where the main component of SpinTAC Velocity Plan is being ran from the ISR. When SpinTAC Velocity Plan is called from the background loop, the percentage of CPU used by the interrupt is calculated below:

Maximum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup + STVELCTL_run + STVELMOVE_run + STVELPLAN_runTick

Maximum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup + STVELCTL_run + STVELMOVE_run + STVELPLAN_runTick + STVELPLAN_run

Minimum Cycles = 23 + 106 + 2355 + 62 + 51 = 2597 cycles

Maximum Cycles = 23 + 106 + 2355 + 62 + 51 + 573 + 704 + 78 = 3952 cycles

For each millisecond, the Minimum Cycles are used 9 times and the Maximum Cycles are used once.

Cycles in 1 millisecond = 2597 * 9 + 5738 * 1 = 27325 cycles

The CPU usage is now

100% * (27325 / 90 MHz) * (10 kHz / 10) = **30.36%**

8.5.2.2.4 Example 4 (SpinTAC Position)

For this example, consider the following scenario:

- CPU Clock = 90 MHz
- Available MIPS = 90 MIPS
- PWM Frequency = 10 kHz
- InstaSPIN Implementation:
  - InstaSPIN-FOC: Full implementation, libraries in ROM and user's code in RAM (Section 8.2.1)
  - SpinTAC Library: Position Converter + Position Control + Position Move (stcurve). Library in ROM and user library code in RAM.
  - Rs Online Disabled
  - ISR vs CTRL = 1
  - CTRL vs SPEED = 10

The percentage of CPU used by the interrupt is calculated, where:

Minimum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup

Maximum Cycles = HAL_acqAdcInt + HAL_readAdcData + Ctrl_run + HAL_writePwmData + Ctrl_setup + STPOSCONV_run + STPOSCTL_run + STPOSMOVE_run

Minimum Cycles = 23 + 106 + 2355 + 62 + 51 = 2597 cycles

Maximum Cycles = 23 + 106 + 2355 + 62 + 51 + 341 + 1125 + 1383 = 5446 cycles

For each millisecond, the Minimum Cycles are used 9 times and the Maximum Cycles are used once.

Cycles in 1 millisecond = 2597 * 9 + 5446 * 1 = 28819 cycles

The CPU usage is now

100% * (28819 / 90 MHz) * (10 kHz / 10) = **32.02%**

Another useful calculation is the number of MIPS used by the application. This can be calculated as follows:

Average MIPS used by InstaSPIN =

(Average % of CPU Used by InstaSPIN/100%) * Available MIPS =

Copyright © 2013–2019, Texas Instruments Incorporated

(32.02 % / 100 %) * 90 MIPS =

28.82 MIPS

And then, we can calculate the average available MIPS for other tasks:

Average user's available MIPS =

Total Available MIPS – Average MIPS used by InstaSPIN(FOC + MOTION) =

90 MIPS – 28.82 MIPS =

61.18 MIPS Available for other tasks

8.5.3 F805xF Devices

8.5.3.1 CPU Cycles

The F2805xF cycle count will be essentially the same as the F2806xF cycle.

8.5.3.2 CPU Load with PWM = 10 kHz

Table 8-25. Full Implementation Executing from ROM and FLASH

F2805xF CPU = 60 MHz Available MIPs = 60 MIPs PWM = 10 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
R_s Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	43.28	25.97	34.03
CTRL vs EST = 2	33.37	20.02	39.98
CTRL vs EST = 3	30.07	18.04	41.96
ISR vs CTRL = 2, CTRL vs EST = 1	24.15	14.49	45.51
CTRL vs EST = 2	19.18	11.51	48.49
CTRL vs EST = 3	17.53	10.52	49.48
ISR vs CTRL = 3, CTRL vs EST = 1	17.77	10.66	49.34
CTRL vs EST = 2	14.47	8.68	51.32
CTRL vs EST = 3	13.37	8.02	51.98
R_s Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	51.05	30.63	29.37
CTRL vs EST = 2	37.25	22.35	37.65
CTRL vs EST = 3	32.65	19.59	40.41
ISR vs CTRL = 2, CTRL vs EST = 1	28.02	16.81	43.19
CTRL vs EST = 2	21.12	12.67	47.33
CTRL vs EST = 3	18.82	11.29	48.71
ISR vs CTRL = 3, CTRL vs EST = 1	20.35	12.21	47.79
CTRL vs EST = 2	15.73	9.44	50.56
CTRL vs EST = 3	14.20	8.52	51.48

8.5.3.3 CPU Load with PWM = 20 kHz

Table 8-26. Full Implementation Executing from ROM and FLASH

F2805xF CPU = 60 MHz Available MIPs = 60 MIPs PWM = 20 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
R_s Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	86.57	51.94	8.06
CTRL vs EST = 2	66.73	40.04	19.96
CTRL vs EST = 3	60.13	36.08	23.92
ISR vs CTRL = 2, CTRL vs EST = 1	48.3	28.98	31.02
CTRL vs EST = 2	38.37	23.02	36.98

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-26. Full Implementation Executing from ROM and FLASH (continued)

F2805xF CPU = 60 MHz Available MIPs = 60 MIPs PWM = 20 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
CTRL vs EST = 3	35.07	21.04	38.96
ISR vs CTRL = 3, CTRL vs EST = 1	35.53	21.32	38.68
CTRL vs EST = 2	28.93	17.36	42.64
CTRL vs EST = 3	26.73	16.04	43.96
R_s Online Enabled, ISR vs CTRL = 1			
CTRL vs EST = 2	74.5	44.7	15.3
CTRL vs EST = 3	65.3	39.18	20.82
ISR vs CTRL = 2, CTRL vs EST = 1	56.03	33.62	26.38
CTRL vs EST = 2	42.23	25.34	34.66
CTRL vs EST = 3	37.63	22.58	37.42
ISR vs CTRL = 3, CTRL vs EST = 1	40.7	24.42	35.58
CTRL vs EST = 2	31.47	18.88	41.12
CTRL vs EST = 3	28.4	17.04	42.96

8.5.4 F2805xM Devices

8.5.4.1 CPU Cycles

8.5.4.1.1 FLASH Execution - SpinTAC Library and User Code

Table 8-27. CPU Cycle Utilization for SpinTAC Library Executing in Flash on F2805xM Device

Function Name		CPU Cycles			Executed From		
		Min	Avg	Max	ROM	RAM	FLASH
STVELCTL_run (Velocity Control)							
	RES = 1, ENB = 0	189	189	189			
	RES = 0, ENB = 1	**614**	**614**	**614**			
	First call after ENB = 1	1077	1077	1077	✓	×	✓
	Change Bandwidth	842	842	842			
	Change Inertia parameter	842	842	842			
	RES = 1, ENB = 1	347	347	347			
STVELMOVE_run (Velocity Move)							
	RES = 1, ENB = 0	220	220	220			
stcurve	**RES = 0, ENB = 1**	**724**	**759**	**1468**			
scurve	**RES = 0, ENB = 1**	**687**	**724**	**1435**	✓	×	✓
trap	**RES = 0, ENB = 1**	**561**	**636**	**1167**			
	RES = 1, ENB = 1	494	494	494			

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-27. CPU Cycle Utilization for SpinTAC Library Executing in Flash on F2805xM Device (continued)

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
STVELPLAN_run (Velocity Plan)						
RES = 1, ENB = 0	183	183	183			
RES = 0, ENB = 1	**238**	**238**	**238**			
First call after ENB = 1	333	333	333			
STAY FSM State	238	238	238			
Transiation FSM State Calculation must be done for each State	436 (fixed) + 320 * Number of Transitions + 388 * Number of EXIT Actions			✓	×	✓
Condition FSM State Calculation must be done for each State	283 (fixed) + 438 * Number of ENTER Actions					
STVELPLAN_runTick (ISR function)	76	100	100			
STVELID_run (Velocity Identify)						
RES = 1, ENB = 0	198	198	198			
RES = 0, ENB = 1	**256**	**278**	**723**			
First call after ENB = 1	1196	1196	1196	✓	×	✓
RES = 1, ENB = 1	292	292	292			
STPOSCOV_run (Position Converter)						
RES = 1, ENB = 0	127	127	127			
RES = 0, ENB = 1	**391**	**398**	**400**			
First call after ENB = 1	1209	1209	1209	✓	×	✓
RES = 1, ENB = 1	140	140	140			
STPOSCTL_run (Position Control)						
RES = 0, ENB =0	201	201	201			
RES = 0, ENB = 1	**1207**	**1212**	**1225**			
First call after ENB = 1	2043	2043	2043			
Change Bandwidth parameter	1729	1729	1729	✓	×	✓
Change Inertia parameter	1729	1729	1729			
RES = 1, ENB = 1	449	449	449			
STPOSMOVE_run (Position Move)						
RES = 0, ENB = 0	520	520	520			
stcurve **RES = 0, ENB = 1**	**790**	**1611**	**3630**			
Velocity Controlled Profile	1467	1588	2778			
scurve **RES = 0, ENB = 1**	**790**	**1564**	**3205**			
Velocity Controlled Profile	1415	1551	2734	✓	×	✓
trap **RES = 0, ENB = 1**	**790**	**1501**	**3130**			
Velocity Controlled Profile	1540	1903	2438			
RES = 1, ENB = 1	996	996	996			

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-27. CPU Cycle Utilization for SpinTAC Library Executing in Flash on F2805xM Device (continued)

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
STPOSPLAN_run (Position Plan)						
RES = 1, ENB = 0	202	202	20			
RES = 0, ENB = 1	255	255	255			
First call after ENB = 1	373	373	373			
STAY FSM State	255	255	255			
Condition FSM State Calculation must be done for each State	501 (fixed) + 323 * Number of Transitions + 382 * Number of EXIT Actions			✓	×	✓
Transition FSM State Calculation must be done for each State	301 (fixed) + 432 * Number of ENTER Actions					
STPOSPLAN_runTick (ISR function)	86	115	115			

Note: The difference in the CPU cycles is due to the lower flash wait states in the F2805xM device as compared to the F2806xM device.

8.5.5 F2802xF Devices

8.5.5.1 CPU Cycles

Table 8-28. Minimum Implementation Memory Usage Executing in FLASH

Section	Memory Usage (16-bit Words)	
	RAM	Flash
Library Interface (.ebss)	0x0326	×
Library (.ebss)	0x0200	×
Code (.text)	0x06B6	0x2ED7
.cinit	×	0x007A
Constants (.econst)	×	0x0080
IQmath (.text)	×	0x00C9

Section 12.3.5.3 summarizes all of the performance data per function, when users' code is loaded and executed from FLASH, on a minimum implementation of InstaSPIN library. Note that the number of cycles does not change significantly between the different implementations since the FAST estimator block remains in ROM for each of these configurations. This estimator block consumes the most cycles of all the InstaSPIN-FOC blocks. For more details on managing execution time in the ISR, see Section 9.1.

Table 8-29. Minimum Implementation Executing in FLASH

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
HAL_acqAdcInt	17	17	17	×	✓	×
HAL_readAdcData	94	94	94	×	✓	×

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-29. Minimum Implementation Executing in FLASH (continued)

Function Name	CPU Cycles			Executed From		
	Min	Avg	Max	ROM	RAM	FLASH
Ctrl_run						
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	2320	2331	2413			
CTRL vs EST = 2	1131	1735	2413			
CTRL vs EST = 3	1131	1536	2413			
ISR vs CTRL = 2, CTRL vs EST = 1	51	1191	2413			
CTRL vs EST = 2	51	893	2413			
CTRL vs EST = 3	51	793	2413			
ISR vs CTRL = 3, CTRL vs EST = 1	51	811	2413			
CTRL vs EST = 2	51	612	2413			
CTRL vs EST = 3	51	544	2413	✓	✓	✓
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	2766	2781	2882			
CTRL vs EST = 2	1129	1969	2882			
CTRL vs EST = 3	1129	1692	2882			
ISR vs CTRL = 2, CTRL vs EST = 1	51	1424	2882			
CTRL vs EST = 2	51	1010	2882			
CTRL vs EST = 3	51	871	2882			
ISR vs CTRL = 3, CTRL vs EST = 1	51	966	2882			
CTRL vs EST = 2	51	689	2882			
CTRL vs EST = 3	51	596	2882			
HAL_writePwmData	110	110	110	×	✓	×
CTRL_setup	26	36	188	×	✓	✓

8.5.5.2 CPU Load with PWM = 10 kHz

Table 8-30. Minimum Implementation Executing from ROM, RAM, and FLASH

F2802xF CPU = 60 MHz Available MIPs = 60 MIPs PWM = 10 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	43.13	25.88	34.12
CTRL vs EST = 2	33.2	19.92	40.08
CTRL vs EST = 3	29.88	17.93	42.07
ISR vs CTRL = 2, CTRL vs EST = 1	24.13	14.48	45.52
CTRL vs EST = 2	19.17	11.5	48.5
CTRL vs EST = 3	17.5	10.5	49.5
ISR vs CTRL = 3, CTRL vs EST = 1	17.8	10.68	49.32
CTRL vs EST = 2	14.48	8.69	51.31
CTRL vs EST = 3	13.35	8.01	51.99
Rs Online Enabled, ISR vs CTRL = 1, CTRL vs EST = 1	50.63	30.38	29.62
CTRL vs EST = 2	37.1	22.26	37.74
CTRL vs EST = 3	32.48	19.49	40.51
ISR vs CTRL = 2, CTRL vs EST = 1	28.02	16.81	43.19
CTRL vs EST = 2	21.12	12.67	47.33
CTRL vs EST = 3	18.8	11.28	48.72
ISR vs CTRL = 3, CTRL vs EST = 1	20.38	12.23	47.77
CTRL vs EST = 2	15.77	9.46	50.54

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-30. Minimum Implementation Executing from ROM, RAM, and FLASH (continued)

F2802xF CPU = 60 MHz Available MIPs = 60 MIPs PWM = 10 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
CTRL vs EST = 3	14.22	8.53	51.47

8.5.5.3 CPU Load with PWM = 20 kHz

Table 8-31. Minimum Implementation Executing from ROM, RAM and FLASH

F2802xF CPU = 60 MHz Available MIPs = 60 MIPs PWM = 20 kHz	CPU Utilization [%]	MIPs Used [MIPS]	MIPS Available [MIPS]
Rs Online Disabled, ISR vs CTRL = 1, CTRL vs EST = 1	86.27	51.76	8.24
CTRL vs EST = 2	66.4	39.84	20.16
CTRL vs EST = 3	59.77	35.86	24.14
ISR vs CTRL = 2, CTRL vs EST = 1	48.27	28.96	31.04
CTRL vs EST = 2	38.33	23	37
CTRL vs EST = 3	35	21	39
ISR vs CTRL = 3, CTRL vs EST = 1	35.6	21.36	38.64
CTRL vs EST = 2	28.97	17.38	42.62
CTRL vs EST = 3	26.7	16.02	43.98
Rs Online Enabled, ISR vs CTRL = 1,			
CTRL vs EST = 2	74.2	44.52	15.48
CTRL vs EST = 3	64.97	38.98	21.02
ISR vs CTRL = 2, CTRL vs EST = 1	56.03	33.62	26.38
CTRL vs EST = 2	42.23	25.34	34.66
CTRL vs EST = 3	37.6	22.56	37.44
ISR vs CTRL = 3, CTRL vs EST = 1	40.77	24.46	35.54
CTRL vs EST = 2	31.53	18.92	41.08
CTRL vs EST = 3	28.43	17.06	42.94

8.6 Digital and Analog Pins

8.6.1 Pin Utilization

Table 8-32 lists the pins used by InstaSPIN.

Table 8-32. Pin Utilization Per Motor

Pin Type	Pin Name	Pins Usage Per Motor	
		Min	Max
Digital	PWM1A	3 (Requires External Fault and External Complementary Mode with Dead Time)	7
	PWM1B (Optional)		
	PWM2A		
	PWM2B (Optional)		
	PWM3A		
	PWM3B (Optional)		
	TZ1 (Optional)		

Copyright © 2013–2019, Texas Instruments Incorporated

Table 8-32. Pin Utilization Per Motor (continued)

Pin Type	Pin Name	Pins Usage Per Motor	
		Min	Max
Analog	IA	5 (Only two currents and no VBUS ripple compensation)	7
	IB		
	IC (Optional)		
	VA		
	VB		
	VC		
	VBUS (Optional)		

8.6.2 F2805x Analog Front-End (AFE)

8.6.2.1 Consideration of AFE Module

In InstaSPIN applications, motor line current and phase voltage are required by the algorithm. Before these analog signals are sampled by the processor, all signals are processed by an analog circuit. The external analog circuits add component cost and increase board size. The 2805x series of processors addresses this issue by adding internal analog conditioning components for the motor feedback signals, called the analog front end (AFE).

For more detailed information about the 2805x device, see *TMS320x2805x Piccolo Technical Reference Manual* (SPRUHE5).

8.6.2.2 Routing Current Signals

Before addressing about the implementation and usage of the PGAs and comparators, it is recommended to consider how current feedback signals are routed from the shunt and then to the input of the PGA. When a shunt resistor is used to measure line current, its value must be small to reduce the amount of power dissipated in the shunt. Because the value is small, so is the resulting voltage drop across the shunt. There is a significant amount of current flowing through the shunt resistors. Copper traces that connect the shunts from the bottom of the power device and then to ground become a resistor in series with the shunt. The parasitic resistance that forms on the copper trace must be taken into consideration when measuring motor line currents with a shunt resistor.

The AFE can have up to three different grounds. The 2805x device has multiple groups of amplifier blocks. Each group of amplifiers has a different ground. M1 ground is used for the group of three PGAs that will feedback three-phase motor currents for this document. For systems with power factor correction, there is another single PGA and its ground is PFC ground. The fixed-gain amplifier block uses M2 ground for its reference and is used in this document for three motor voltage feedbacks.

Two options for the feedback of motor shunt current signals to the M1 PGA block of the AFE are discussed. The first option is to use only the internal op-amps for the current feedback as shown in Figure 8-10. All three op-amps share the same ground for the inverting input and therefore a differential signal of the shunt current cannot be created. With single-ended signals, careful layout must be done when grounding the shunts to reduce the amount of differing trace resistance between shunts. It is advised to have the shunt grounds as close together as possible. A trace must run from the point that the shunts come together to the M1gnd pin of the integrated circuit. Because common mode noise can be added to the amplifier, the M1gnd pin and PGA inputs must be made as short as possible. The three phase current traces must be routed as close to the M1gnd trace as possible to reduce the size of the Faraday loop. The Faraday loop is created around the phase current trace that starts from the top of the shunt to the IC and then back on the M1gnd trace to the bottom of the shunt, through the shunt and back to the top of the shunt.

Copyright © 2013–2019, Texas Instruments Incorporated

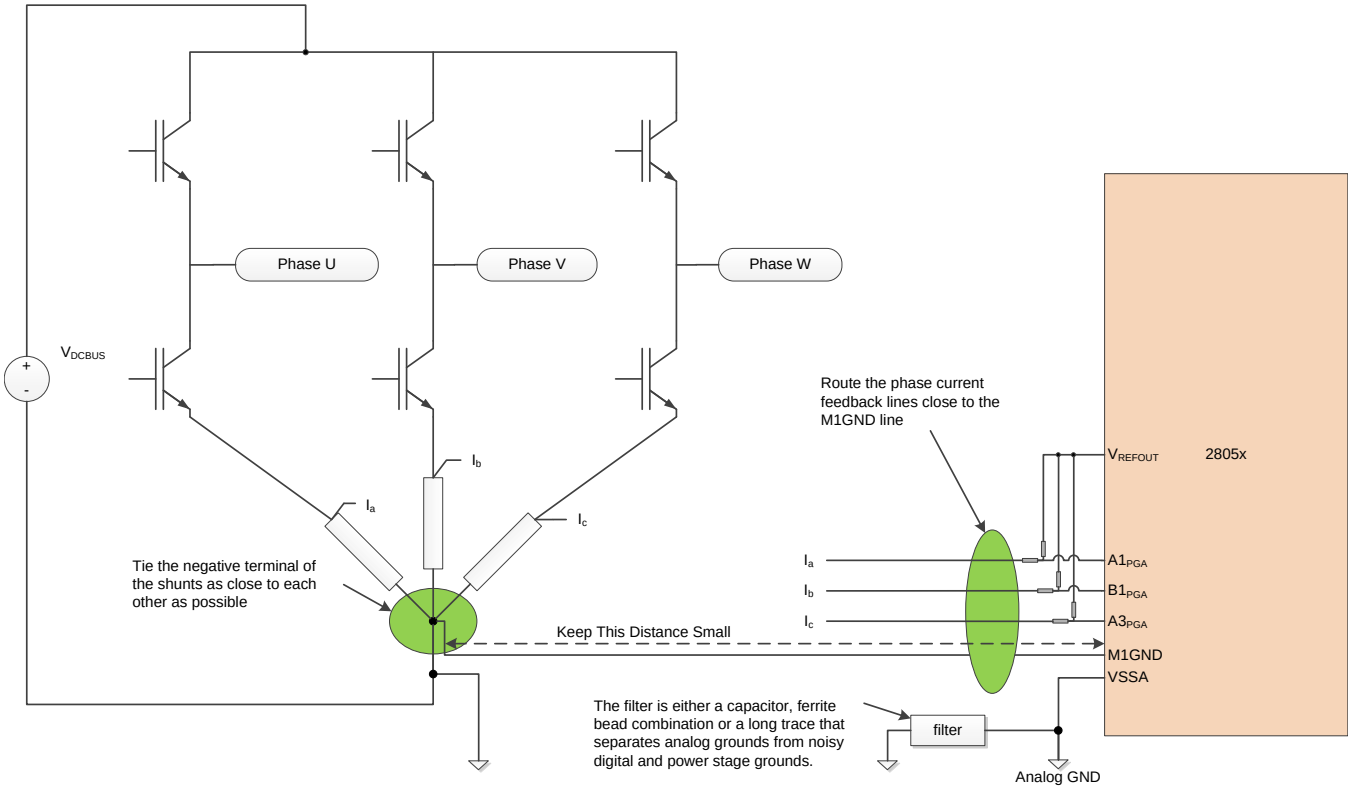


Figure 8-10. Current Signal Routing Directly to PGAs With Single-Ended Connections

The second, and most noise immune option, is to use external op-amps in a differential amplifier configuration. A true Kelvin connection can feedback directly to the differential amplifier, and then the output of the differential amplifier is sent into the PGA input. Figure 8-11 shows a typical layout when using external differential op-amps. Since the Kelvin connection has low impedance and is a truly differential signal, it provides excellent noise immunity. The external op-amp circuit converts the differential circuit into a single-ended output. The single-ended output is more susceptible to noise and therefore it is best to place the output of the op-amp as close to the AFE input of the processor.

Copyright © 2013–2019, Texas Instruments Incorporated

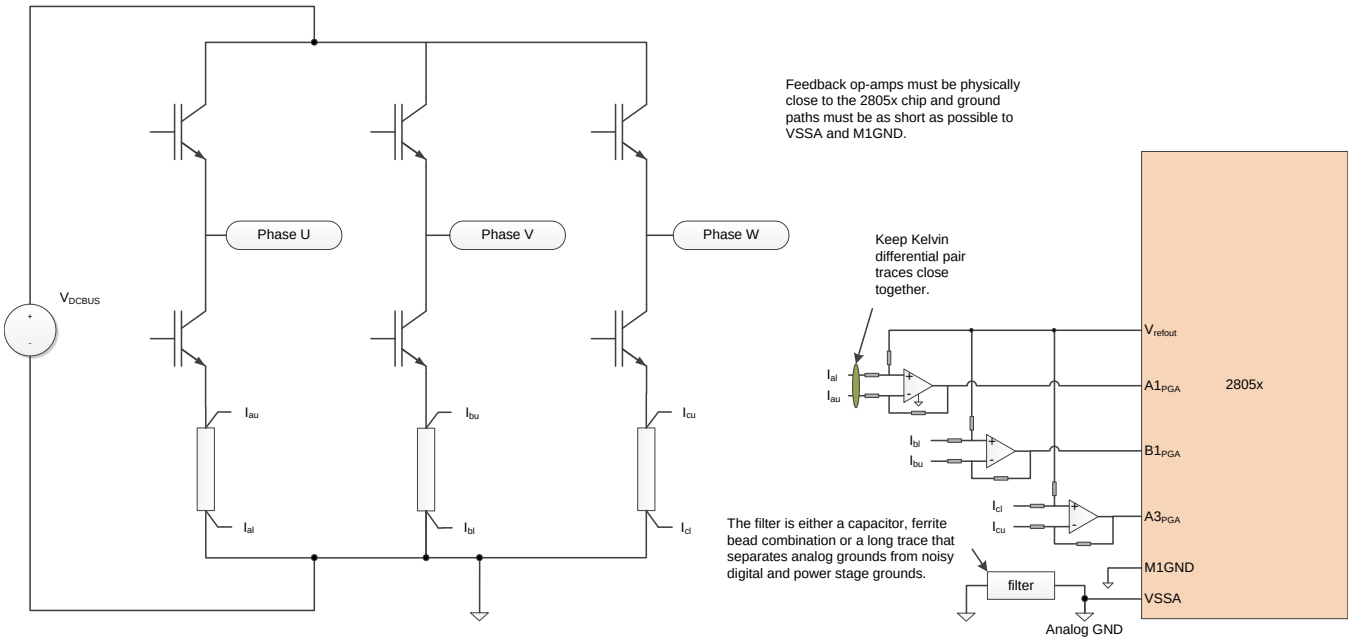


Figure 8-11. Feedback of Phase Currents Using External Differential Amplifiers

Why use the PGAs when external amplifiers are already being used? One case would be if many different current rated motors are powered with the same inverter. Amplification of the current signal can be adjusted to best suit the motor size that is controlled. The output of the PGA block is the input of the comparator windows. The PGA still needs to be connected to enable the use of the fault detection circuitry.

8.6.2.3 Voltage Reference Connection

Current can flow through the shunt in both positive and negative directions which will create both a positive and negative voltage that is fed back to the shunt amplifier circuit. Most cost-effective motor inverters do not have both positive and negative power supplies that can handle this bipolar signal. A bipolar current signal is brought into an amplifier that will only be effective from zero to the positive voltage supply. To allow the unipolar op-amp circuit to measure a bipolar signal, a voltage reference is summed into the non-inverting side of the current feedback op-amps. The AFE of the 2805x device contains a 6-bit DAC with a voltage follower for providing an output reference for this reason. A circuit configuration that can use a voltage reference to measure the bipolar current signal is shown in Figure 8-12.

Copyright © 2013–2019, Texas Instruments Incorporated

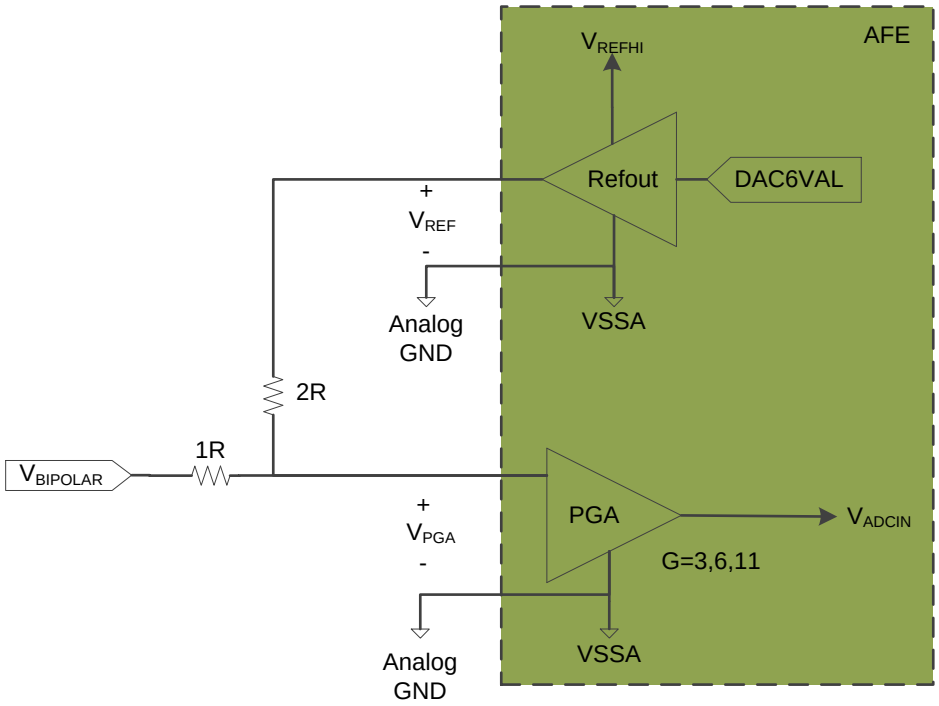


Figure 8-12. Using the AFE's Built-In Voltage Reference For Measuring a Bipolar Signal

Equation 24 shows how to calculate the voltage at V_{PGA}. As an example, set the PGA gain = 3. V_{ADCIN} will be $2V_{BIPOLAR} + V_{REF}$. Assume the system's V_{REFHI} is 3.3 V. To allow for maximum voltage swing in both directions, V_{REF} is set to 1.65 V. Now the maximum peak $V_{BIPOLAR}$ voltage that can be measure is ±0.825 V.

$$V_{PGA}=\frac{2R\cdot(V_{BIPOLAR}-V_{REF})}{(1R+2R)}+V_{REF}=\frac{2}{3}V_{BIPOLAR}+\frac{1}{3}V_{REF} \quad (24)$$

Suppose the same hardware is used and a higher resolution is required. The PGA gain = 6. V_{ADCIN} is $4V_{BIPOLAR} + 2V_{REF}$. V_{REF} must be adjusted to be 0.825 V. The maximum peak $V_{BIPOLAR}$ voltage that can be measured is ±0.4125 V.

The voltage reference output is adjusted by a 6-bit DAC. The VREFOUTCTL register controls the DAC's voltage output by Equation 25 below.

$$V_{REF}=\frac{V_{REFHI}\cdot(VREFOUTCTL_{DACVAL}+1)}{64} \quad (25)$$

8.6.2.4 Routing Voltage Signals

In sinusoidal motor control drives, the voltage signals vary slowly when compared to current signals. Therefore, larger hardware filters can be applied to the voltage feedback signal which helps to make it less susceptible to noise. Voltage signals are unipolar, so no special circuit and reference have to be used. Lower voltage motors (under 400 V_{DCBUS}) typically only require resistor dividers with a capacitive low-pass filter. For a brushless DC motor control the voltage needs as little phase shift as possible and, therefore, the low-pass filtering depends on the maximum speed achieved by the motor. The only critical layout of voltage feedback signals is that the low-pass filter capacitor must be located as close to the AFE or A/D input pin as possible.

Copyright © 2013–2019, Texas Instruments Incorporated

Real-Time Structure

Topic **Page**

9.1 InstaSPIN Software Execution Clock Tree..... 359
9.2 Decimating in Software for Real-Time Scheduling 362
9.3 Decimating in Hardware 377

Copyright © 2013–2019, Texas Instruments Incorporated

9.1 InstaSPIN Software Execution Clock Tree

There are several clock decimations when using InstaSPIN. The first clock that needs to be considered is the interrupt clock, which is generated by a peripheral clocked with the CPU clock. Typically, the interrupt service routine (ISR) is triggered by the end of conversion (EOC) of the ADC. This conversion is triggered by the PWM module.

First of all, let us review how the PWM frequency is configured based on user's parameters from user.h. Starting from the CPU clock, user defines, in MHz, what the CPU clock rate is:

```
//! \brief Defines the system clock frequency, MHz (6xF and 6xM devices)
//!
#define USER_SYSTEM_FREQ_MHz       (90)

//! \brief Defines the system clock frequency, MHz (2xF devices)
//!
#define USER_SYSTEM_FREQ_MHz       (60)
```

Then, the PWM frequency in kHz is defined, which results in the interrupt frequency.

```
//! \brief Defines the Pulse Width Modulation (PWM) frequency, kHz
//!
#define USER_PWM_FREQ_kHz          (15.0)

//! \brief Defines the Pulse Width Modulation (PWM) period, usec
//!
#define USER_PWM_PERIOD_usec          (1000.0/USER_PWM_FREQ_kHz)

//! \brief Defines the Interrupt Service Routine (ISR) frequency, Hz
//!
#define USER_ISR_FREQ_Hz           (USER_PWM_FREQ_kHz *    1000.0)

//! \brief Defines the Interrupt Service Routine (ISR) period, usec
//!
#define USER_ISR_PERIOD_usec       USER_PWM_PERIOD_usec
```

So far, the CPU clock sets the PWM frequency, which also sets the frequency of the ISR. Now the ISR is actually not triggered by the PWM timer itself, but it is triggered by the end of conversion of the ADC which was started by the PWM timer.

Figure 9-1 is a timing diagram of the clocks from the CPU all the way to the ISR generation.

Copyright © 2013–2019, Texas Instruments Incorporated

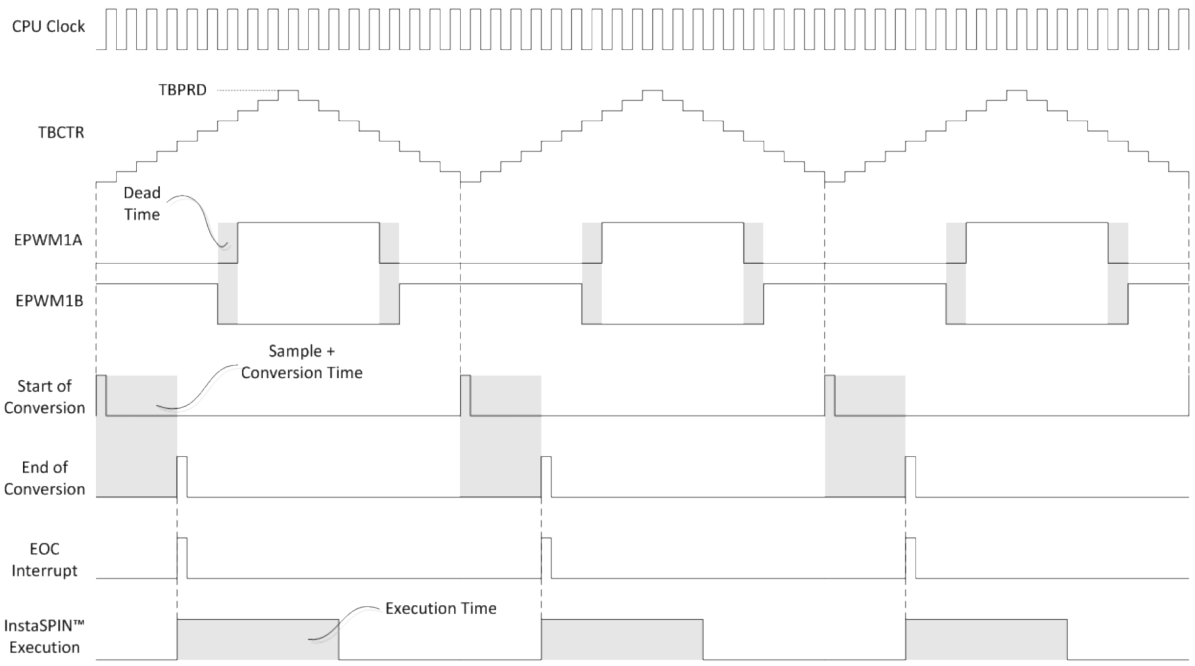


Figure 9-1. Clock Timing - CPU to ISR Generation

This timing diagram represents the interrupt triggering scenario that Texas Instruments delivers for InstaSPIN software package because this is the safest way the conversions will be ready when fetching the interrupt. Other scenarios might be considered by the user, such as ADC early interrupt, PWM interrupt or CPU timer interrupt. The only requirement is that those interrupts are generated at a fixed period.

Note that the execution time can be measured in several different ways. Here are some examples on how to measure execution time:

- *GPIO Toggle.* An easy but not so accurate execution time measurement is to simply set a GPIO at the very beginning of the code to be measured and the clearing the same GPIO right after it. This method is very graphical since it can be displayed in a scope, and sine the interrupts are periodic, it will give a good trigger for the scope to measure execution time. The time can then be converted to CPU cycles if needed. One thing to consider using this method though is the time it takes for a particular architecture to set and clear a GPIO, as well as the interrupt fetch and return times.
- *CPU Timer Capture.* A much more accurate execution time measurement is with a CPU timer. This can be done by running a CPU timer at the same clock as the CPU clock, with no prescaler or postcaler, and then read the timer after the code has been executed. This will give us the CPU cycles needed to execute the code of interest.

From the InstaSPIN execution timing, there are several decimation values, also known as tick rates that allow different execution clock rates for different portions of control code within InstaSPIN. The following tick rates are available for InstaSPIN:

```
//! \brief Defines the number of pwm clock ticks per isr clock tick
//!        Note: Valid values are 1, 2 or 3 only
#define USER_NUM_PWM_TICKS_PER_ISR_TICK       (1)

//! \brief Defines the number of isr ticks per controller clock tick
//!
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK      (1)

//! \brief Defines the number of controller clock ticks per current controller clock tick
//!
#define USER_NUM_CTRL_TICKS_PER_CURRENT_TICK  (1)
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
//! \brief Defines the number of controller clock ticks per estimator clock tick
//!
#define USER_NUM_CTRL_TICKS_PER_EST_TICK      (1)

//! \brief Defines the number of controller clock ticks per speed controller clock tick
//!
#define USER_NUM_CTRL_TICKS_PER_SPEED_TICK  (10)

//! \brief Defines the number of controller clock ticks per trajectory clock tick
//!
#define USER_NUM_CTRL_TICKS_PER_TRAJ_TICK   (10)
```

In order to show all these tick rates, see Figure 9-2. The following acronyms are defined for easier reference within the software execution clock tree diagram:

USER_NUM_PWM_TICKS_PER_ISR_TICK -> /ETPS

USER_NUM_ISR_TICKS_PER_CTRL_TICK -> /ISRvsCTRL

USER_NUM_CTRL_TICKS_PER_CURRENT_TICK -> /CTRLvsCURRENT

USER_NUM_CTRL_TICKS_PER_EST_TICK -> /CTRLvsEST

USER_NUM_CTRL_TICKS_PER_SPEED_TICK -> /CTRLvsSPEED

USER_NUM_CTRL_TICKS_PER_TRAJ_TICK -> /CTRLvsTRAJ

In the case of the F2806x device, the software execution clock tree starts with a SYSCLKOUT of 90 MHz, and everything else is decimated from that clock. For the F2805x amd F2802x devices, the maximum frequency is 60 MHz, instead of 90 MHz.

After a clock prescaler, which is set to one by default, to get the best resolution of the PWM generator, we have the TBPRD register (see Figure 9-2). This register has a period value so that the output creates the PWM frequency.

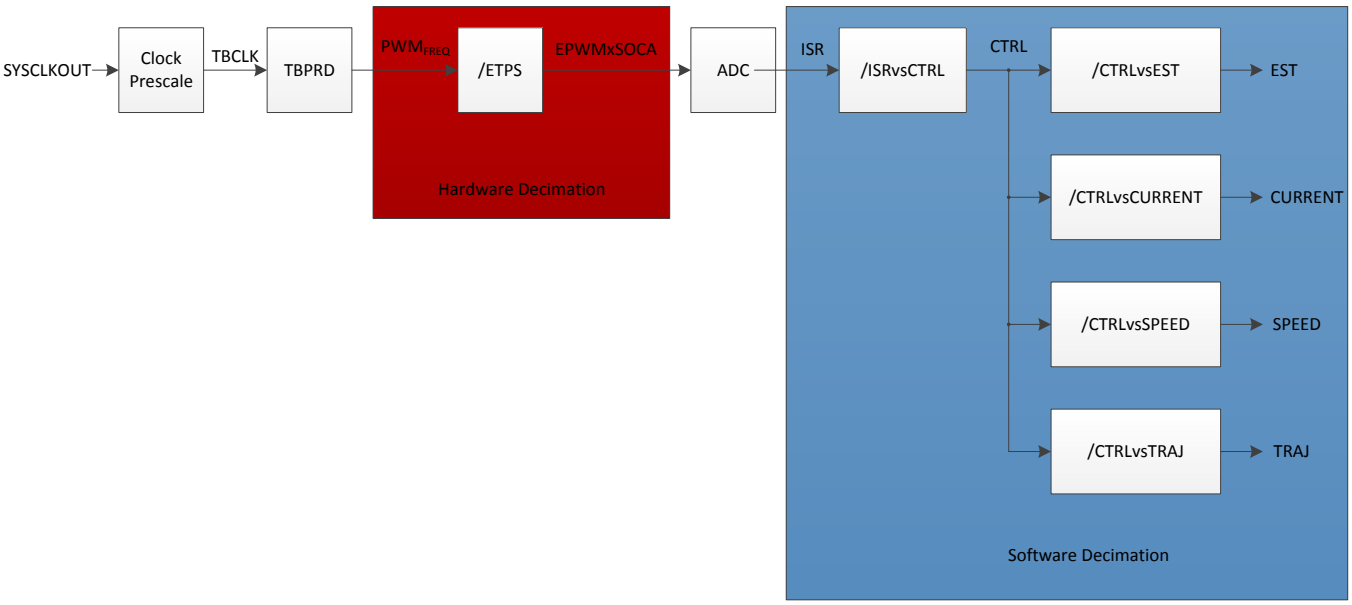


Figure 9-2. Software Execution Clock Tree

The first decimation of the software execution clock tree is in hardware. Depending on the value of the ETPS register (Event Trigger Prescale Register), the PWM frequency can be divided 1, 2 or 3 times. This is useful when the ADC start of conversion signal needs to be triggered every PWM cycle, or every 2 or every 3 PWM cycles. This hardware decimation is controlled by the USER_NUM_PWM_TICKS_PER_ISR_TICK definition in user.h.

Copyright © 2013–2019, Texas Instruments Incorporated

The second decimation block is done in software and will be explained in detail in the following section.

9.2 Decimating in Software for Real-Time Scheduling

The highlighted software tick rates shown in Figure 9-3 are used to decimate the execution of InstaSPIN in software, also known as real-time scheduling tick rates.

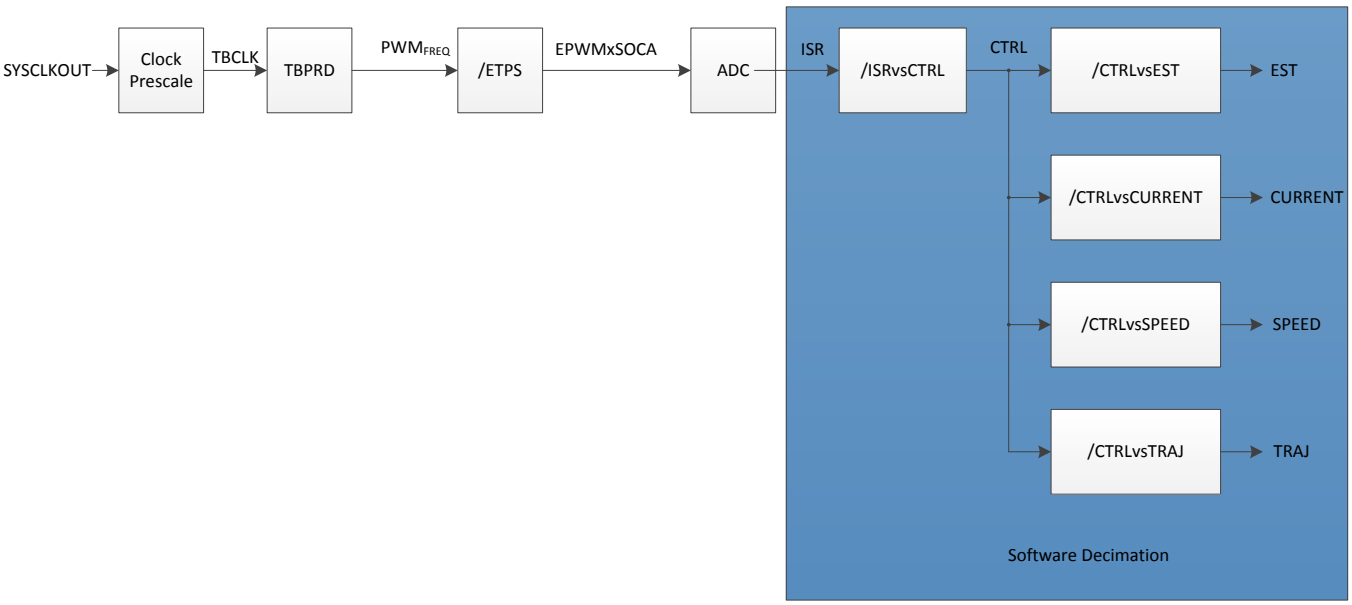


Figure 9-3. Real-Time Scheduling Tick Rates

9.2.1 USER_NUM_ISR_TICKS_PER_CTRL_TICK

The first tick rate defines the main rate at which InstaSPIN as a whole will be executed from the end of conversion ISR. When this tick rate is greater than one, every time InstaSPIN is executed there will be a check in an internal counter, and if this counter hasn't reached the tick rate value, it will return from InstaSPIN execution. There is no code executed inside InstaSPIN library other than the check of this counter. Figure 9-4 shows how the tick counter is checked in the ISR.

Copyright © 2013–2019, Texas Instruments Incorporated

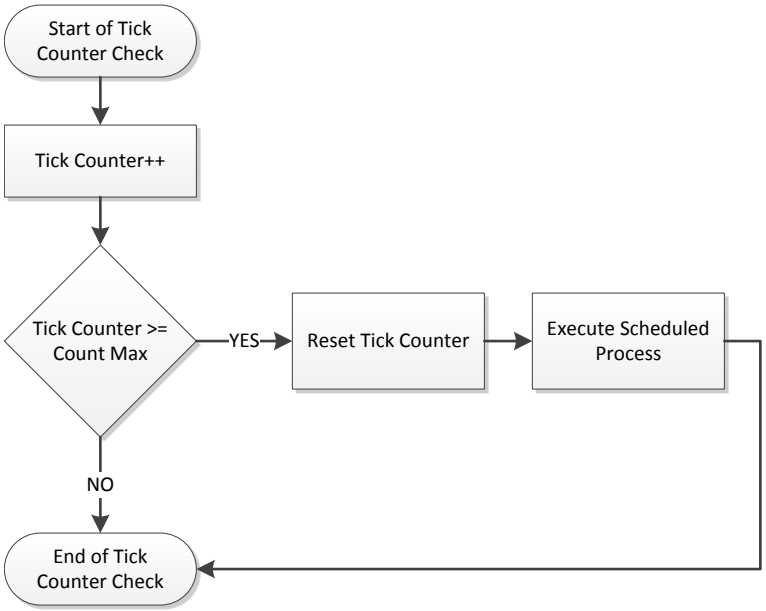


Figure 9-4. Tick Counter Flowchart

Figure 9-5 shows InstaSPIN execution with a tick rate of 2.

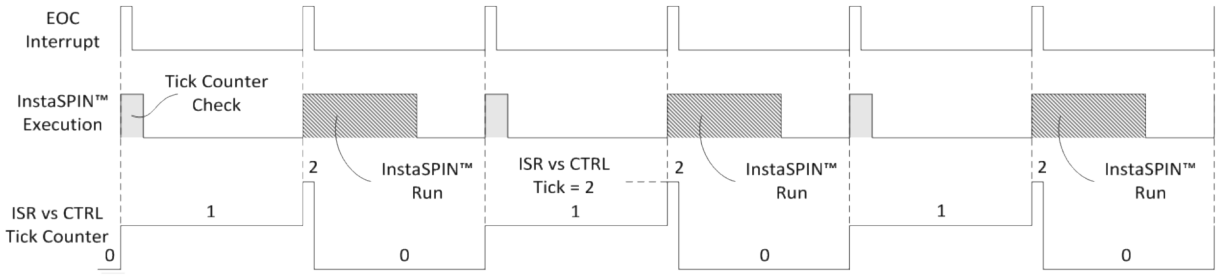


Figure 9-5. InstaSPIN Timing

Figure 9-6 can also be represented as a software execution clock tree as follows.

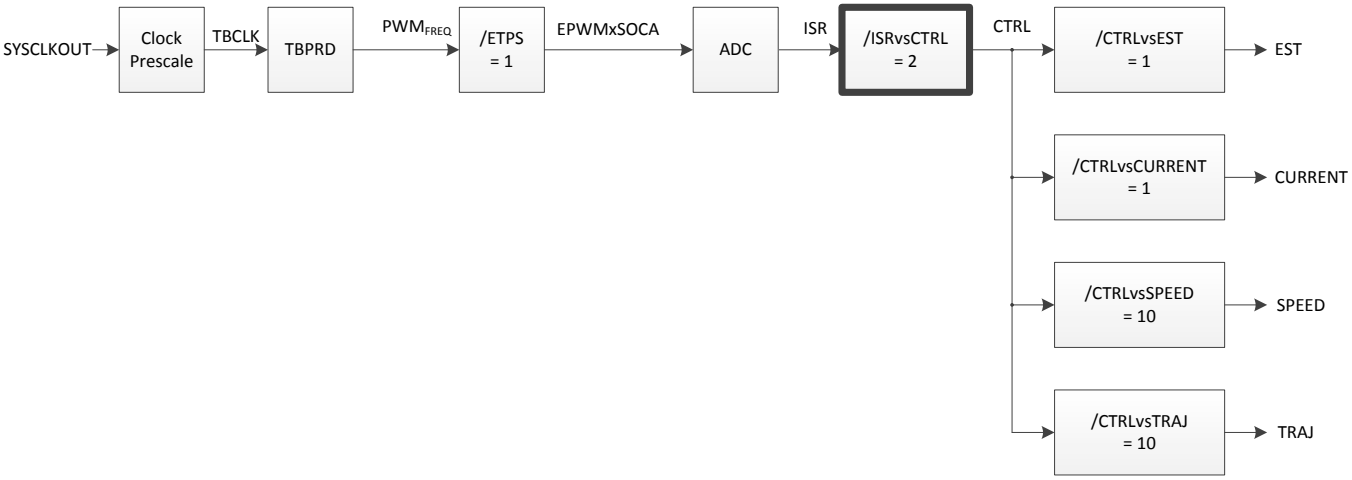


Figure 9-6. InstaSPIN Timing Software Execution Clock Tree

Copyright © 2013–2019, Texas Instruments Incorporated

Notice how the highlighted block has a value of two, causing a divide by two in the software execution clock tree.

There are two main reasons why a tick rate from the interrupt to the controller might be higher than 1:

- The first reason is to reduce the CPU usage.
- The second one is to allow a higher PWM frequency and reduce the tick rate so that InstaSPIN can still be executed at higher frequencies.

For example, if the PWM frequency is 50 kHz, if no hardware decimation is used (which will be discussed later in this document) the end of conversion ISR is at the same rate, 50 kHz. There needs to be enough time in 1/50 kHz = 20 µs to execute all the functions. If the functions within the ISR take 30 µs, then:

Execution time > ISR Period → 30 µs > 20 µs

This will lead to interrupt overrun, causing ADC samples to be overwritten, and control timing will also be affected.

In cases where the interrupt is shorter than the execution time, it is safe to use different ISR to CTRL tick rates if the following guidelines are taken into consideration.

Verify there is enough time in the interrupt to execute InstaSPIN.

This is because when executing InstaSPIN in the interrupt service routine there has to be enough time to avoid conversion overrun. For example, if an ISR hasn't been serviced, and a second one comes in, the first one was completely lost, and the timing is affected. A good example is shown in Figure 9-7, when the ISR has enough time so that InstaSPIN completes execution with no ISR overrun.

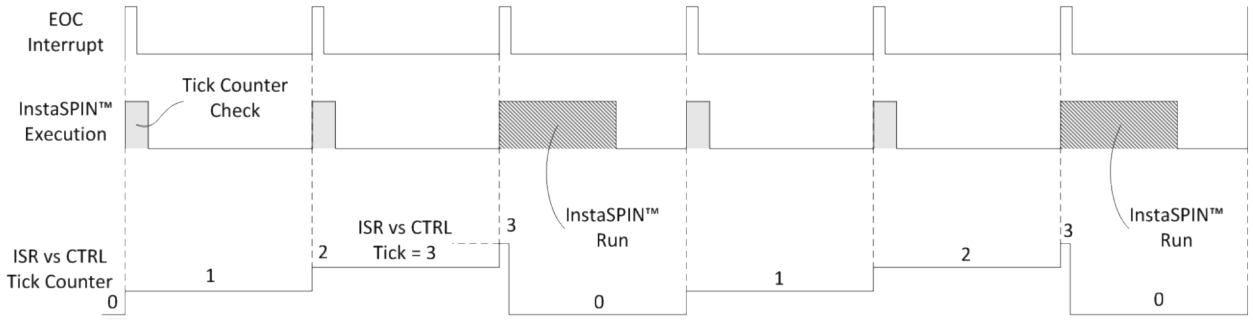


Figure 9-7. InstaSPIN Timing Completes Execution with No ISR Overrun

Figure 9-8 shows the software execution clock tree representation of this timing diagram.

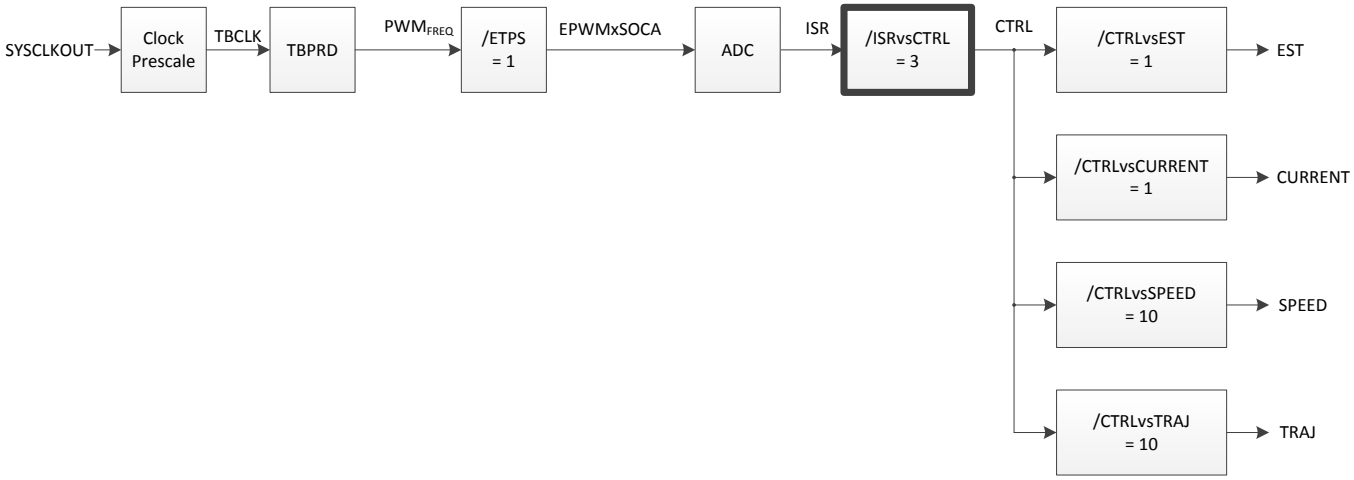


Figure 9-8. InstaSPIN Timing Software Execution Clock Tree - No ISR Overrun

Copyright © 2013–2019, Texas Instruments Incorporated

Another example using decimation rates with a higher PWM frequency, leads to a shorter ISR, but even though one ISR is not serviced right away, it eventually does with no ISR overrun (Figure 9-9). This would work ok as well, without affecting performance.

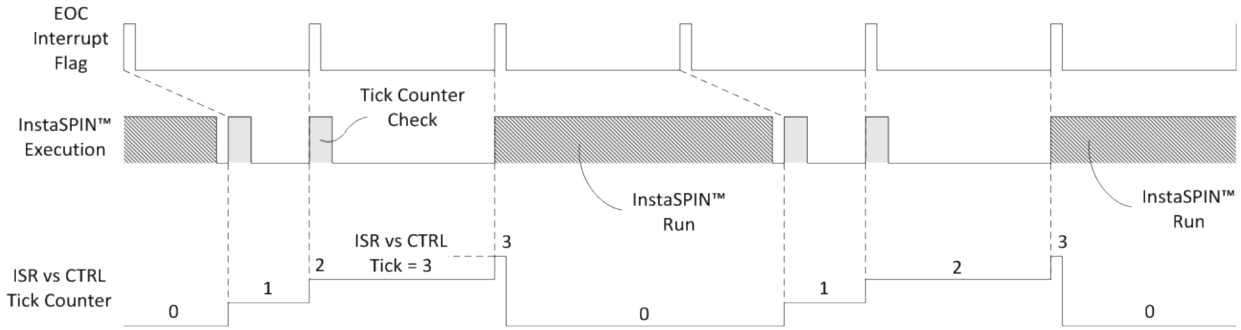


Figure 9-9. InstaSPIN Timing with a Higher PWM Frequency

On the other hand, if the PWM frequency is setup too high, the end of conversion interrupt might be overrun by a second interrupt. An overrun condition is undesirable since it will cause a complete set of ADC samples to be lost. In addition to that, an interrupt overrun causes a complete interrupt to be missed and as a result of that, the timing of the InstaSPIN state machine will be wrong, since it depends on the periodicity of the interrupts. Not keeping a good timing schedule in the InstaSPIN library causes issues such as angle estimation not being accurate, speed estimation being off, speed and current controllers not performing as desired, just to list a few. The following example shows an interrupt overrun condition. Keep in mind that InstaSPIN execution is the same, what we are changing is the PWM frequency (hence the end of conversion interrupt frequency) and the decimation number. The first example of interrupt overrun is without decimation, so all of InstaSPIN executed at every interrupt (Figure 9-10).

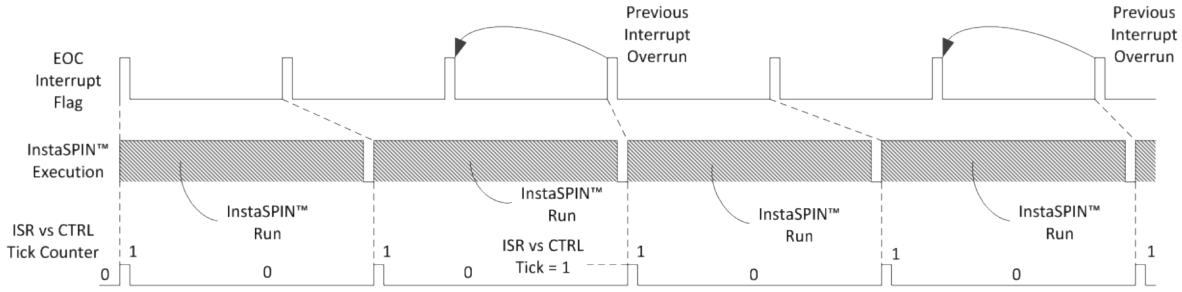


Figure 9-10. Interrupt Overrun without Decimation Timing

Figure 9-11 shows the software execution clock tree values of this timing diagram.

Copyright © 2013–2019, Texas Instruments Incorporated

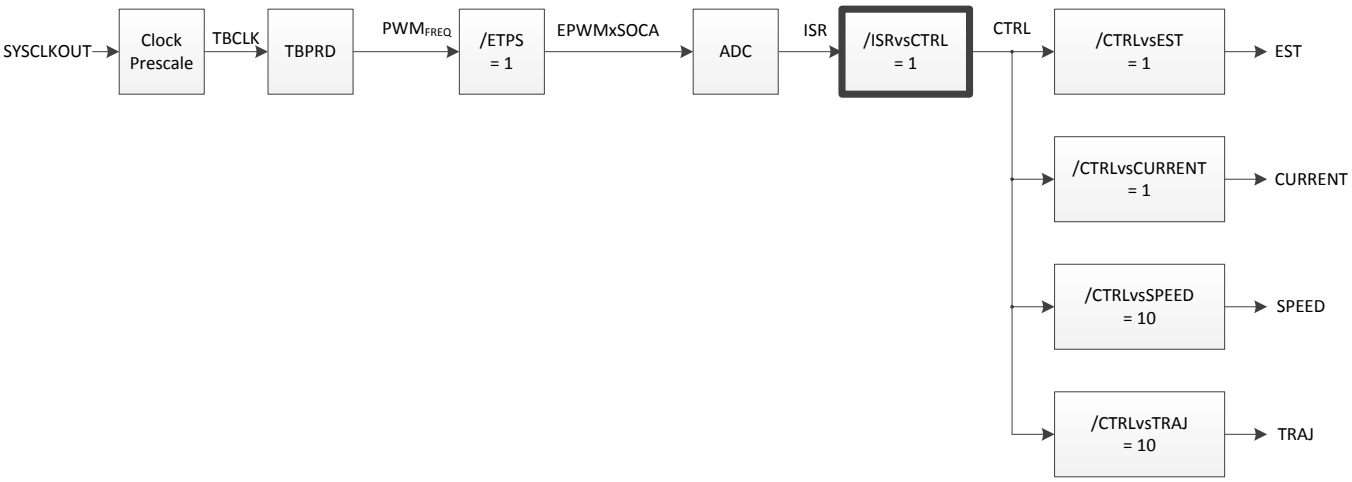


Figure 9-11. Interrupt Overrun without Decimation Software Execution Clock Tree

Notice how after a few interrupts a complete interrupt is overrun. The problem with this is that now the timing is shifted, so internally in the InstaSPIN state machine and controllers, timing is now slower than it really is.

Another example is using decimation rates. In Figure 9-12 a decimation rate of 2 is used, and as you can see, even using decimation rates we have the limitation of interrupt overrun.

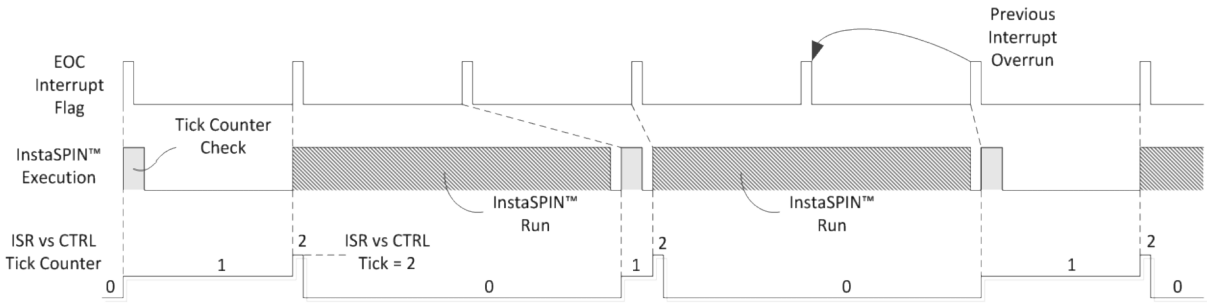


Figure 9-12. Interrupt Overrun with Decimation Timing

Figure 9-13 shows the software execution clock tree values of this timing diagram.

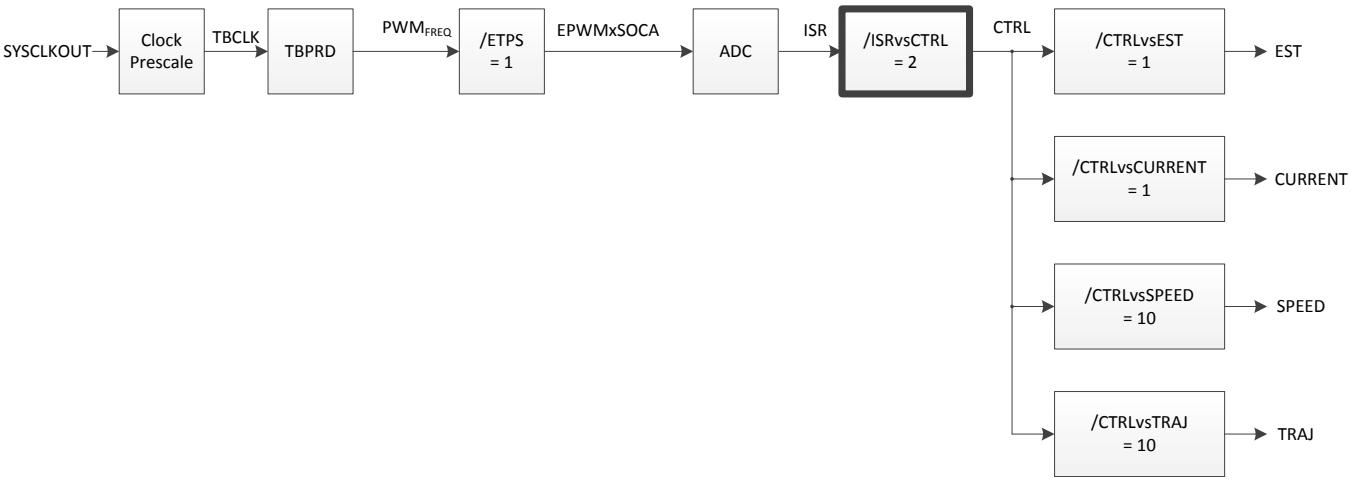


Figure 9-13. Interrupt Overrun with Decimation Software Execution Clock Tree

Copyright © 2013–2019, Texas Instruments Incorporated

Set frequency of InstaSPIN to at least 10 times the electrical frequency of the motor.

The second aspect to be considered when setting PWM frequency and ISR vs. CTRL tick rate is the number of InstaSPIN runs versus the electrical frequency of the motor. This is because when running in a closed loop system, where field oriented control depends on an electrical angle, there should be enough estimated angle updates per electrical cycle to keep the field properly oriented. An analogy of this requirement is when an AC signal needs to be digitally sampled. This is related to Nyquist frequency, where a frequency just above the sampled frequency is enough to avoid aliasing. In a field oriented control system Nyquist frequency is not enough to provide an efficient motor control. The recommended InstaSPIN run rate is at least 10 times of the electrical frequency of the motor. In order to know the electrical frequency of a motor, i.e. for a Permanent Magnet Synchronous Motor (PMSM) we need to know the speed and the number of poles. For example:

Pole Pairs: 4

Speed: 7500 RPM

Electrical Frequency: Speed in RPM * Pole Pairs / 60 = 7500 * 4 / 60 = 500 Hz

Minimum Recommended InstaSPIN run rate = 10 * Electrical Frequency = 5000 Hz

In this example, we have chosen an ISR vs. CTRL tick rate of 3, resulting in an ISR frequency of 5000 * 3 = 15000 Hz.

Figure 9-14 shows the resulting waveforms from this example.

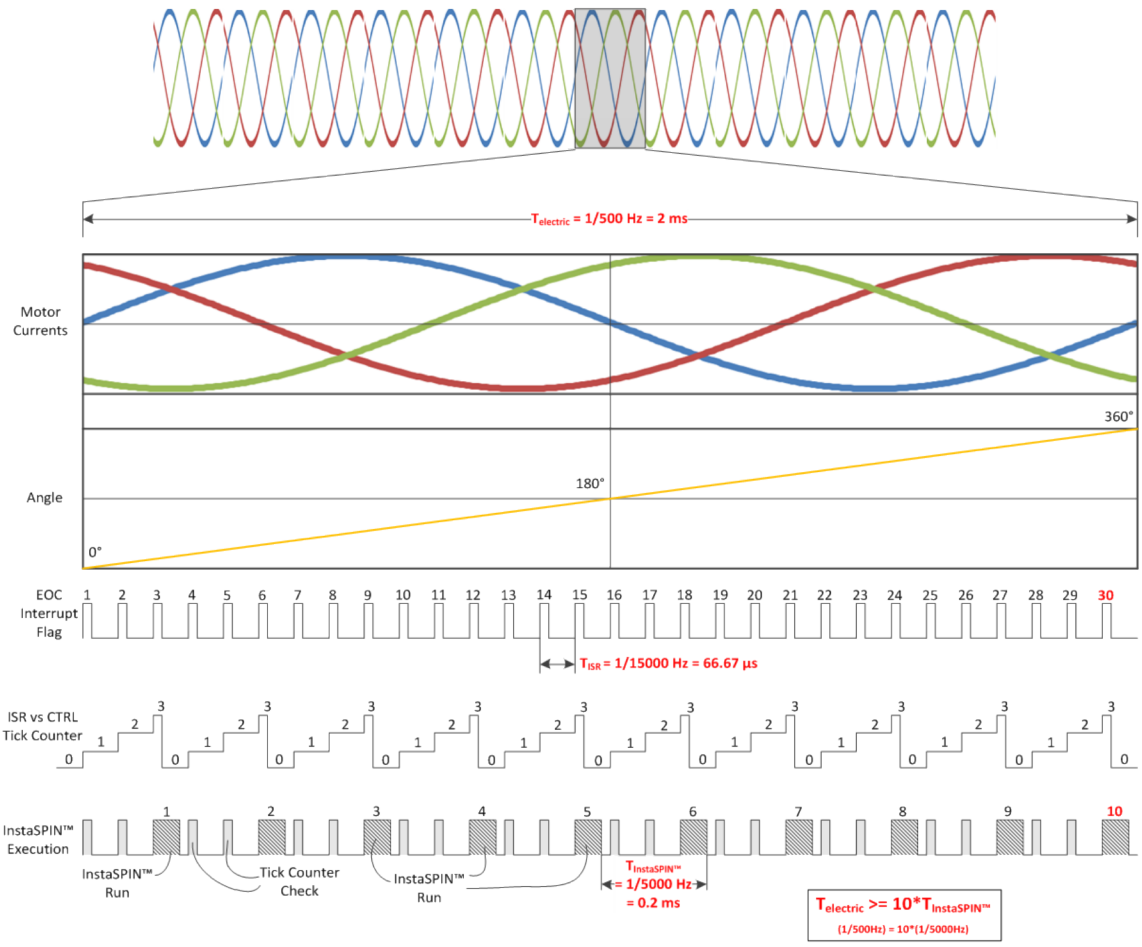


Figure 9-14. ISR Frequency Waveforms

Figure 9-15 shows the software execution clock tree numbers for this example.

Copyright © 2013–2019, Texas Instruments Incorporated

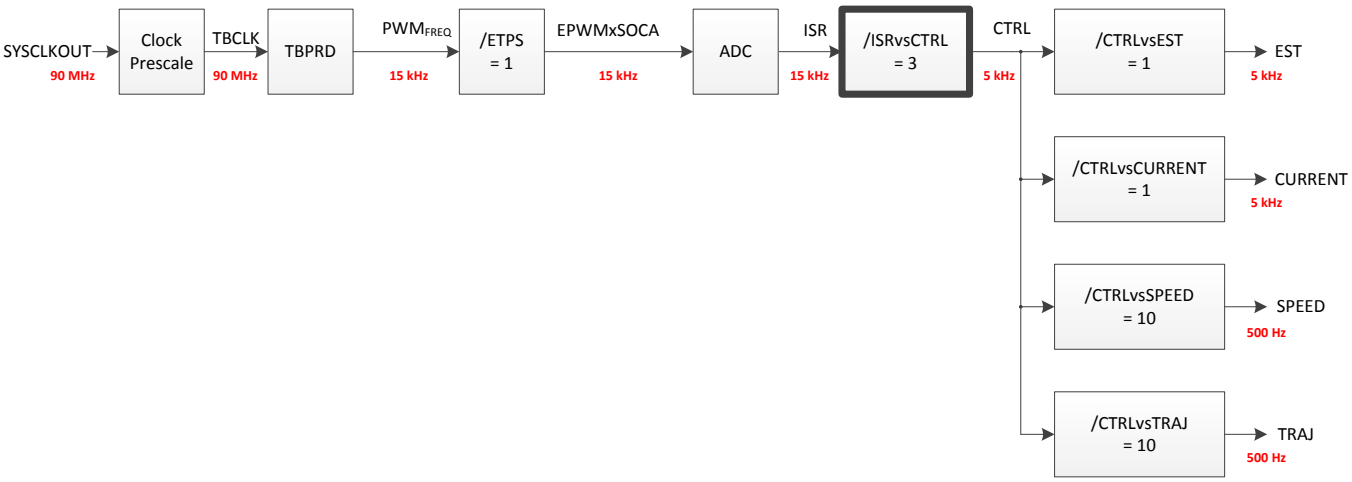


Figure 9-15. Software Execution Clock Tree for ISR Waveforms

9.2.2 USER_NUM_CTRL_TICKS_PER_CURRENT_TICK

The second tick rate to be discussed is the controller tick per current tick. This tick rate is used to slow down the current controllers with respect to the InstaSPIN execution rate. This tick rate only reduces the rate at which the current controllers are executed, which doesn't really help alleviate the CPU loading since there are only two PI controllers. It does reduce the current control performance though, so it is recommended to keep this tick rate equal to one, which means that the current controllers will be executed at the same rate as InstaSPIN execution. In order to show an example of how this tick rate can be used, consider the Figure 9-16. Also in this example we have chosen an ISR tick per CTRL tick rate of 3 to show how the CTRL per current tick is cascaded from the first tick rate.

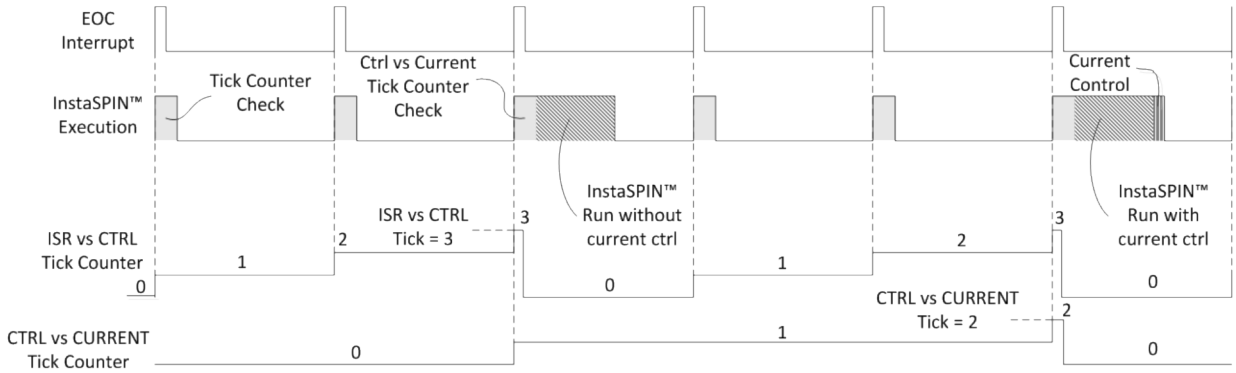


Figure 9-16. Tick Rate Timing

Figure 9-17 shows the values for this example in highlighted boxes.

Copyright © 2013–2019, Texas Instruments Incorporated

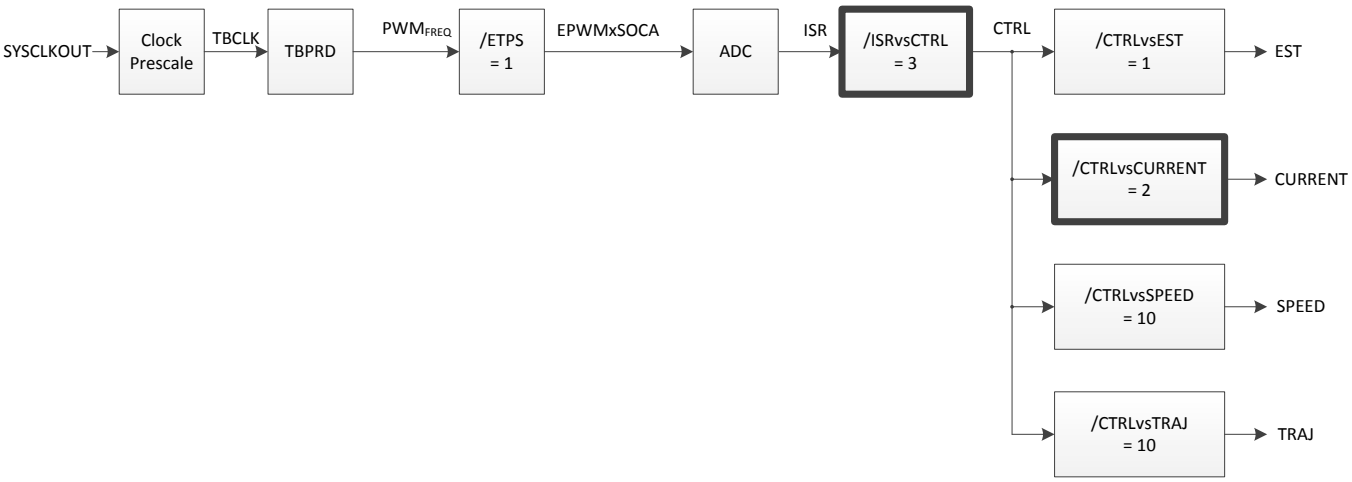


Figure 9-17. Tick Rate Software Execution Clock Tree

9.2.3 USER_NUM_CTRL_TICKS_PER_EST_TICK

The third decimation rate is to execute the estimator inside of InstaSPIN, also known as the FAST™ algorithm. This is one of the most popular tick rates available in InstaSPIN since it decimates the most time consuming part of InstaSPIN, which is the FAST estimator. As shown in the previous tick rate that decimates the current controllers, this tick rate decimates the estimator execution. To show an example of how this is cascaded from the InstaSPIN execution clock, consider Figure 9-18. It shows an ISR per CTRL tick rate of 1, a CTRL per CURRENT tick rate of 2, and a CTRL per EST tick rate of 2 as well. This shows how several tick rates can be combined to achieve a desired CPU bandwidth, and it also shows dependencies of other clocks within InstaSPIN.

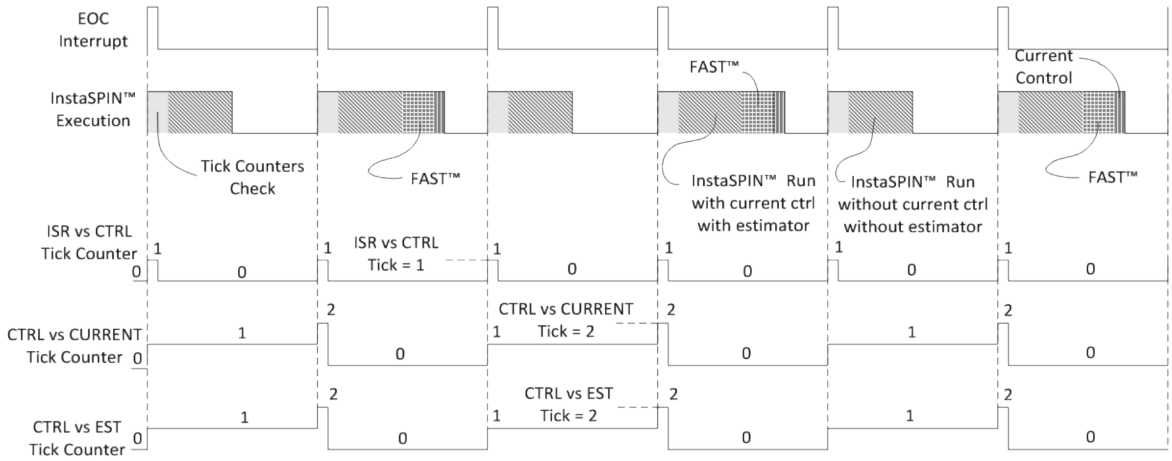


Figure 9-18. FAST Estimator Tick Rate Timing

Figure 9-19 represents the values of this timing diagram in highlighted boxes.

Copyright © 2013–2019, Texas Instruments Incorporated

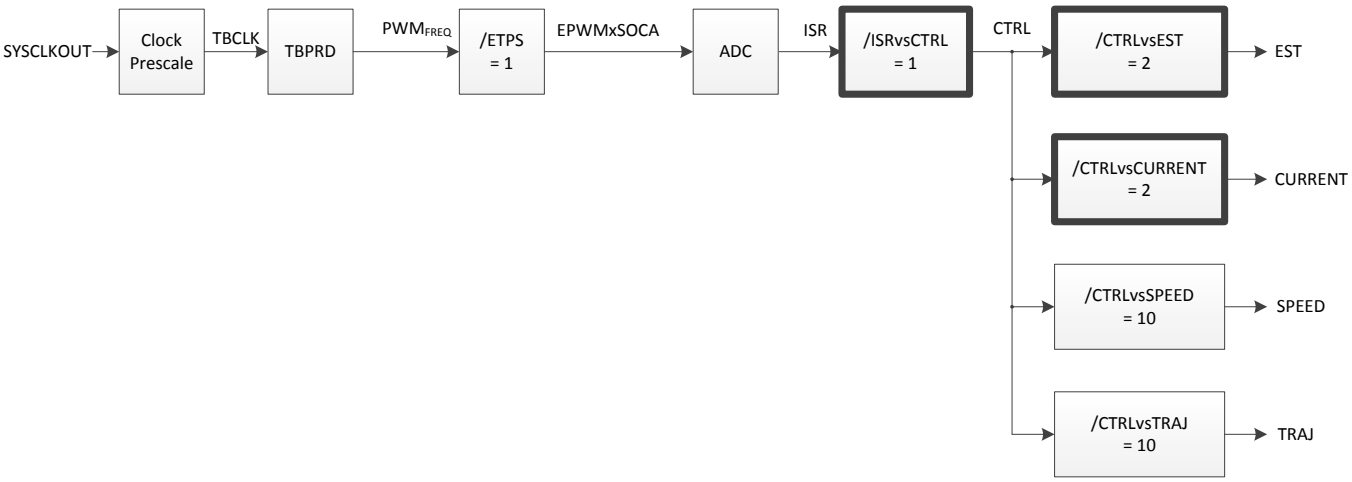


Figure 9-19. FAST Estimator Tick Rate Software Execution Clock Tree

9.2.4 Practical Example

A case example is shown next, where the CPU loading is restricted by the application and the PWM frequency requirement is fixed. For example, consider the following parameters for our case study:

InstaSPIN Execution time only checking the tick rate counters: 2.7 µs

FAST estimator execution time: 12.9 µs

InstaSPIN Execution time without the FAST estimator: 14.2 µs

Total of InstaSPIN with FAST: 27.1 µs = 12.9 µs + 14.2 µs

PWM Frequency requirements: 50 kHz (TISR = 20 µs)

A typical example where such a high PWM frequency is needed is when the motor has a very low inductance. Having a low PWM frequency would create undesirable current ripple due to the low inductance. A solution for these applications is to have a higher PWM frequency. In the example a 50 kHz PWM frequency is required.

The first configuration we should try is with the ISR to CTRL, CTRL to CURRENT and CTRL to EST all to 1, so that we get the best performance. If we try these tick rates to one, we get Figure 9-20.

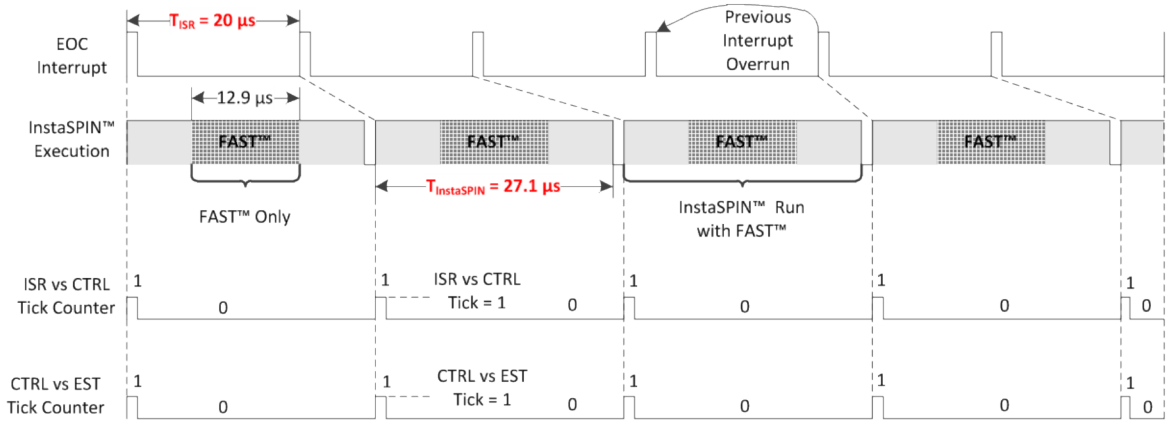


Figure 9-20. Tick Rates Timing

As can be seen from the timing diagram, the interrupt is shorter than what needs to be executed, so this will lead to interrupt overrun, hence undesirable behavior.

Copyright © 2013–2019, Texas Instruments Incorporated

$T_{ISR} < T_{InstaSPIN} \rightarrow 20\ \mu s < 27.1\ \mu s \rightarrow$ this leads to ISR overrun, hence unexpected InstaSPIN results

Figure 9-21 represents the values of this timing diagram in highlighted boxes.

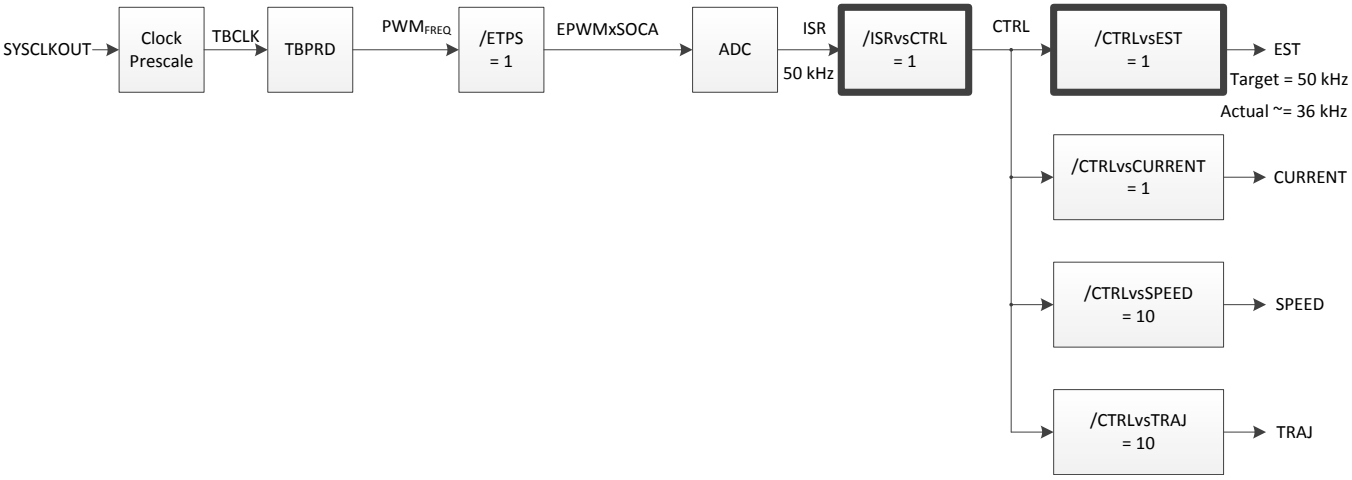


Figure 9-21. Tick Rates Software Execution Clock Tree

Notice how the interrupt time is not enough to execute InstaSPIN at the same rate and it never catches up with execution. In fact, after a few interrupts there are missing interrupts, which will cause unexpected results. A solution to this problem is to use the tick rates so that the FAST estimator runs at a lower rate compared to the rest of InstaSPIN. In Figure 9-22, let's see if we can solve the overrun problem with a CTRL vs. EST tick rate of 2.

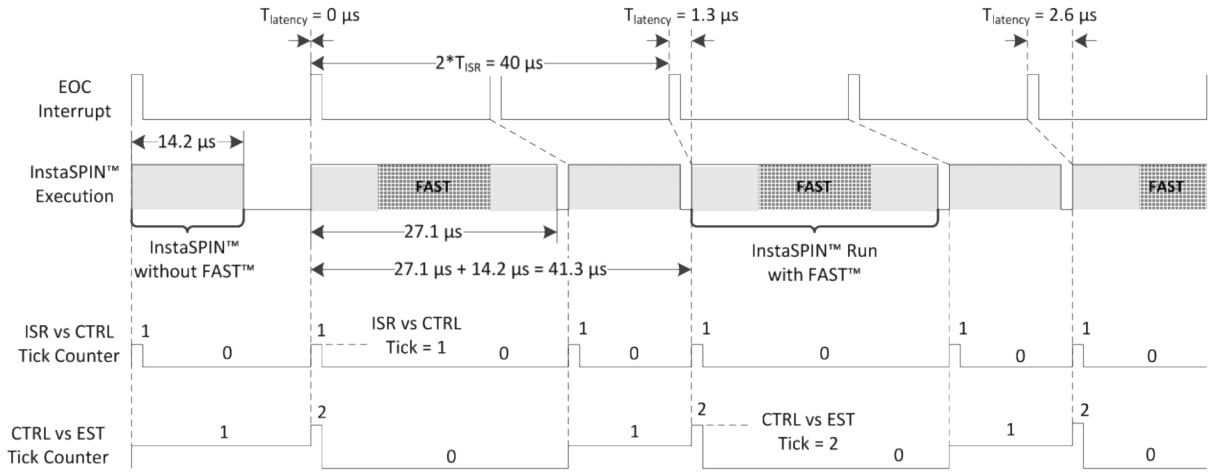


Figure 9-22. CTRL vs. EST Timing - Tick Rate = 2

In this case two interrupts have to be considered to measure timing since there is a tick rate of two being used for the estimator, and as can be seen from the diagram, the time for 2 interrupts is shorter than what needs to be executed without and with FAST, so this will lead to interrupt overrun, hence undesirable behavior.

$2 * T_{ISR} < (T_{InstaSPIN\ without\ FAST} + T_{InstaSPIN\ with\ FAST}) \rightarrow 40\ \mu s < 41.3\ \mu s \rightarrow$ unexpected InstaSPIN results

Figure 9-23 represents the values of this timing diagram in highlighted boxes.

Copyright © 2013–2019, Texas Instruments Incorporated

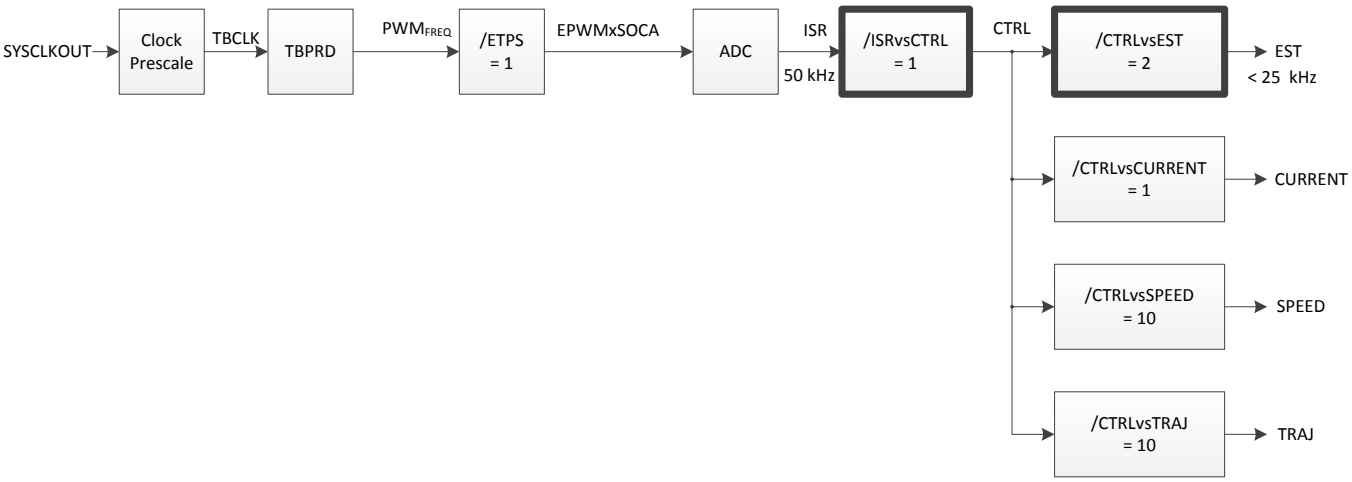


Figure 9-23. CTRL vs. EST Software Execution Clock Tree - Tick Rate = 2

As can be shown, every time InstaSPIN with FAST is executed there is an increasing latency. In three cycles we see how this latency increases from 0 µs, to 1.3 µs and then 2.6 µs. We can predict that in a few more cycles there will be an interrupt overrun, since the execution time is not catching up with the interrupt rate.

The solution for this case study is to increase the CTRL vs. EST tick rate even further, to three, so that the latency is back to zero every InstaSPIN cycle as can be shown in Figure 9-24.

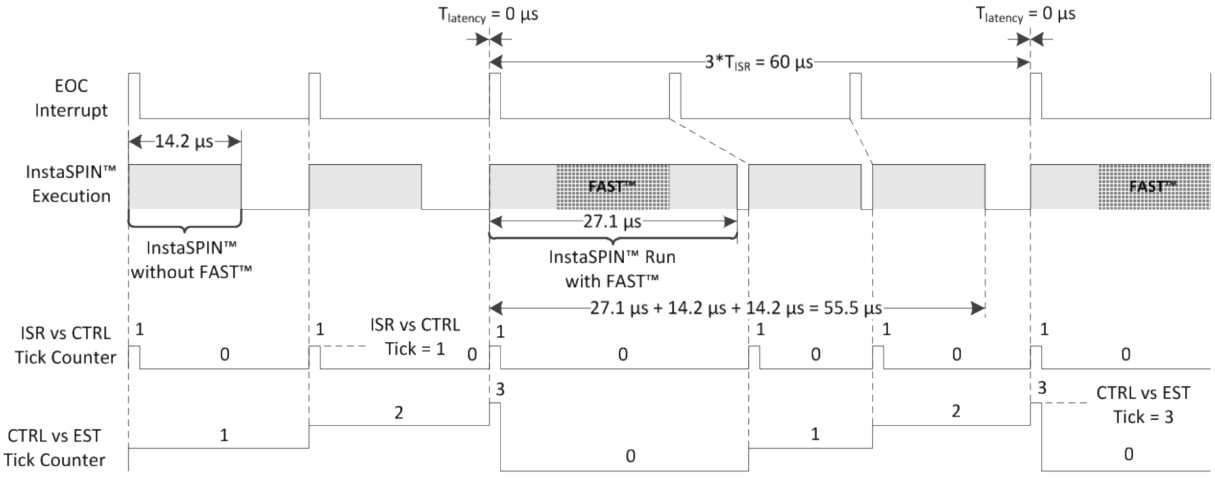


Figure 9-24. CTRL vs. EST Timing - Tick Rate = 3

In this case three interrupts have to be considered to measure timing since there is a tick rate of three being used for the estimator, and as can be seen from the diagram, the time for 3 interrupts is longer than what needs to be executed without and with FAST, so this will avoid interrupt overrun.

$3 * T_{ISR} < (2 * T_{InstaSPIN\ without\ FAST} + T_{InstaSPIN\ with\ FAST}) \rightarrow 60\ \mu s > 55.5\ \mu s \rightarrow$ expected InstaSPIN results

The available CPU for other tasks outside the ISR or other lower priority interrupts can be calculated as follows:

$3 * T_{ISR} - (2 * T_{InstaSPIN\ without\ FAST} + T_{InstaSPIN\ with\ FAST}) = 4.5\ \mu s$

CPU % left = 100 % * 4.5 µs / 60 µs = 7.5 %

2806x MIPS left = CPU % left * Max MIPS = 7.5 % * 90 MIPS = 6.75 MIPS

Figure 9-25 represents the values of this timing diagram in highlighted boxes.

Copyright © 2013–2019, Texas Instruments Incorporated

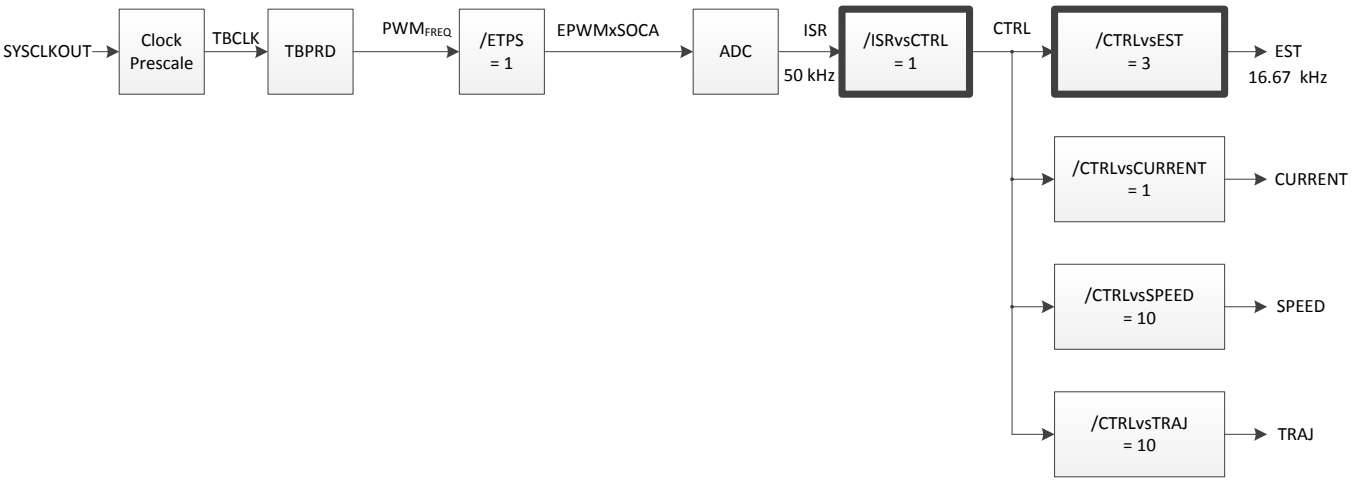


Figure 9-25. CTRL vs. EST Software Execution Clock Tree - Tick Rate = 3

Another solution to this problem is to change the ISR vs. CTRL tick rate to 2. Considering an InstaSPIN run of 2.7 µs when only the tick counters are checked, we have
Changed paragraph in Section 18.2.3, Set Digital IO to connect to QEP Peripheral. As can be seen, selecting a tick rate of 2 for the ISR vs. CTRL tick rate is enough to avoid any conversion overrun.

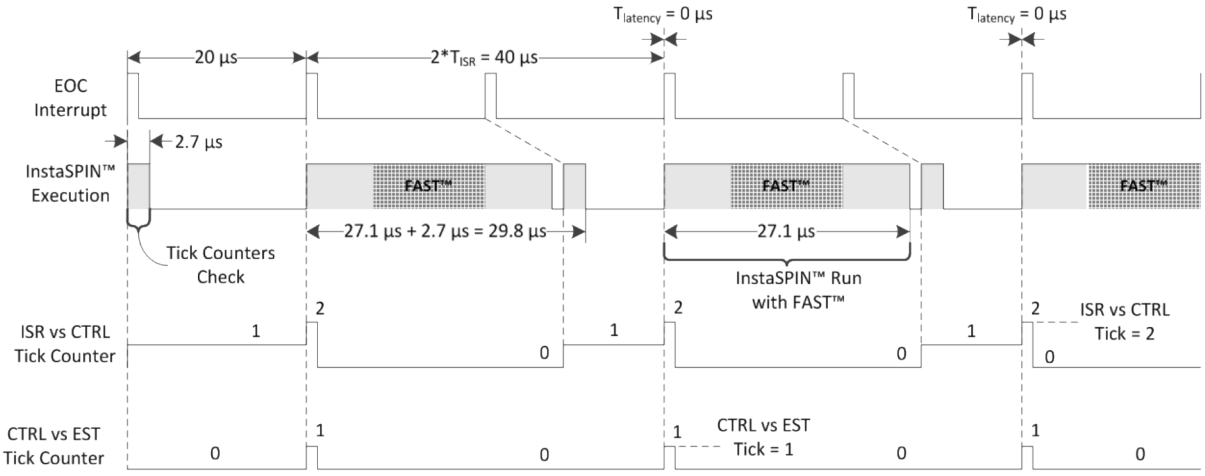


Figure 9-26. ISR vs. CTRL Timing - Tick Rate = 2

In this case two interrupts have to be considered to measure timing since there is a tick rate of two being used for the controller (CTRL), and as can be seen from the diagram, the time for 2 interrupts is longer than what needs to be executed without and with the controller, so this will avoid interrupt overrun.

$2 * T_{ISR} < (T_{InstaSPIN\ without\ CTRL} + T_{InstaSPIN\ with\ CTRL}) \rightarrow 40\ \mu s > 29.8\ \mu s \rightarrow$ expected InstaSPIN results

The available CPU for other tasks outside the ISR or other lower priority interrupts can be calculated as follows:

$2 * T_{ISR} - (T_{InstaSPIN\ without\ CTRL} + T_{InstaSPIN\ with\ CTRL}) = 10.2\ \mu s$

CPU % left = 100 % * 10.2 µs / 40 µs = 25.5 %

2806x MIPS left = CPU % left * Max MIPS = 25.5 % * 90 MIPS = 22.95 MIPS

Figure 9-27 represents the values of this timing diagram in highlighted boxes.

Copyright © 2013–2019, Texas Instruments Incorporated

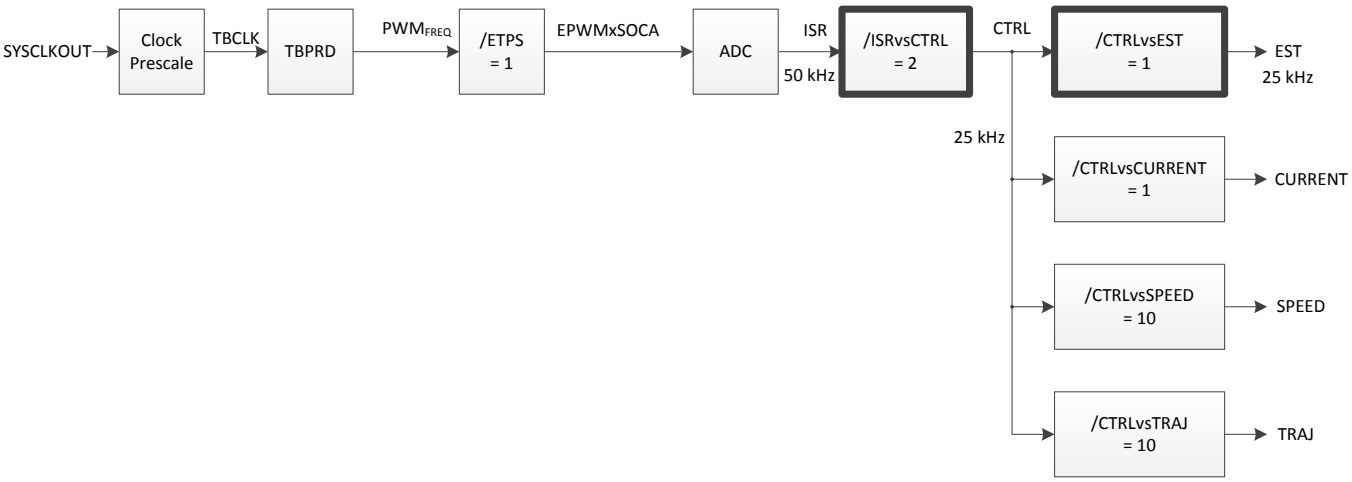


Figure 9-27. ISR vs. CTRL Software Execution Clock Tree - Tick Rate = 2

9.2.5 USER_NUM_CTRL_TICKS_PER_SPEED_TICK

This decimation rate is to execute the speed controller inside of InstaSPIN with respect to the controller (CTRL). A typical value of the speed controller tick rate is between 5 and 10. This is to allow the current controllers to settle at a faster rate compared to a speed controller. The time constant of the speed controller is set by the mechanical load coupled to the motor's shaft, which is much slower than the time constant set by the inductances in the motor. The following example shows a typical value of 10 in the speed controller tick rate, and the timing diagram shows how this is decimated from the controller (CTRL).

A typical value of 10 is used, so that the current controllers are executed at a rate 10 times faster than the speed controller. This is typical since the speed controller usually sets the reference of the current controller, and current controller needs to have some time to control to a specific set point.

Figure 9-28 shows how a speed controller tick rate of 10 is used.

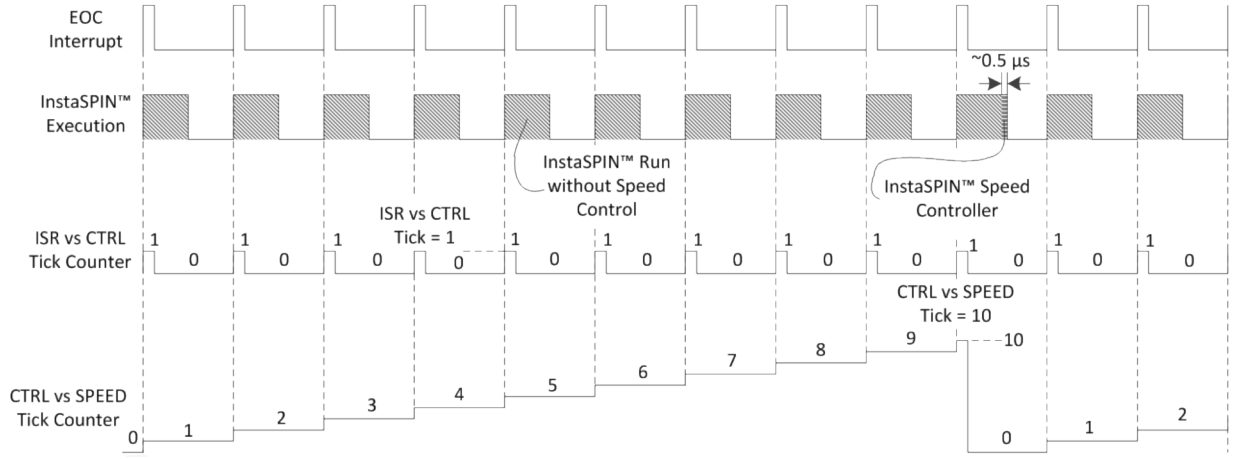


Figure 9-28. Speed Controller Timing - Tick Rate = 10

Figure 9-29 represents the values of this timing diagram in highlighted boxes.

Copyright © 2013–2019, Texas Instruments Incorporated

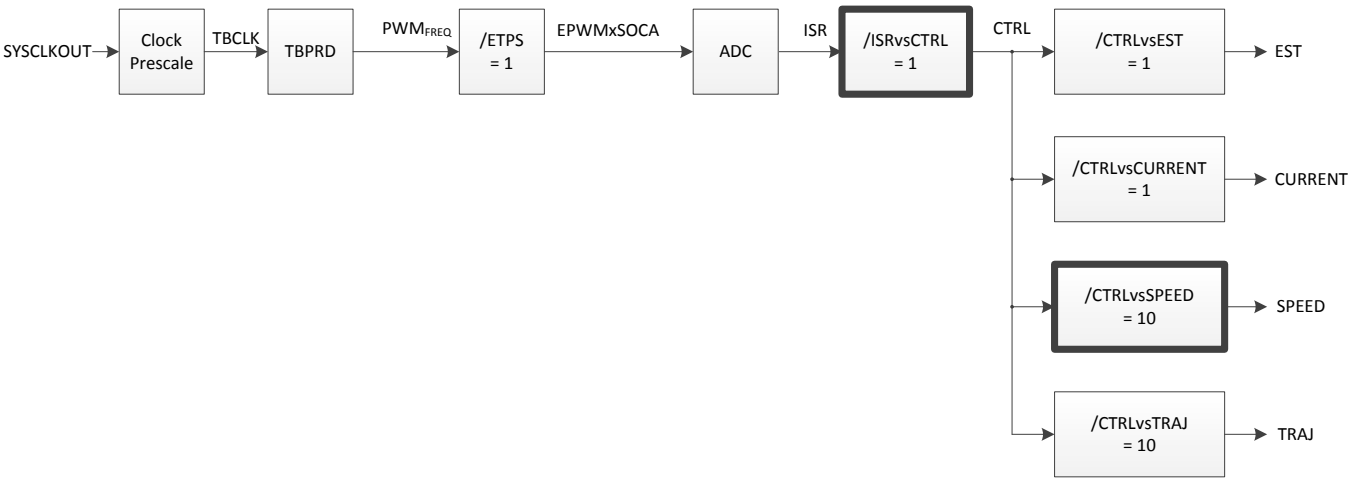


Figure 9-29. Speed Controller Software Execution Clock Tree - Tick Rate = 10

9.2.6 USER_NUM_CTRL_TICKS_PER_TRAJ_TICK

The last decimation rate in the software is related to the trajectory generation within InstaSPIN. The trajectory module is used in the library to provide timing. One example of the trajectories used inside the library is to create a ramp of the speed reference. Another example of the trajectories used is when the motor is being accelerated during the identification process. All of these timings are done by trajectories inside InstaSPIN. All these times are based on the CTRL vs. TRAJ tick rate. Having a different decimation value in this tick rate does not help very much with CPU loading, so it is recommended to match the Speed Controller rate (default of 10) for this tick rate. For illustration purposes, Figure 9-30 shows the CTRL vs TRAJ tick rate.

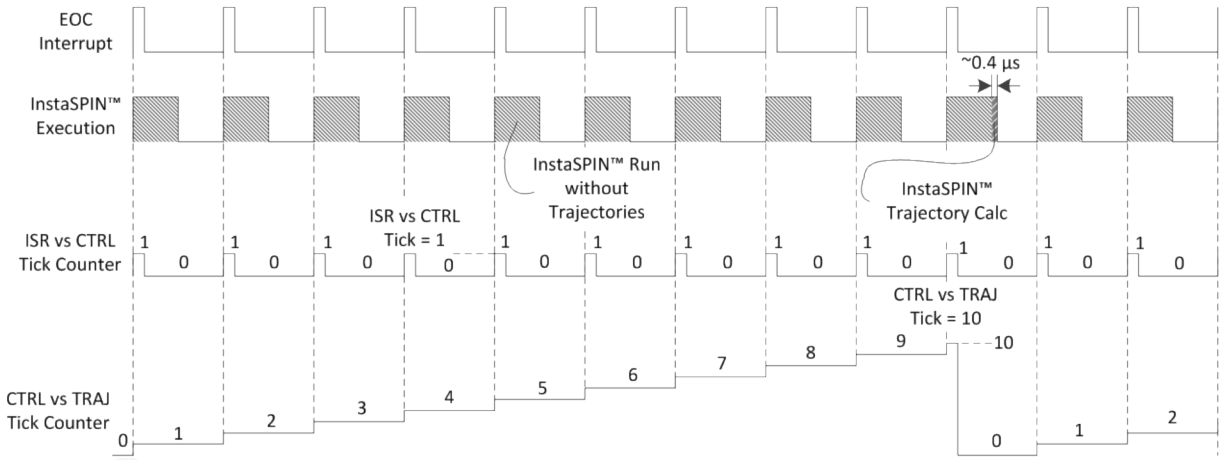


Figure 9-30. CTRL vs TRAJ Tick Rate Timing

Figure 9-31 represents the values of this timing diagram in highlighted boxes.

Copyright © 2013–2019, Texas Instruments Incorporated

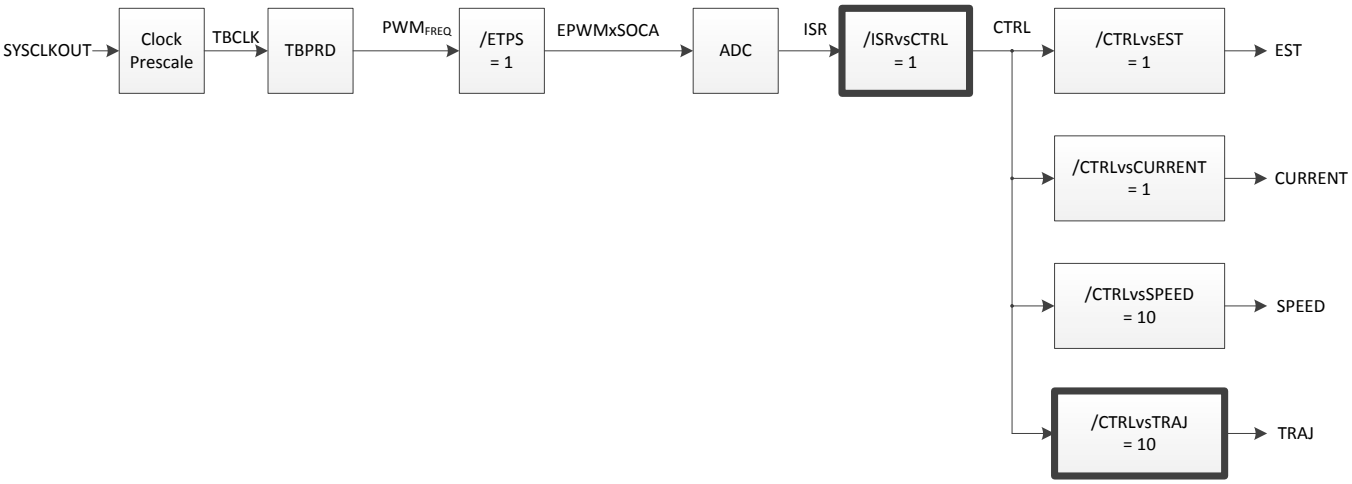


Figure 9-31. CTRL vs TRAJ Tick Rate Software Execution Clock Tree

In summary, all the tick rates, and their dependencies are shown in Figure 9-32 with the following times referenced in the diagram.

SYSCLKOUT = 90 MHz

FOC (InstaSPIN without FAST) = 14.2 µs

FAST = 12.9 µs

Current Control = 1.0 µs

Speed Control = 0.5 µs

Trajectory Run = 0.4 µs

Copyright © 2013–2019, Texas Instruments Incorporated

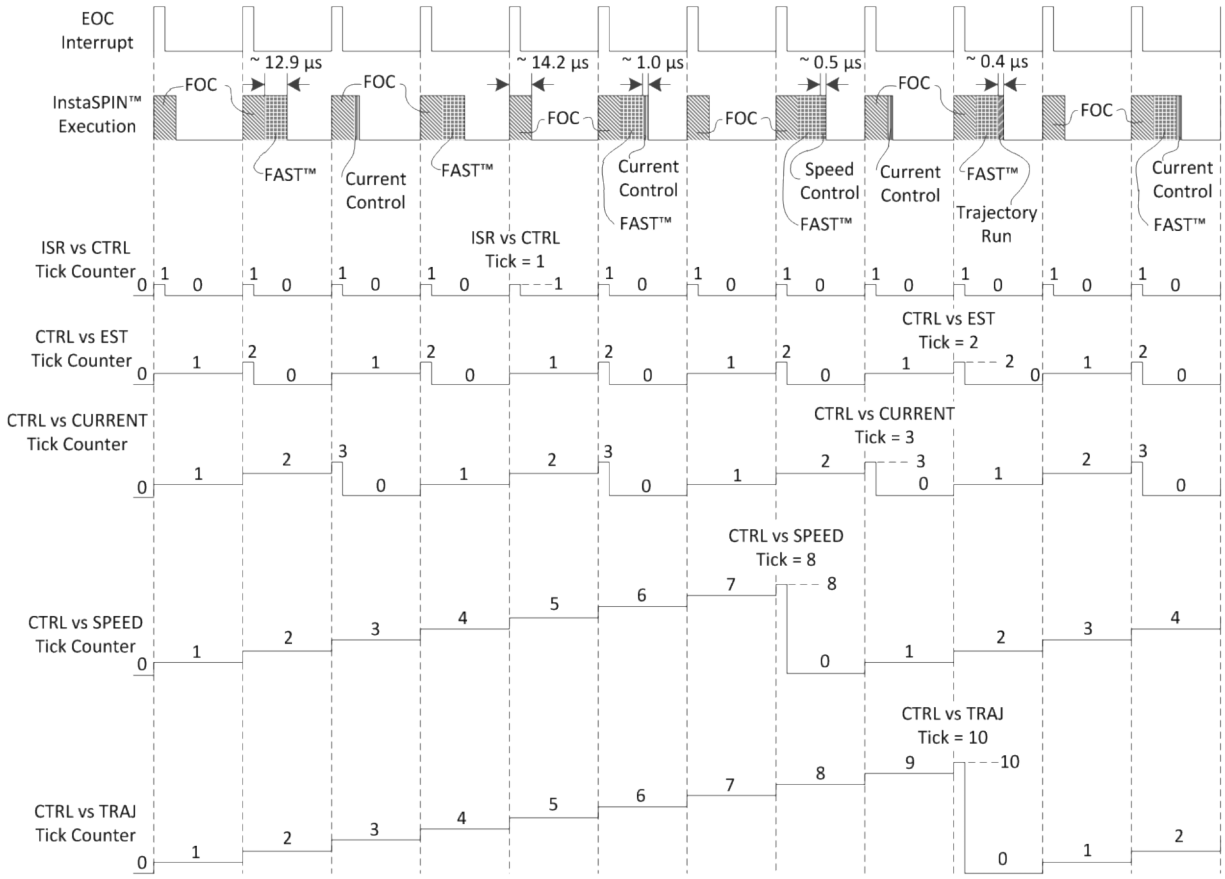


Figure 9-32. All Tick Rates and Dependencies Timing

Figure 9-33 represents the values of this timing diagram in highlighted boxes.

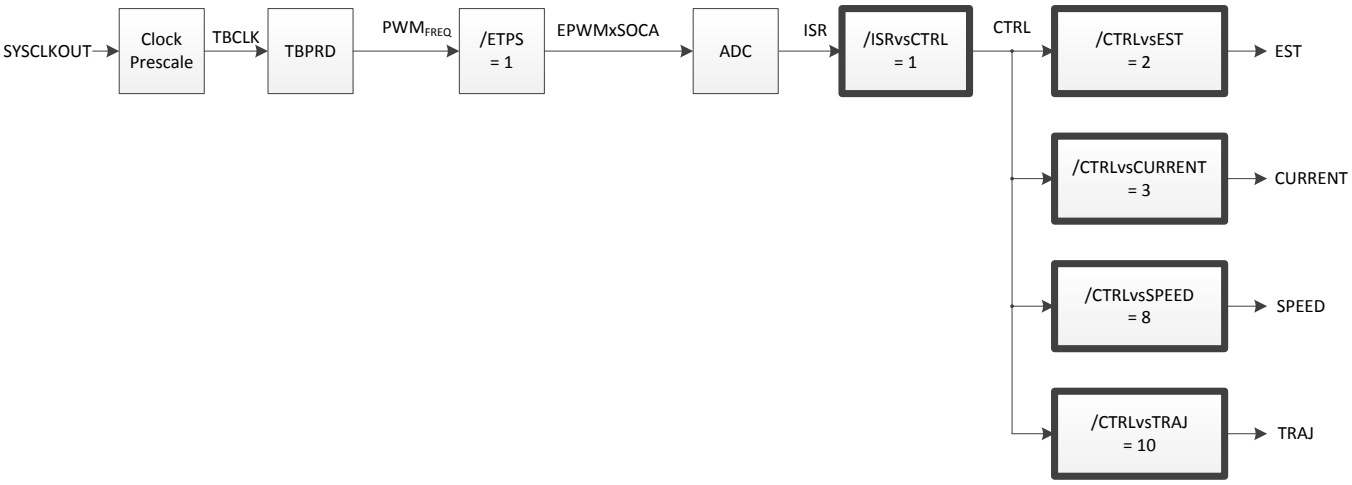


Figure 9-33. All Tick Rates and Dependencies Software Execution Clock Tree

9.3 Decimating in Hardware

The highlighted tick rate shown in Figure 9-34 is used to decimate the execution of InstaSPIN in hardware.

Copyright © 2013–2019, Texas Instruments Incorporated

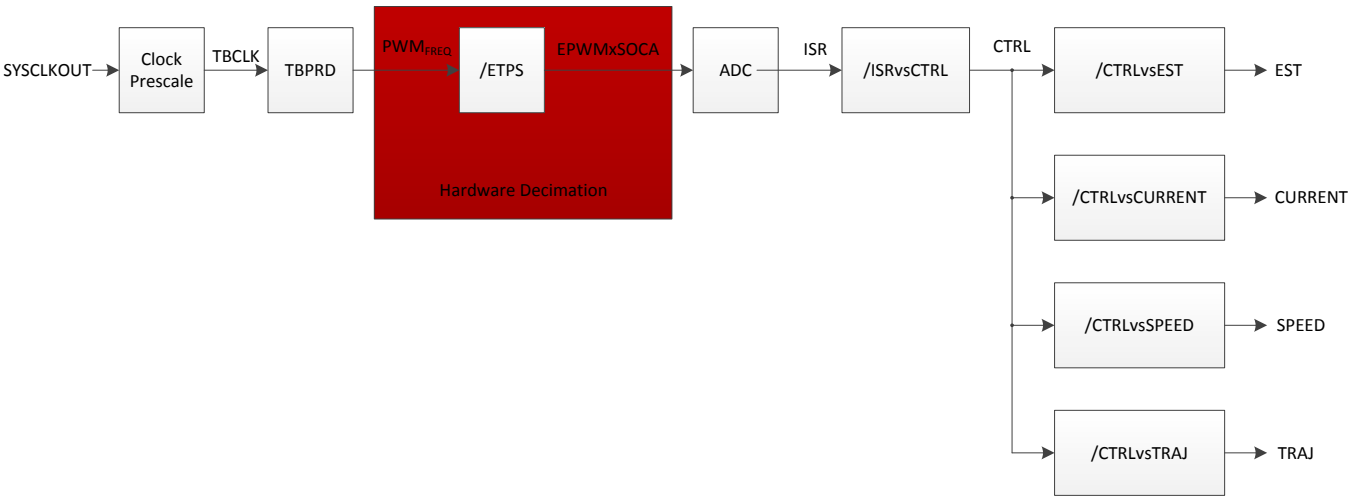


Figure 9-34. Hardware Decimation Software Execution Clock Tree

The only decimation in hardware possible is to trigger the conversions of the ADC at a different rate, other than every PWM cycle. The following configuration in file <user.h>:

```
//! \brief Defines the number of pwm clock ticks per isr clock tick
//!        Note: Valid values are 1, 2 or 3 only
#define USER_NUM_PWM_TICKS_PER_ISR_TICK        (1)
```

With the above example, a start of conversion (SOC) event is triggered every single PWM period, leading to Figure 9-35.

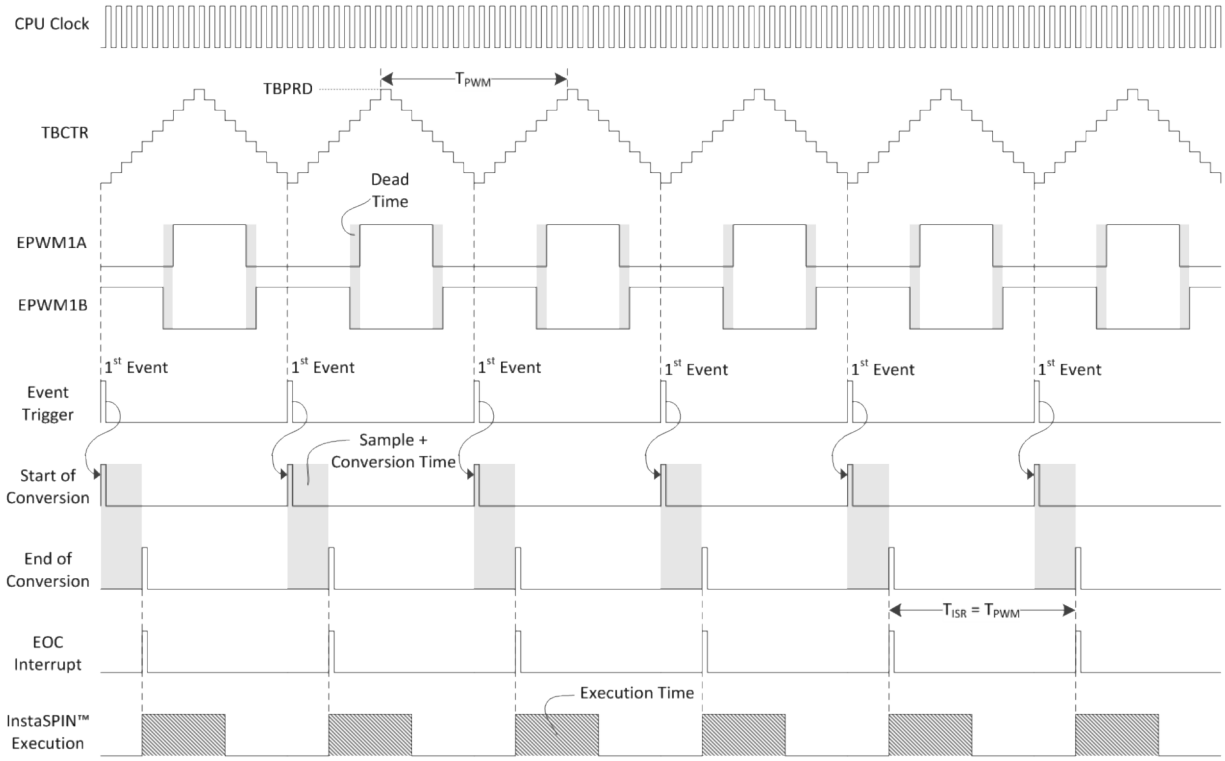


Figure 9-35. SOC Event Timing

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 11-3 represents the values of this timing diagram in highlighted boxes.

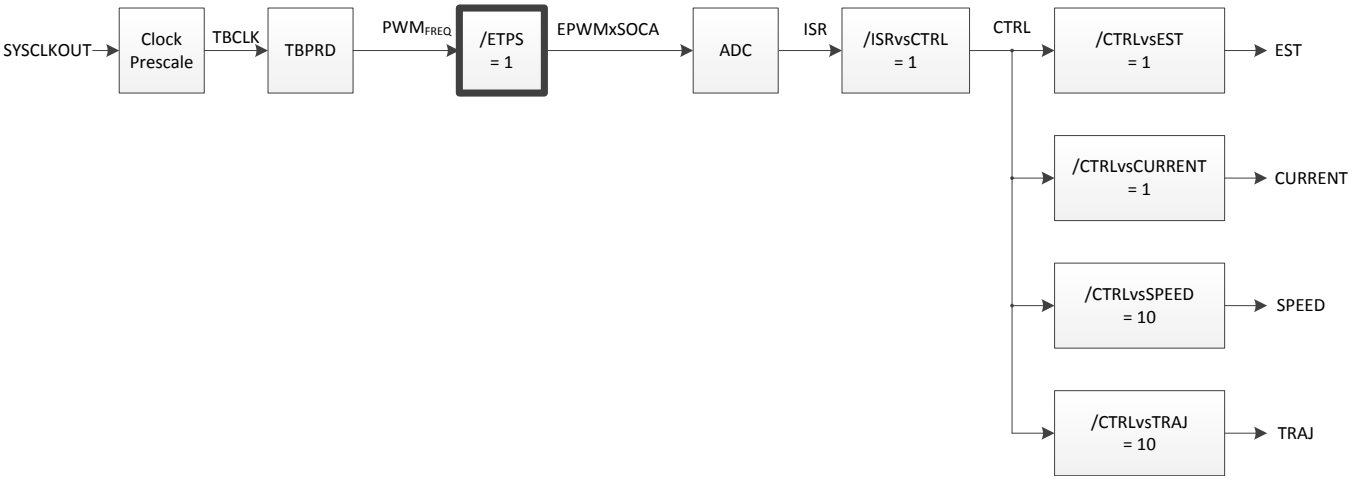


Figure 9-36. SOC Event Software Execution Clock Tree

If a requirement to have a higher PWM frequency in the application, a way of doing this in hardware is by triggering conversions every second or every third PWM cycle. The following example shows how to configure the PWM to trigger conversions on every second PWM cycle:

```
//! \brief Defines the number of pwm clock ticks per isr clock tick
//!        Note: Valid values are 1, 2 or 3 only
#define USER_NUM_PWM_TICKS_PER_ISR_TICK        (2)
```

Figure 9-37 shows the respective timing diagram.

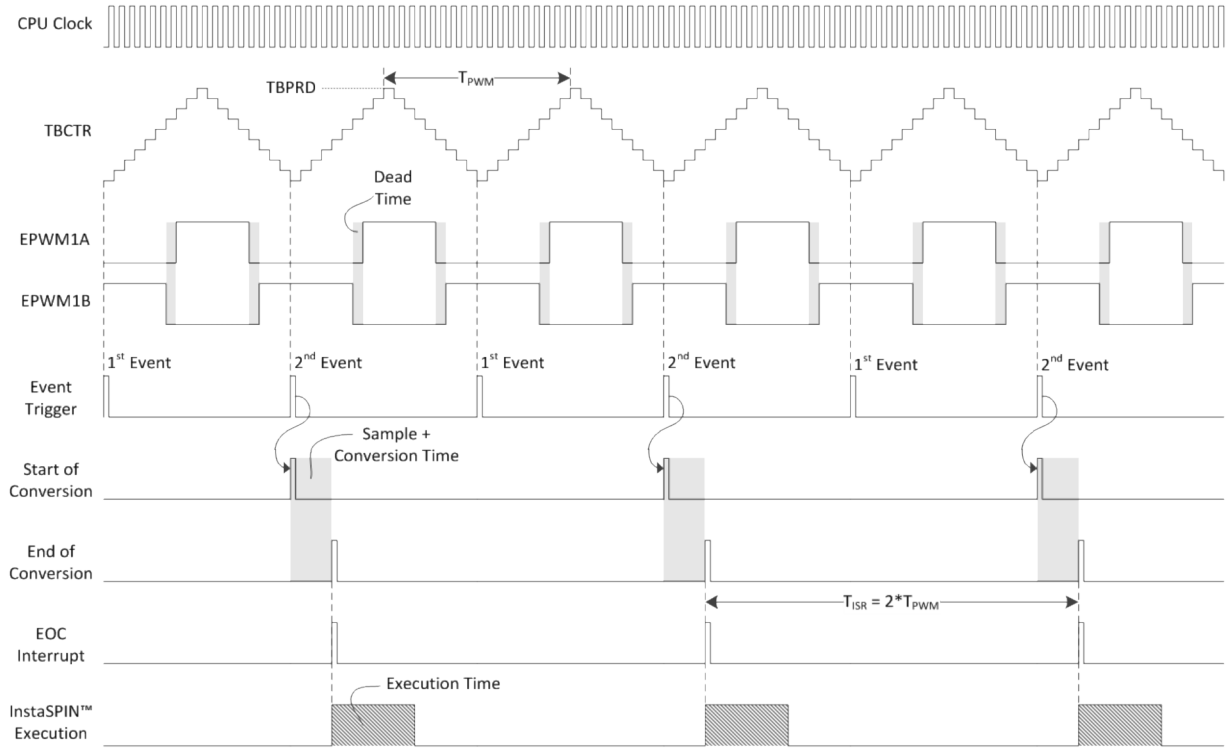


Figure 9-37. PWM Converions on Every Second PWM Cycle Timing

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 9-38 represents the values of this timing diagram in highlighted boxes.

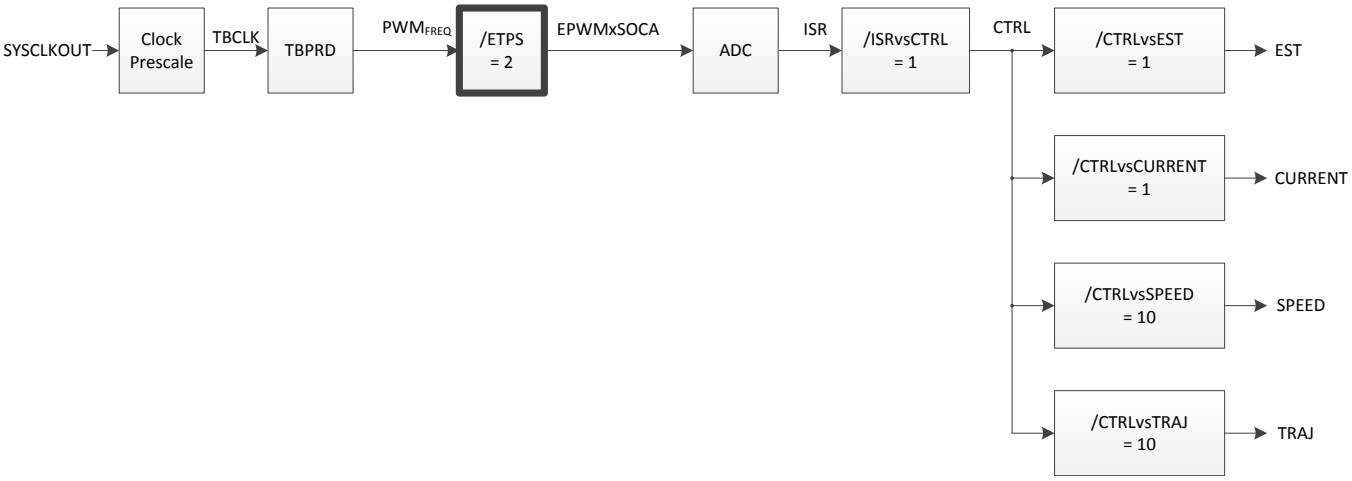


Figure 9-38. PWM Converions on Every Second PWM Cycle Software Execution Clock Tree

If even higher frequency is required, the PWM module can also trigger conversions every third PWM cycles, configured as follows:

```
//! \brief Defines the number of pwm clock ticks per isr clock tick
//!        Note: Valid values are 1, 2 or 3 only
#define USER_NUM_PWM_TICKS_PER_ISR_TICK        (3)
```

Section 18.3.1.2 shows the respective timing diagram.

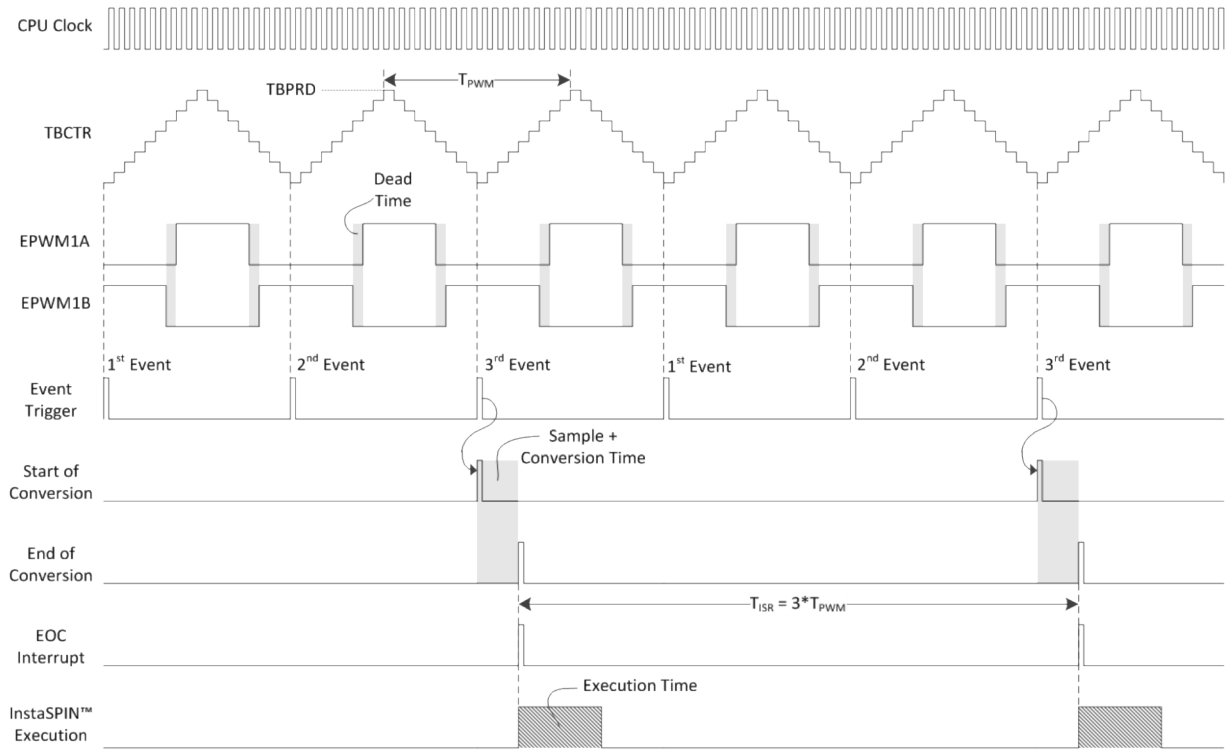


Figure 9-39. PWM Converions on Every Third PWM Cycle Timing

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 9-40 represents the values of this timing diagram in highlighted boxes.

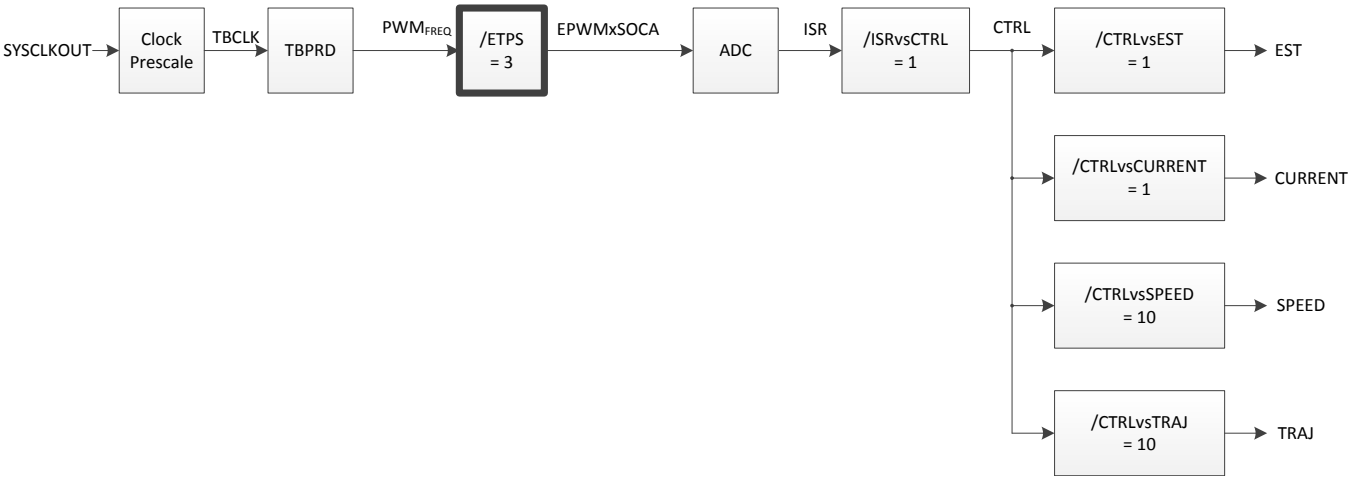


Figure 9-40. PWM Converions on Every Third PWM Cycle Software Execution Clock Tree

Notice how the interrupt period changes with respect to the PWM period. This allows a higher PWM frequency maintaining a higher interrupt period. A higher interrupt period allows InstaSPIN to execute in time, even though the PWM frequency is higher.

Copyright © 2013–2019, Texas Instruments Incorporated

Managing Startup Time

Once the motor has been fully identified, or motor parameters have been loaded from user.h file, there are four possible startup times depending on the enabled recalibration features. These recalibration features are:

- Offset Recalibration
- Stator Resistance (Rs) Recalibration

These two features can be enabled or disabled independently from each other. The main motivation for the user to experiment with different startup methods is to meet the startup requirements of an application. For more details about enabling or disabling these recalibration features, as well as configuring the times and currents for each recalibration feature, see Section 6.7.

Topic **Page**

10.1 Startup with Offsets and Rs Recalibration 383
10.2 Startup with Only Offsets Recalibration..... 384
10.3 Startup with Rs Recalibration..... 385
10.4 Startup with No Recalibration 387
10.5 Bypassing Inertia Estimation 388

Copyright © 2013–2019, Texas Instruments Incorporated

10.1 Startup with Offsets and Rs Recalibration

This is the slowest but most accurate startup. It consists of three stages before the motor is spun to a commanded torque or speed reference. Figure 10-1 shows the controller and estimator state machines when a startup is done with offsets and Rs recalibration enabled.

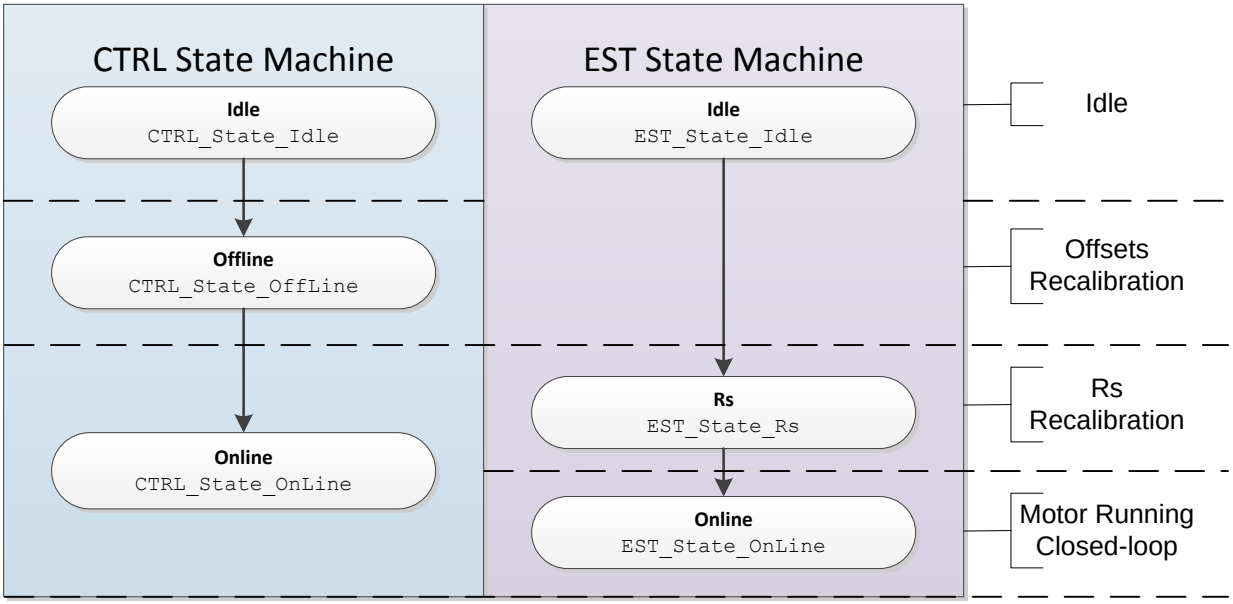


Figure 10-1. Startup with Offsets and Rs Recalibration

Figure 10-2 shows current and output voltage for each state. The first state is the Offsets Recalibration state and the second is Rs Recalibration. The third stage is the online state when the commanded speed or torque is followed in closed-loop.

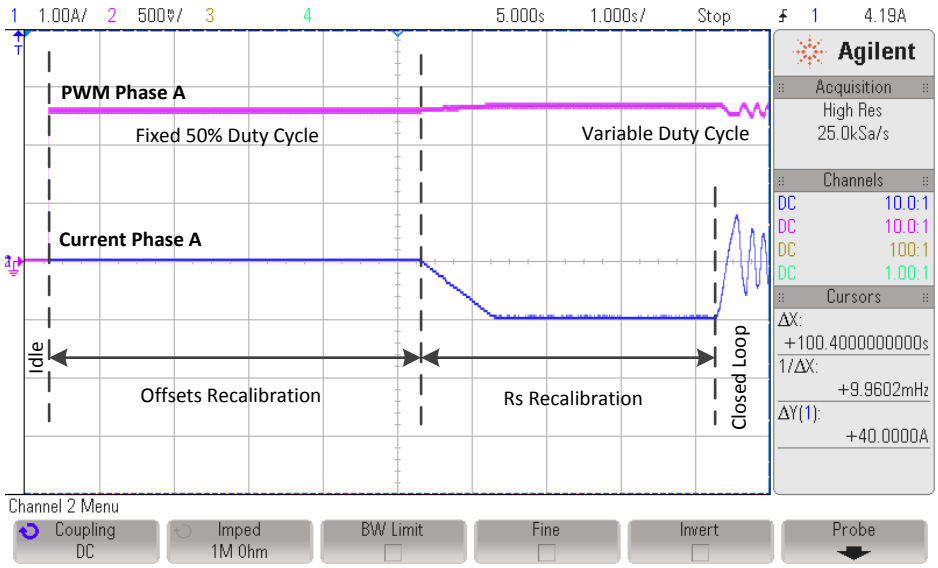


Figure 10-2. Current and Output Voltage for Each State

The timing associated with each state, as well as the current used for Rs recalibration is explained in detail in Chapter 6. In order to enable both offset and Rs recalibration the following two functions must be called prior to enabling the controller:

Copyright © 2013–2019, Texas Instruments Incorporated

```
// Enable Offset Recalibration
CTRL_setFlag_enableOffset(handle, TRUE);

// Enable Rs Recalibration
EST_setFlag_enableRsRecalc(obj->estHandle, TRUE);
```

The controller is enabled by calling the following function:

```
// enable the controller
CTRL_setFlag_enableCtrl(ctrlHandle, TRUE);
```

10.2 Startup with Only Offsets Recalibration

This startup method, with Rs recalibration disabled, is commonly utilized when offsets might have changed, but the motor has not changed. A typical scenario when this approach is used is when different boards run the same motor. Another example is when the same board has been running for a long period of time and the hardware components for the voltage and current feedback might have changed in value due to ambient conditions or component tolerances. In this last example it is recommended to run the offsets recalibration as needed depending on the quality of the hardware components used in a particular board.

Figure 10-3 shows the states when running only offsets recalibration before running in closed loop.

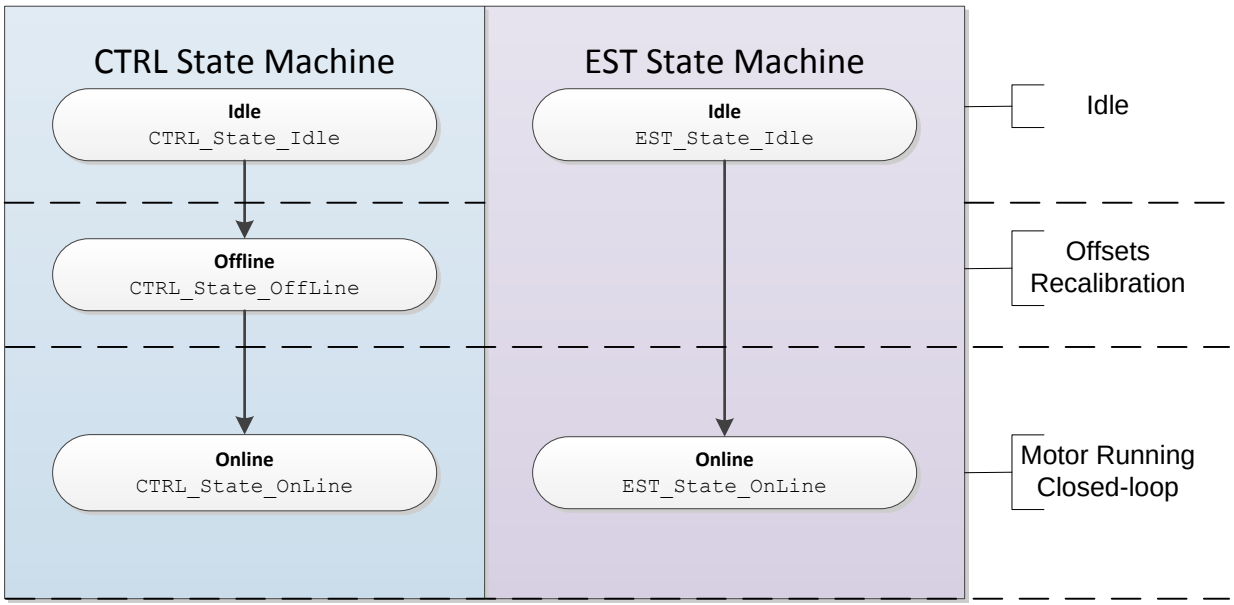


Figure 10-3. Startup with Only Offsets Recalibration

Figure 10-4 shows the current and output voltage waveform associated with the offset state. Before running the motor in closed loop, the offsets are recalibrated with a fixed 50% duty cycle. After that, the motor is then run in closed loop, where the voltage and current would depend on the commanded speed as well as the mechanical load.

Copyright © 2013–2019, Texas Instruments Incorporated

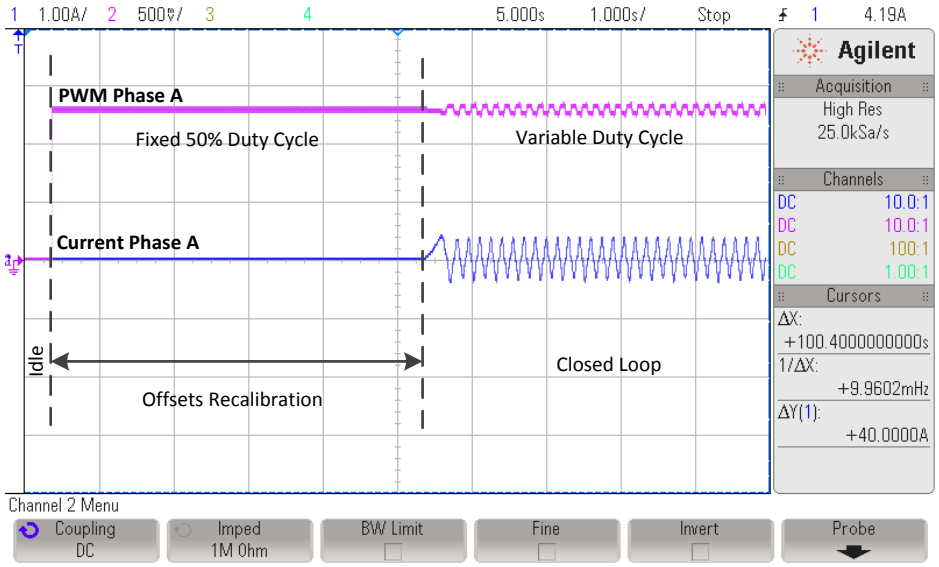


Figure 10-4. Offset State Current and Output Voltage

The timing associated with the offset state is explained in detail in Chapter 6. In order to enable offset recalibration and disable Rs recalibration the following two functions must be called prior to enabling the controller:

```
// Enable Offset Recalibration
CTRL_setFlag_enableOffset(handle, TRUE);

// Disable Rs Recalibration
EST_setFlag_enableRsRecalc(obj->estHandle, FALSE);
```

The controller is enabled by calling the following function:

```
// enable the controller
CTRL_setFlag_enableCtrl(ctrlHandle, TRUE);
```

10.3 Startup with Rs Recalibration

This startup method, with offsets recalibration disabled, is typical when resistance has changed but the offsets have not changed. An example of this condition is if the ambient temperature has changed, causing the stator resistance to change. Also, if the system has been in the field for a long time, it is recommended to run periodic updates to the stator resistance to make sure the software has an accurate representation of the motor model before startup up the motor in closed loop. Figure 10-5 shows the states before closing the loop when only the stator resistance (Rs) is recalibrated

Copyright © 2013–2019, Texas Instruments Incorporated

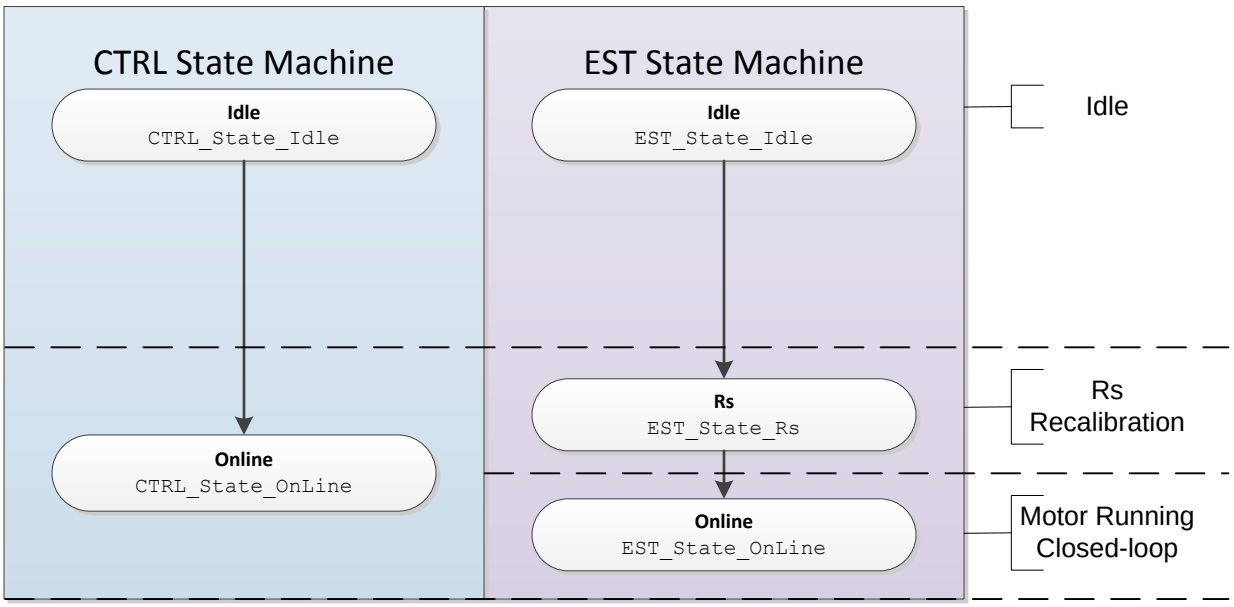


Figure 10-5. Startup with Rs Recalibration

Figure 10-6 shows the current and output voltage waveforms when only Rs is recalibrated before running in closed loop.

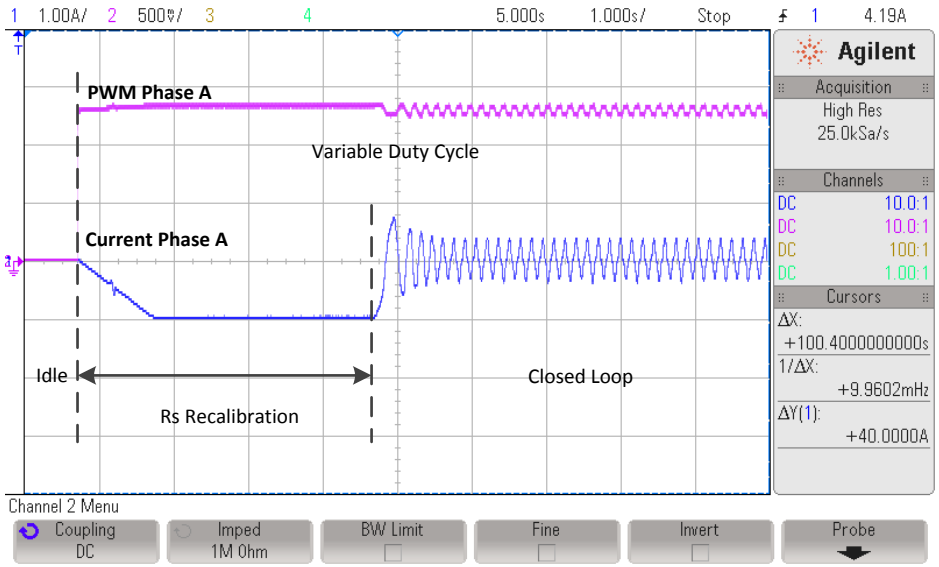


Figure 10-6. Rs Recalibration Current and Output Voltage

The timing associated with the Rs recalibration state as well as the current used to recalibrate Rs is explained in detail in Chapter 6. In order to disable offset recalibration and enable Rs recalibration the following two functions must be called prior to enabling the controller:

```
// Disable Offset Recalibration
CTRL_setFlag_enableOffset(handle, FALSE);

// Enable Rs Recalibration
EST_setFlag_enableRsRecalc(obj->estHandle, TRUE);
```


Copyright © 2013–2019, Texas Instruments Incorporated

The controller is enabled by calling the following function:

```
// enable the controller
CTRL_setFlag_enableCtrl(ctrlHandle, TRUE);
```

10.4 Startup with No Recalibration

This startup approach is the fastest method to get the motor running in closed loop. It does not recalibrate offsets or resistance. As soon as the controller is enabled, the motor is run in closed loop. This method should only be used when the offsets and stator resistance are well known. For details of how to handle full-load conditions at start-up, see Chapter 14. Figure 10-7 shows how the motor is run in closed loop right after the idle state, without any recalibration.

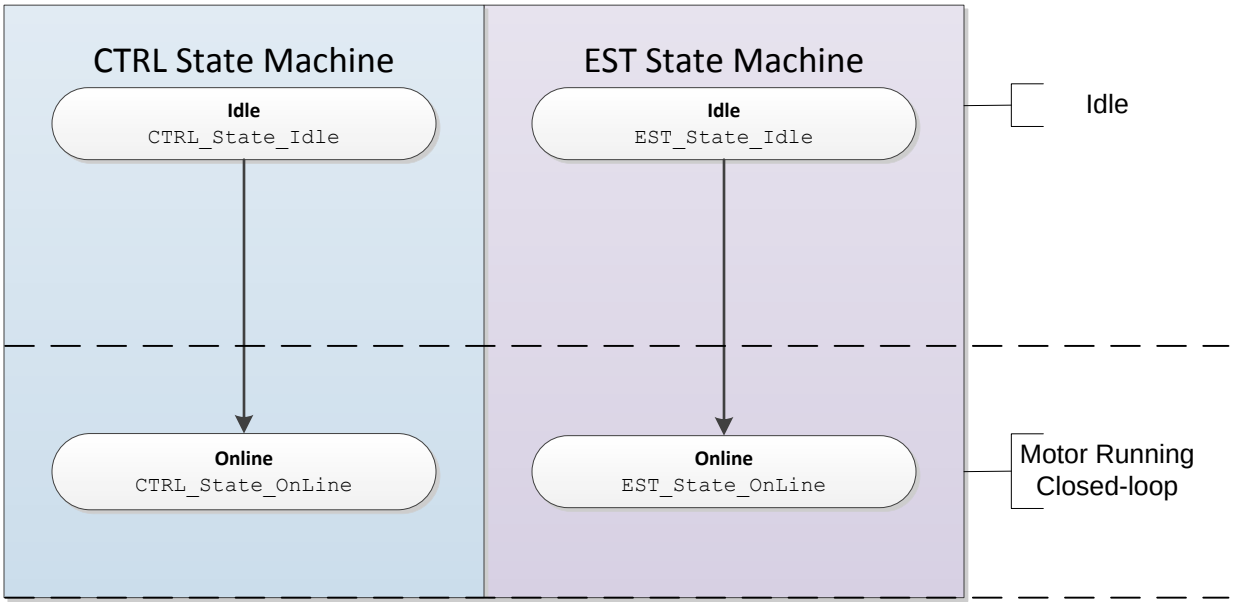


Figure 10-7. Startup with No Recalibration

Figure 10-8 shows the current and output voltage waveforms when offsets and Rs recalibration is bypassed.

Copyright © 2013–2019, Texas Instruments Incorporated

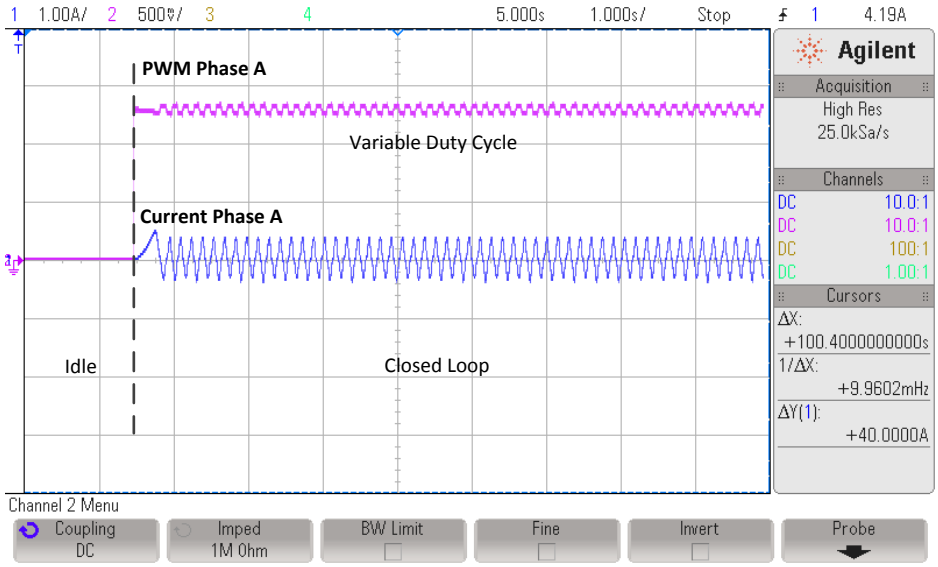


Figure 10-8. Rs Recalibration Bypass Current and Output Voltage

In order to disable both offset and Rs recalibration the following two functions must be called prior to enabling the controller:

```
// Disable Offset Recalibration
CTRL_setFlag_enableOffset(handle, FALSE);

// Disable Rs Recalibration
 EST_setFlag_enableRsRecalc(obj->estHandle, FALSE);
```

The controller is enabled by calling the following function:

```
// enable the controller
CTRL_setFlag_enableCtrl(ctrlHandle, TRUE);
```

10.5 Bypassing Inertia Estimation

If the motor inertia has been previously estiamted, or the motor inertia is known, you can accelerate the system start-up time by bypassing the inertia estimation process. The inertia estimation process should be done during development with a representative inertia attached to the motor shaft. Since the motor inertia is configured during development, SpinTAC Velocity Identify does not need to be included in the final product.

The motor inertia is required for the SpinTAC speed controller. In MotorWare labs, motor inertia is configured as a default value in ST_MOTOR_INERTIA_A_PER_KRPM, located in spintac.h. This definition is covered in greater detail in Section 4.7.1.1. More information on SpinTAC Velocity Identify can be found in Chapter 7.

If your project does not use MotorWare, the motor inertia is set in the SpinTAC speed controller during the initialization process using the Inertia parameter in the SpinTAC speed controller global structure. The unit for Inertia is PU/(pu/s^2). Where PU is the user unit for current [A] and pu/s^2 is the user unit for acceleration [krpm/s]. Typically inertia is specified in Kg*m^2 or N*m*s^2. The user must convert the real world inertia unit into the scaled unit that is used by the SpinTAC speed controller.

Copyright © 2013–2019, Texas Instruments Incorporated

Equation 26 can be used to convert between inertia specified in Kg*m^2 and the scaled units that are needed for the SpinTAC speed controller. The result of this equation should be provided as the Inertia input to the SpinTAC speed controller.

$$\text{Inertia}\left[\frac{\text{PU}}{\frac{\text{pu}}{\text{s}^2}}\right] = \frac{\omega_{\text{NORM}} \times 2\pi}{\varphi_{\text{EMF}} \times A_{\text{NORM}} \times \text{PP}} \times \text{Inertia}\left[\text{Kg} \times \text{m}^2\right] \tag{26}$$

In this equation, the following symbols are used:

- ω_{NORM} is defined as the scale between frequency in Hz and frequency in pu. The value is defined as USER_IQ_FULL_SCALE_FREQ_Hz in user.h. For more information, see Section 4.1.1.
- φ_{EMF} is defined as the Back EMF in Webers of the motor. This value is defined as USER_MOTOR_RATED_FLUX in user.h. For more information, see Section 4.6.7.
- A_{NORM} is defined as the scale between current in amps and current in PU. The value is defined as USER_IQ_FULL_SCALE_CURRENT_A in user.h. For more information, see Section 4.1.5.
- PP is defined as the number of pole pairs in the motor. The value is defined as USER_MOTOR_NUM_POLE_PAIRS in user.h. For more information, see Section 4.6.2.

While you can use this equation to calculate the inertia of your system, it is always preferred to use SpinTAC Velocity Identify to estimate the system inertia. This will provide the most accurate value of the system inertia and will take into account objects that might be difficult to calculate the inertia of.

Copyright © 2013–2019, Texas Instruments Incorporated

Tuning Regulators

Topic **Page**

11.1 PI Controllers Introduction 391
11.2 PI Design for Current Controllers 393
11.3 PI Design for Speed Controllers 397
11.4 Calculating PI Gains Based On Stability and Bandwidth 399
11.5 Calculating Speed and Current PI Gains Based on Damping Factor 401
11.6 Considerations When Adding Poles to the Speed Loop 406
11.7 Speed PI Controller Considerations: Current Limits, Clamping and Inertia 408
11.8 Considerations When Designing PI Controllers for FOC Systems 411
11.9 Sampling and Digital Systems Considerations 415

Copyright © 2013–2019, Texas Instruments Incorporated

11.1 PI Controllers Introduction

Looking back at some history, this is how the PI controller was invented in the 1920s. An engineer named Nicolas Minorsky was designing automatic steering systems for the US Navy in the early 1920s by observing how a helmsman steered a ship under different conditions. According to Wikipedia.org, he noticed that the actions of the helmsman could be approximated by a simple amplification of the error signal under calm conditions, but this simple model was inadequate to describe the helmsman's response during a steady disturbance like a stiff gale. This finally led to the addition of an integral term to correct for continuous steady-state errors. Later, the derivative term was added to improve controllability even further.

Continuing with the Wikipedia.org narrative, test trials of his automatic steering system based on a PI controller were carried out on the USS New Mexico. After some adjustments, he was able to control the angular error to less than two degrees. When the D term was added, the error improved to within one sixth of a degree, which was better than what most helmsmen could achieve manually. Minorsky published his findings (also in the early 1920s). We know today that his discovery launched a new era in the design of control systems.

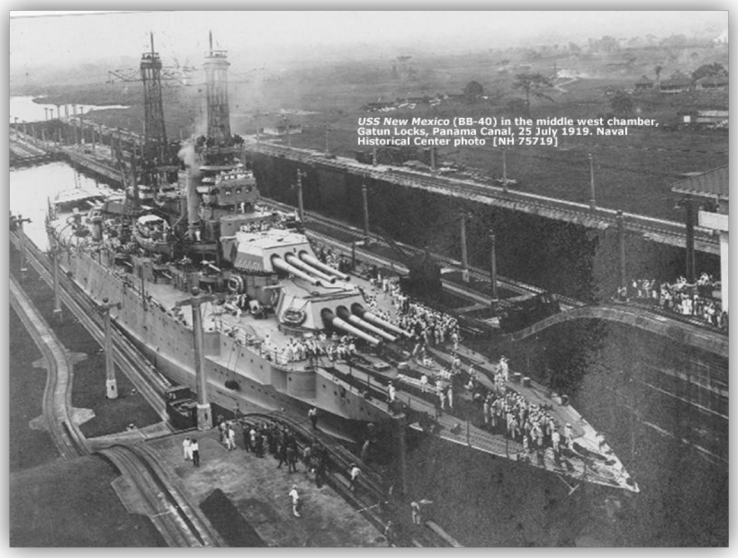


Figure 11-1. USS New Mexico Around the Time it was Retrofitted with PID Control

A very common question we get in our seminars is, "How do you tune a PI controller?" We typically show Bode plots or show some simulation data to show that the process is somewhat empirical, and very subjective to the kind of response for which you are looking.

This section of InstaSPIN User's Guide is put together to help customers design and tune PI control loops (regardless of whether they are speed loops or current loops) in a much more deterministic way. Granted, there are still plenty of degrees of freedom depending on what kind of response users are looking for, as well as an endless litany of subtle variations on the basic PI structure itself. But by following some basic rules that will be explained later in this document, users should be able to tune a PI loop.

This analysis is limited to loads having only real poles. If the load under consideration has prominent complex poles resulting from excessive torsional resonance between the motor and load, then the controller will have to be more sophisticated than a simple PI structure anyway to cancel the resonance effects. But in most cases, a stiff shaft coupler should tame the torsional resonance to the point where the use of a standard PI control structure is acceptable. Also, it is assumed that the load has no viscous damping. In most designs these assumptions are valid. However, if the tuning process described in this section does not work for a given design, it is likely that complex poles or viscous damping exist somewhere in the load which is affecting the results.

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 11-2 shows a parallel path topology of a PI controller. The error signal is split into two separate paths: one which is directly amplified and the other is amplified and then integrated. The integrator is included to drive the steady-state error of the system to zero, since any non-zero steady-state error will result in a boundless integrator output. These two signal paths are then combined at the output once again via a simple addition operation.

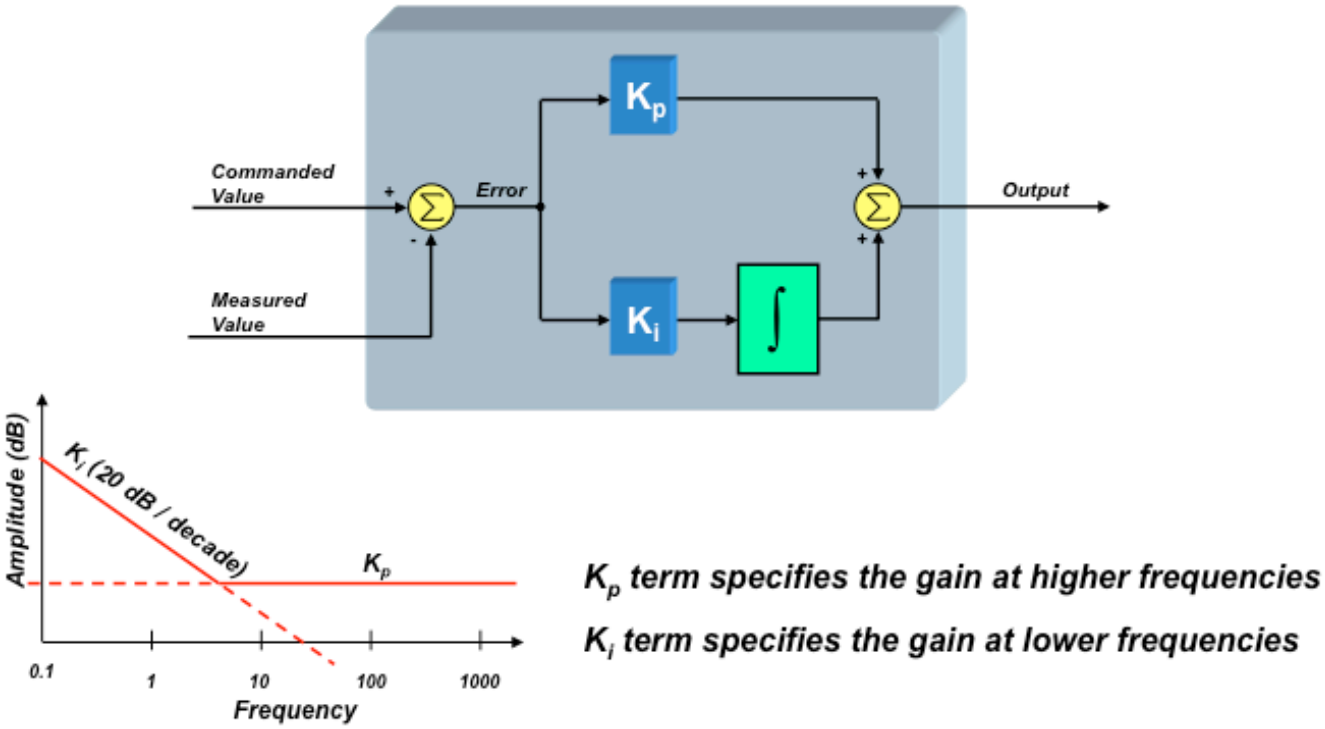


Figure 11-2. Parallel Path Topology

But how do you set the values for Kp and Ki? This has been the subject for much debate, and it is rather difficult to intuitively understand the effect that each term has on your motor control system. The Kp term sets the high frequency gain of the control loop, as shown above. The Ki term sets the low frequency gain, and theoretically has infinite gain at DC. The frequency which delineates the high frequencies from the low frequencies is referred to as the "zero" of the controller and corresponds to the inflection point in the frequency plot.

While the integrator plays a crucial role in the operation of the PI controller, it also brings a set of challenges with it. For example, let's say that the error in your control loop is zero, which means the controlled signal is equal to the commanded signal. Now add a small offset to the controlled signal and watch what happens. Since the error signal is no longer zero, the integrator output will start growing, and growing in an attempt to null the error signal again.

Now remove the offset and watch what happens. The controlled signal will eventually return to the commanded value once again, but not right away. The integrator output is still very large, which causes the controlled signal to wildly overshoot the commanded value while the integrator output is cleared. During this time, the profile of the "controlled" signal is not controlled at all, and may even result in damage to your system if not constrained. It is like winding up a spring tightly and then suddenly releasing it. That is why this effect in PI controllers is called windup. There are many ways to mitigate the windup effect, but most techniques involve some sort of limiting of the integrator's output. We will discuss this in more detail later in this section.

Another popular form of the PI controller (and the one we will use for our analysis) is the "series" topology which is shown in Figure 11-3.

Copyright © 2013–2019, Texas Instruments Incorporated

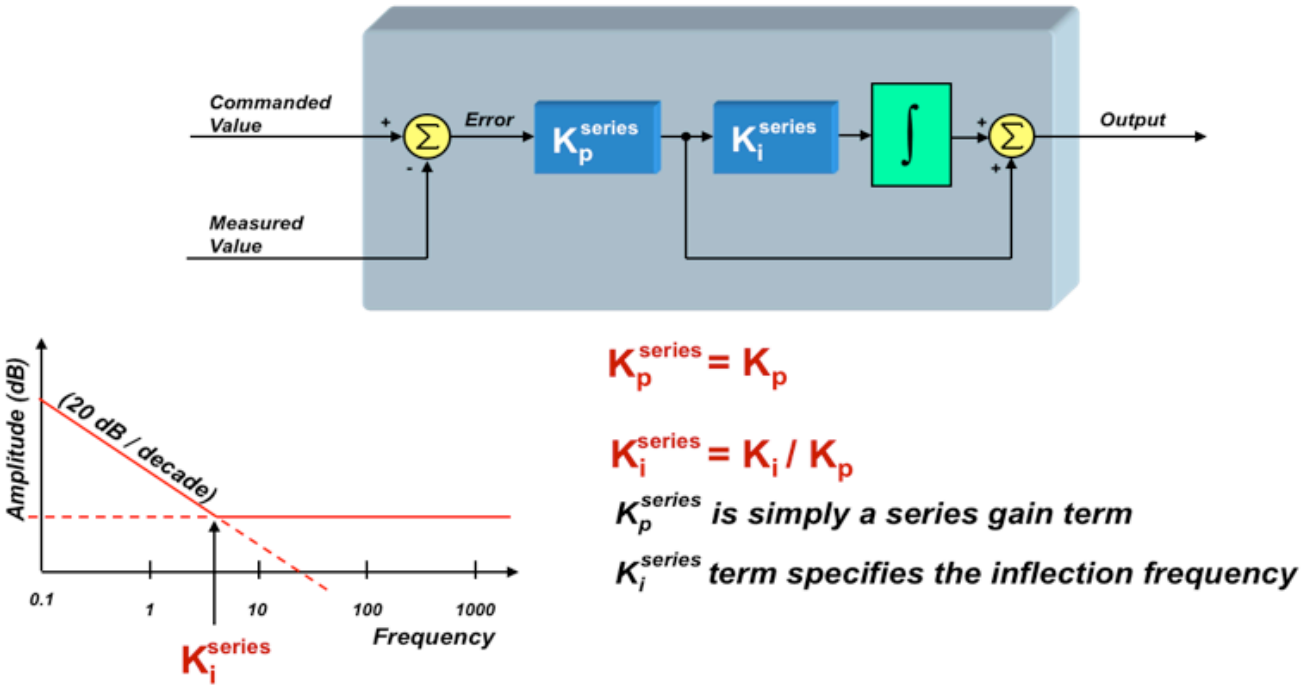


Figure 11-3. Series Topology

From this diagram, we can see that:

$$K_p^{series} = K_p$$
$$K_i^{series} = \frac{K_i}{K_p} \quad (27)$$

But in this structure, K_p^{series} sets the gain for ALL frequencies, and K_i^{series} directly defines the inflection point (zero) of the controller in rad/sec. Both forms are pretty much equal in terms of software complexity. However, many engineers prefer the series form over the parallel form since K_p^{series} and K_i^{series} directly correlate to tangible system parameters. It's pretty easy to understand the effect that K_p^{series} has on the controller's performance, since it is simply a gain term in your open-loop transfer function. But what is the system significance of the zero inflection point? This will be discussed next.

11.2 PI Design for Current Controllers

In the previous section, we briefly reviewed the history of the PI controller and presented two forms that are commonly used today. Regardless of which form you use, the frequency responses look identical, as shown in Figure 11-4. As can be seen from the graph, the gain of the PI controller has a pronounced effect on system stability. But it turns out that the inflection point in the graph (the "zero" frequency) also plays a significant but, perhaps, more subtle role in the performance of the system. To understand this, we will need to dive into some math to derive the transfer function for the PI controller, and understand how the controller's "zero" plays a role in the overall system response.

Copyright © 2013–2019, Texas Instruments Incorporated

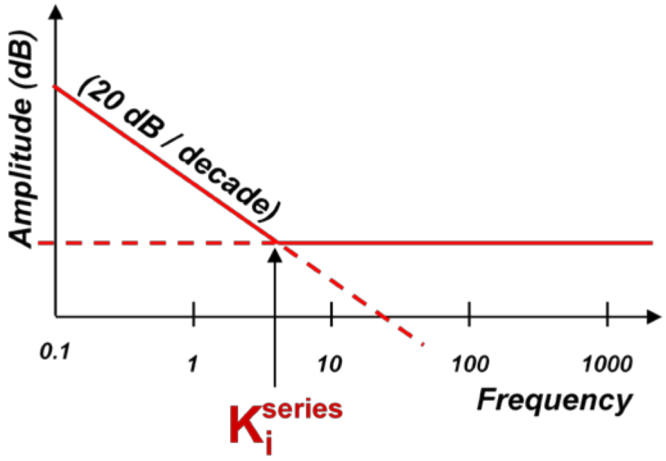


Figure 11-4. Frequency Response

Using the series form of the PI controller, we can define its "s-domain" transfer function from the error signal to the controller output as:

$$PI(s) = \frac{K_p^{series} \times K_i^{series}}{s} + K_p^{series} = \frac{K_p^{series} \times K_i^{series}\left(1 + \frac{s}{K_i^{series}}\right)}{s} \quad (28)$$

From this expression, we can clearly see the pole at s = 0, as well as the zero at $s = K_i^{series}$ rad / sec. So, why is the value of this zero so important? To answer this question, let's drop the PI controller into a current controller which is regulating the current of a motor, as shown in Figure 11-5.

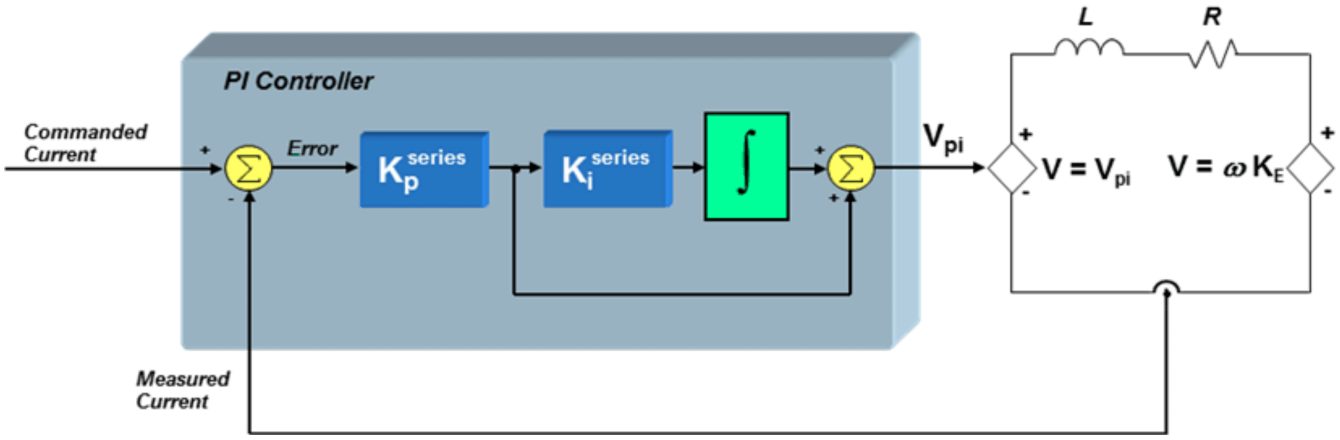


Figure 11-5. PI Controller in a Current Controller

We will use a first-order approximation of the motor winding to be a simple series circuit containing a resistor, an inductor, and a back-EMF voltage source. Assuming that the back-EMF voltage is a constant for now (since it usually changes slowly with respect to the current), we can define the small-signal transfer function from motor voltage to motor current as:

$$\frac{I(s)}{V(s)} = \frac{\frac{1}{R}}{\left(1 + \frac{L}{R}s\right)} \quad (29)$$

Copyright © 2013–2019, Texas Instruments Incorporated

If we also assume that the bus voltage and PWM gain scaling are included in the K_p^{series} term, we can now define the "loop gain" as the product of the PI controller transfer function and the V-to-I transfer function of the RL circuit:

$$G_{loop}(s) = PI(s) \times \frac{I(s)}{V(s)} = \left(\frac{K_p^{series} \times K_i^{series} \left(1 + \frac{s}{K_i^{series}} \right)}{s} \right) \left(\frac{\frac{1}{R}}{\left(1 + \frac{L}{R} s \right)} \right) \tag{30}$$

To find the total system response (closed-loop gain), we must use the following expression which you probably remember from your college control systems class:

$$G(s) = \frac{G_{loop}(s)}{1 + G_{loop}(s)} \qquad \text{(Assuming the feedback term H(s) = 1)} \tag{31}$$

Substituting Equation 30 into Equation 31 yields:

$$G(s) = \frac{\left(\frac{K_p^{series} \times K_i^{series} \left(1 + \frac{s}{K_i^{series}} \right)}{s} \right) \left(\frac{\frac{1}{R}}{\left(1 + \frac{L}{R} s \right)} \right)}{1 + \left(\frac{K_p^{series} \times K_i^{series} \left(1 + \frac{s}{K_i^{series}} \right)}{s} \right) \left(\frac{\frac{1}{R}}{\left(1 + \frac{L}{R} s \right)} \right)} \tag{32}$$

Notice that the expression is getting bigger and bigger, however with some algebra, we can reduce this expression to the following:

$$G(s) = \frac{\left(1 + \frac{s}{K_i^{series}} \right)}{\left(\frac{L}{K_p^{series} \times K_i^{series}} \right) s^2 + \left(\frac{R}{K_p^{series} \times K_i^{series}} + \frac{1}{K_i^{series}} \right) s + 1} \tag{33}$$

The denominator is a second order expression in "s" which means there are two poles in the transfer function. If we are not careful with how we select K_p^{series} and K_i^{series}, we can easily end up with complex poles. Depending on how close those complex poles are to the jω axis, our system could have some resonant peaks. So let's assume right away that we want to select K_p^{series} and K_i^{series} in such a way as to avoid complex poles. In other words, we can factor the denominator into an expression as follows, where C and D are real numbers:

$$\left(\frac{L}{K_p^{series} \times K_i^{series}} \right) s^2 + \left(\frac{R}{K_p^{series} \times K_i^{series}} + \frac{1}{K_i^{series}} \right) s + 1 = (1 + Cs)(1 + Ds) \tag{34}$$

If we multiply out the expression on the right side of the equation, and compare the results with the left side of the equation, we see that in order to obtain real poles, the following conditions must be satisfied:

$$\frac{L}{K_p^{series} \times K_i^{series}} = C \times D \tag{35}$$

And:

Copyright © 2013–2019, Texas Instruments Incorporated

$$\frac{R}{K_p^{series} \times K_i^{series}} + \frac{1}{K_i^{series}} = C \times D \tag{36}$$

As a first attempt to solve Equation 35 and Equation 36, let's simply equate the terms on both sides of Equation 36. In other words:

$$\frac{R}{K_p^{series} \times K_i^{series}} = C \text{ and } \frac{1}{K_i^{series}} = D \tag{37}$$

The reason we recommended these substitutions will now become clear. If we replace the denominator of Equation 33 with its factored equivalent expression as shown in Equation 34, and then make the substitutions recommended in Equation 37, we get the following:

$$G(s) = \frac{\left(1 + \frac{s}{K_i^{series}}\right)}{\left(1 + \frac{R}{K_p^{series} \times K_i^{series}} s\right)\left(1 + \frac{s}{K_i^{series}}\right)} \tag{38}$$

Notice that the "D" substitution results in a pole which cancels out the zero in the closed-loop gain expression. By choosing C and D correctly, we not only end up with real poles, but we can create a closed-loop system response that has only one real pole and no zeros. No peaky frequency responses or resonant conditions, just a simple single-pole low-pass response.

Additionally, by substituting the expressions for C and D recommended in Equation 37 back into Equation 35, we get the following equality:

$$K_i^{series} = \frac{R}{L} \quad \checkmark \tag{39}$$

Keep in mind that K_i^{series} is the frequency at which the controller zero occurs. So in order to get the response described in Equation 38, all we have to do is to set K_i^{series} (the controller zero frequency) to be equal to the pole of the plant.

So, now we know how to set K_i^{series}. But how do we set K_p^{series}? Let's rewrite the closed-loop system response G(s), making all of the substitutions we have discussed up to now, and see what we get:

$$G(s) = \frac{1}{\frac{L}{K_p^{series}} s + 1} \Rightarrow K_p^{series} = L \times Bandwidth \quad \checkmark \tag{40}$$

In summary, there are some simple rules you can use to help you design your PI controller for your current loop:

K_i^{series} sets the zero of the PI controller. When controlling a plant parameter with only one real pole in its transfer function (e.g., the current in a motor), K_i^{series} should be set to the value of this pole. Doing so will result in pole/zero cancellation, and create a closed-loop response that also only has a single real pole. In other words, very stable response with no resonant peaking.

K_p^{series} sets the bandwidth of the closed-loop system response. As seen by Equation 40, the higher K_p^{series} is, the higher the current loop bandwidth will be. We will discuss how to select an appropriate bandwidth in a later section. It happens that K_p^{series} is equal to the inductive impedance for whatever bandwidth frequency you select.

In the following section we will discuss how to design a cascaded PI speed loop which contains a PI current controller as the inner loop.

Copyright © 2013–2019, Texas Instruments Incorporated

11.3 PI Design for Speed Controllers

In the last section we explained how to calculate the P and I coefficients (actually the K_p^{series} and K_i^{series} coefficients in a series structure) for a current loop controller for a motor. We saw that K_i^{series} could be used to eliminate the zero in the closed-loop system response, resulting in a system having only one real pole (i.e., well behaved and stable). K_p^{series} sets the bandwidth of the closed-loop system response.

In this section let's back out and take a look at the speed control loop, which contains another PI controller. Can designing the speed loop be just as simple? Do the coefficient values perform the same system functions they did with the current controller?

It turns out that closing the speed loop is a little more complicated than closing the current loop. Also, to properly design the speed loop, we need to know more system parameters than we did for the current loop. This can be seen in Figure 11-6 which shows all of the components that comprise a cascaded speed control loop. By "cascaded," we mean a control system that consists of an outer loop with one or more inner loops. It bears mentioning again that we are only considering the case of a load with a single lumped sum inertia which is tightly coupled to the motor shaft and no viscous damping.

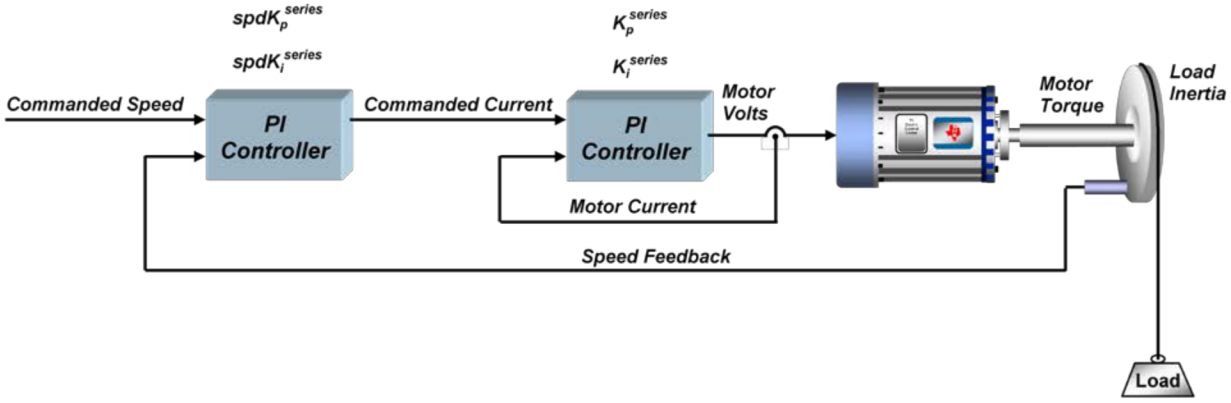


Figure 11-6. Cascaded Speed Control Loop

Let's start with the current control loop since this is where we left off in our last section. Assuming we design the current loop as discussed in my previous section, the closed-loop transfer function is:

$$G_{current}(s) = \frac{1}{\frac{L}{K_p^{series}}s + 1} \tag{41}$$

Where 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. To avoid confusing the coefficients of the speed controller with those of the current controller, we will call the speed controller's coefficients $spdK_p^{series}$ and $spdK_i^{series}$ as shown in Figure 11-6. In the series form of the PI controller, $spdK_p^{series}$ is the error multiplier term ($^{spd}K_p^{series}$), and $spdK_i^{series}$ is the integrator multiplier term ($^{spd}K_i^{series}$). We can use the same equation we did in the last section to define the transfer function of the speed PI controller:

$$PI_{speed}(s) = \frac{spdK_p^{series} \times spdK_i^{series}}{s} + spdK_p^{series} = \frac{spdK_p^{series} \times spdK_i^{series}\left(1 + \frac{s}{spdK_i^{series}}\right)}{s} \tag{42}$$

The transfer function from motor current to motor torque will vary as a function of what type of motor you are using. For a Permanent Magnet Synchronous Motor under Field Oriented Control, the transfer function between q-axis current and motor torque is:

Copyright © 2013–2019, Texas Instruments Incorporated

$$Mtr(s) = \frac{3}{2}\ \frac{P}{2}\lambda_r = \frac{3}{4}P\lambda_r \tag{43}$$

Where:

P = the number of rotor poles

λr = the rotor flux (which is also equal to the back-EMF constant (Ke) in SI units)

For an AC Induction machine, the transfer function between q-axis current and motor torque would be:

$$Mtr(s) = \frac{3}{4}P\frac{Lm^2}{Lr}I_d \tag{44}$$

Where:

P = the number of stator poles

Lm = the magnetizing inductance

Lr = the rotor inductance

I_d = the component of current that is lined up with the rotor flux

For now, let's assume we are using a Permanent Magnet Synchronous Motor.

Finally, the load transfer function from motor torque to load speed is:

$$Load(s) = \frac{1}{J}\frac{1}{s} \tag{45}$$

Where:

J = the inertia of the motor plus the load

Multiplying all these terms together results in the composite open-loop transfer function:

$$GH(s) = \left(\frac{spdK_p^{series} \times spdK_i^{series}\left(1 + \frac{s}{spdK_i^{series}}\right)}{s}\right)\left(\frac{1}{\frac{L}{K_i^{series}}s + 1}\right)\left(\frac{3}{4}P\lambda_r\right)\left(\frac{1}{J}\frac{1}{s}\right) \tag{46}$$

Let's combine all the motor and load parameters at the end of this equation into a single constant K:

$$K = \frac{3P\lambda_r}{4J} \tag{47}$$

Simplifying, we get:

$$GH(s) = \frac{K \times spsK_p^{series} \times spdK_i^{series}\left(1 + \frac{s}{spdK_i^{series}}\right)}{s^2\left(1 + \frac{L}{K_p^{series}}s\right)} \tag{48}$$

From inspection of Equation 48, we can determine the following characteristics of the speed controller's open-loop transfer function:

- Two poles at s = 0, resulting in an attenuation rate at low frequencies of 40 dB per decade of frequency.
- An additional pole at $s = \frac{K_p^{series}}{L}$ (the current controller's pole)
- A zero at $s = spdK_i^{series}$

Copyright © 2013–2019, Texas Instruments Incorporated

In order for stable operation, the pole at $s = \frac{K_p^{series}}{L}$ must be higher in frequency than the zero at $s = spdK_i^{series}$. Other than that, there is an infinite number of combinations of $spdK_p^{series}$ and $spdK_i^{series}$ which could be used to yield different system responses, depending on whether you want higher bandwidth or better stability. There is a procedure to define a single parameter which is proportional to system stability and inversely proportional to bandwidth, which can be used to set both $spdK_p^{series}$ and $spdK_i^{series}$ automatically to yield the maximum phase margin for the selected bandwidth. We will cover the details of this procedure in the next section.

11.4 Calculating PI Gains Based On Stability and Bandwidth

At the end of last section, we discussed the possibility of using a single parameter that could help tune the speed PI loop in a motor control system. To develop this parameter, let's review the open-loop transfer function for the entire speed loop:

$$GH(s) = \frac{K \times spsK_p^{series} \times spdK_i^{series}\left(1 + \frac{s}{spdK_i^{series}}\right)}{s^2\left(1 + \frac{L}{K_p^{series}}s\right)} \quad (49)$$

Where:

- K is the coefficient that contains several terms related to the motor and load
- $spdK_p^{series}$ and $spdK_i^{series}$ are the PI coefficients for the speed loop
- L is the motor inductance
- K_p^{series} is one of the PI coefficients for the current loop
- s is the Laplace frequency variable

Assuming that the pole at $s = \frac{K_p^{series}}{L}$ occurs at a higher frequency than the zero at $s = spdK_i^{series}$, and that the unity gain frequency occurs somewhere in-between these two frequencies, we should end up with a Bode plot that looks something like Figure 11-7.

Copyright © 2013–2019, Texas Instruments Incorporated

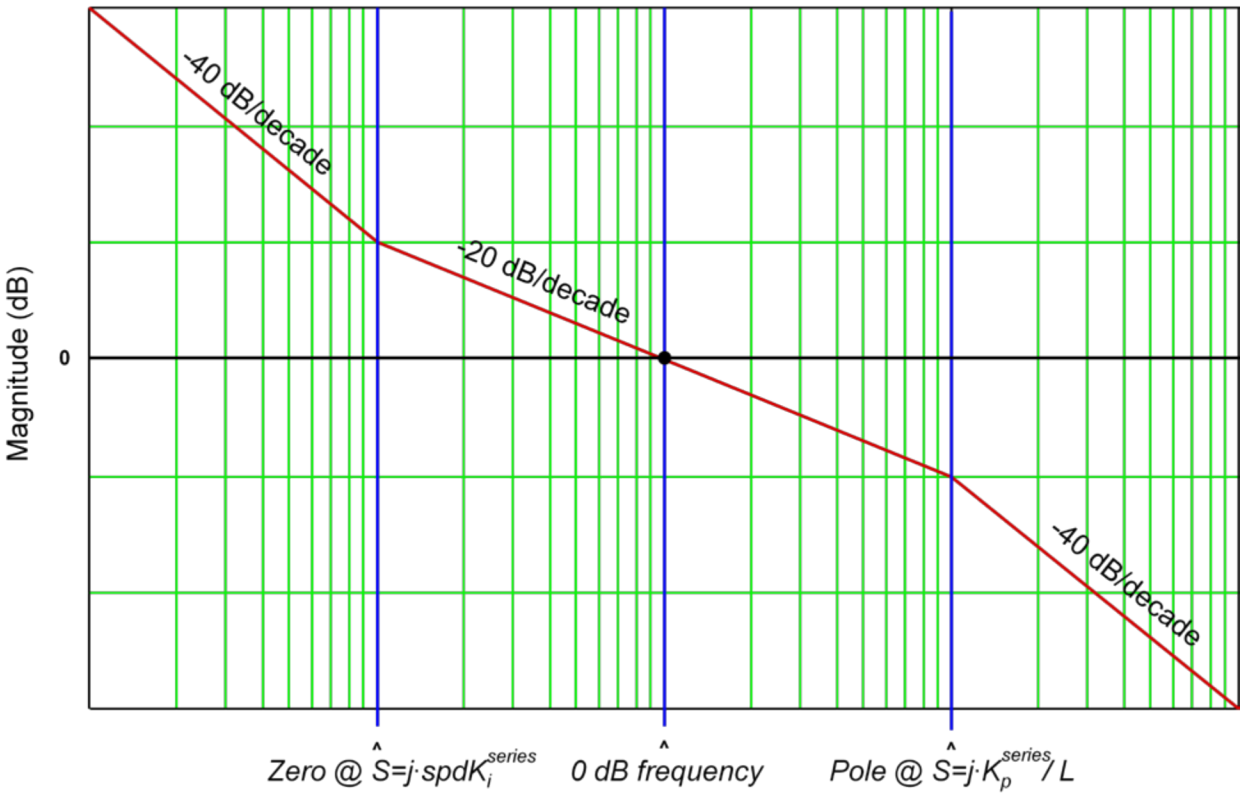


Figure 11-7. Bode Plot

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 order to achieve maximum phase margin (phase shift: 180°) for a given separation of the pole and zero frequencies, the 0 dB frequency should occur exactly half way in-between these frequencies on a logarithmic scale. In other words,

$$\omega_{0\,dB} = \delta \times \omega_{zero} \tag{50}$$

and,

$$\omega_{pole} = \delta \times \omega_{0\ dB} \tag{51}$$

Combining Section 12.3.5.6 and Equation 51 we can establish that:

$$\omega_{pole} = \delta^2 \times \omega_{zero} \tag{52}$$

From Equation 49, we see that ω_{pole} and ω_{zero} are already defined in terms of the PI coefficients. Therefore,

$$spdK_i^{series} = \frac{K_p^{series}}{\delta^2 L} \quad \checkmark \tag{53}$$

Where "δ" we will define as the damping factor. The larger δ is, the further apart the zero corner frequency and the current loop pole will be. And the further apart they are, the phase margin is allowed to peak to a higher value in-between these frequencies. This improves stability at the expense of speed loop bandwidth. If δ = 1, then the zero corner frequency and the current loop pole are equal, which results in pole/zero cancellation and the system will be unstable. Theoretically, any value of δ > 1 is stable since phase margin > 0. However, values of δ close to 1 result in severely underdamped performance.

We will talk more about δ later, but for now, let's turn our attention towards finding the last remaining coefficient: $spdK_p^{series}$. From Section 12.3.5.6 we see that the open-loop transfer function of the speed loop from Equation 49 will be unity gain (0 dB) at a frequency equal to the zero inflection point frequency multiplied by δ. In other words,

Copyright © 2013–2019, Texas Instruments Incorporated

$$\left| \frac{K \times spdK_p^{series} \times spdK_i^{series} \left(1 + \frac{s}{spdK_i^{series}} \right)}{s^2 \left(1 + \frac{s}{\delta^2 \times spdK_i^{series}} \right)} \right|_{s=j\times\delta\times spdK_i^{series}} = 1 \tag{54}$$

By performing the indicated substitution for "s" in Equation 54 and solving, we obtain:

$$\frac{\delta \times K \times spdK_p^{series}}{\delta^2 \left(\frac{K_p^{series}}{\delta^2 \times L} \right)} = 1 \tag{55}$$

Finally, we can solve for $spdK_p^{series}$:

$$spdK_p^{series} = \frac{K_p^{series}}{L \times \delta \times K} = \frac{\delta \times spdK_i^{series}}{K} \quad \checkmark \tag{56}$$

At this point, let's step back and try to see the forest for the trees. We have just designed a cascaded speed controller for a motor which contains two separate PI controllers: one for the inner current loop and one for the outer speed loop. In order to get pole/zero cancellation in the current loop, we chose K_i^{series} as follows:

$$K_i^{series} = \frac{R}{L} \tag{57}$$

K_p^{series} sets the bandwidth of the current controller:

$$\text{Bandwidth} = \frac{K_p^{series}}{L} \tag{58}$$

Once we have defined the parameters for the inner loop current controller, we select a value for the damping factor (δ) which allows you to precisely quantify the tradeoff between speed loop stability and bandwidth. Then it is a simple matter to calculate $spdK_p^{series}$ and $spdK_i^{series}$:

$$spdK_i^{series} = \frac{K_p^{series}}{\delta^2 \times L} \tag{59}$$

$$spdK_p^{series} = \frac{\delta \times spdK_i^{series}}{K} \tag{60}$$

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. In the next section, let's take a closer look at the damping factor, and we will come up with a way to set the current loop bandwidth based on the desired speed loop bandwidth.

11.5 Calculating Speed and Current PI Gains Based on Damping Factor

So far in this series, we have discussed how to distill the design of a cascaded speed controller from four PI coefficients down to two "system" parameters. One of those parameters is simply the bandwidth of the current controller. The other is the damping factor (δ). The damping factor represents the tradeoff between system stability and system bandwidth in a single number. Keep in mind that we are only considering loads which contain only torque and inertial components (i.e., no torsional resonance or viscous damping). Let's move forward by taking a closer look at the damping factor in both the time and frequency domains.

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 11-8 illustrates the open-loop magnitude and phase response for a system where the current controller bandwidth is arbitrarily set to 100 Hz. For our purposes, it really doesn't matter what the current bandwidth is, as it only serves to provide a reference point on the frequency axis. However, the shape of the curves won't change, regardless of what the current bandwidth is. The damping factor is swept from 1.5 to 70 in 8 discrete steps to show how it affects system response. A value of 1.0 corresponds to the condition where the open-loop gain intercepts 0 dB right at the frequency of the current controller bandwidth. This results in pole/zero cancellation at this frequency with a phase margin of zero. It goes without saying that zero phase margin equals bad things for your system.

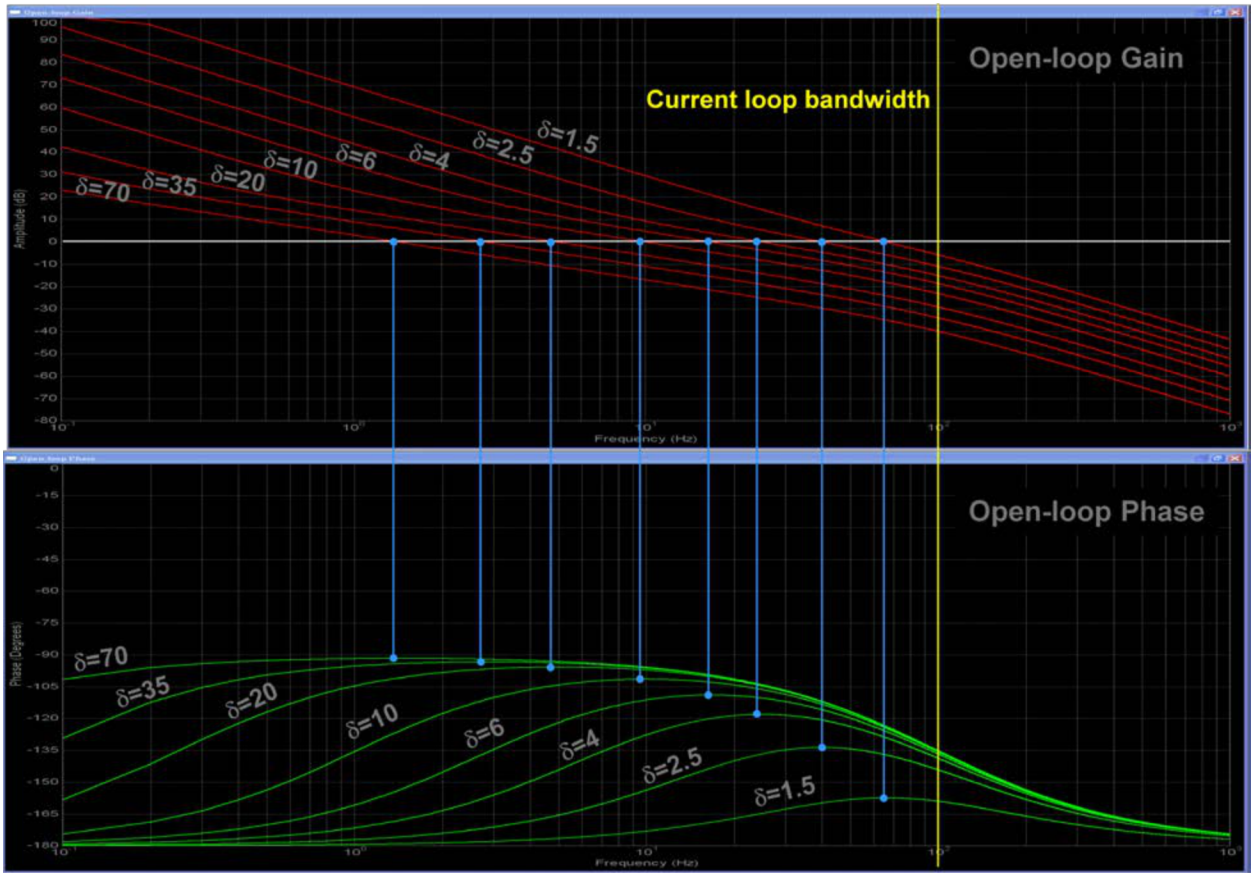


Figure 11-8. Speed Controller Open Loop Magnitude and Phase Response as a Function of δ

One of the goals with the damping factor equations is to achieve the maximum stability possible for a given bandwidth. This is seen on the open-loop phase plots which indicate the phase margin peaks to its maximum value right at the frequency where the open-loop gain plots cross 0 db. As the stability factor is increased, you eventually reach a point of diminishing returns as the signal phase shift approaches -90 degrees. However, the gain margin continues to improve at the expense of a much slower system response.

Figure 11-9 illustrates the closed loop magnitude response of the speed loop, again assuming a current controller bandwidth of 100 Hz. Just like the open-loop response, the actual current controller bandwidth is irrelevant in determining the shape of the curves and only serves to associate the curves with a specific frequency reference point.

Copyright © 2013–2019, Texas Instruments Incorporated

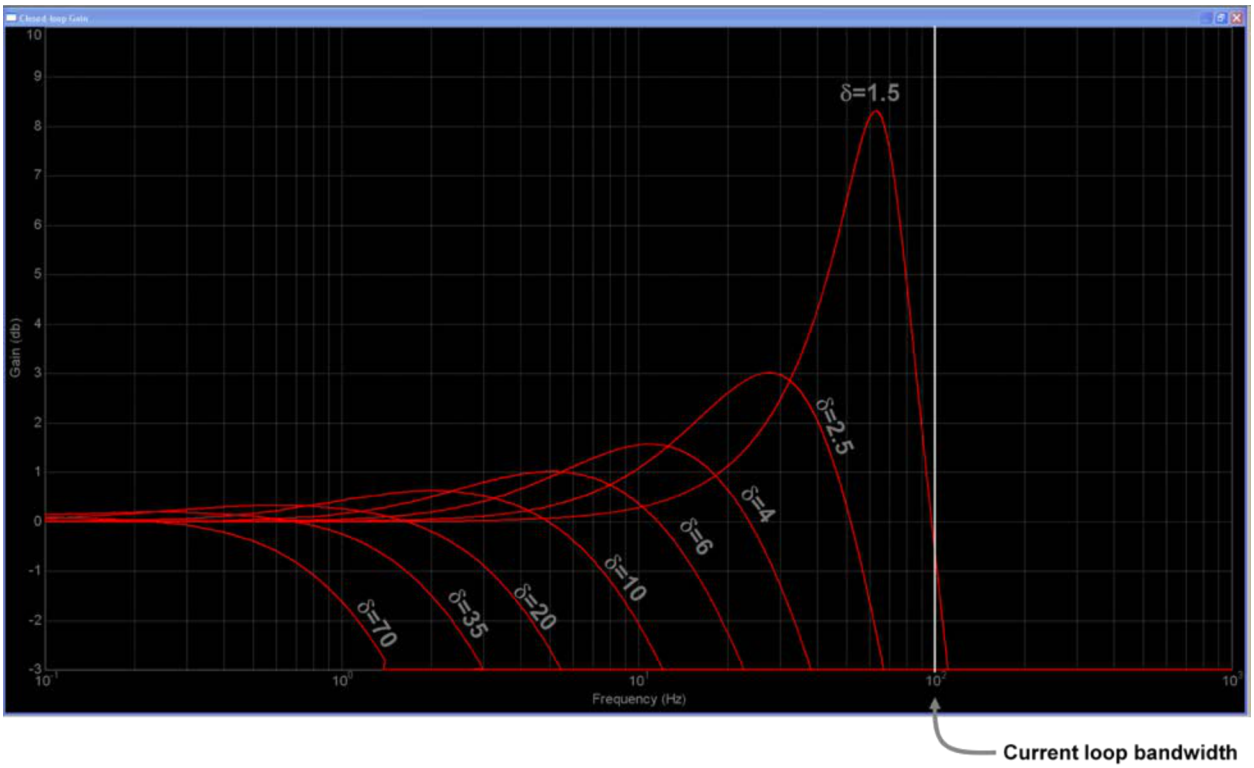


Figure 11-9. Speed Controller Closed Loop Bandwidth as a Function of δ

The required frequency separation between the -3 dB cutoff point of the speed closed loop response and the current controller pole is clearly seen along the bottom of the graph for various values of the damping factor. As the damping factor approaches unity, the complex poles in the speed loop approach the required frequency sepdamped ringing. This is perhaps better visualized in Figure 11-10, which shows the normalized step response of the system for various values of the damping factor. Values below 2 are usually unacceptable due to the large amount of overshoot. At the other end of the scale, values much above 30 usually unacceptable due to the large amount of overshoot. At the other end of the scale, values much above 30 usually don't work either because of the extremely long rise and settling times, as seen in the step response curves. In-between these values is usually your design target window.

Copyright © 2013–2019, Texas Instruments Incorporated

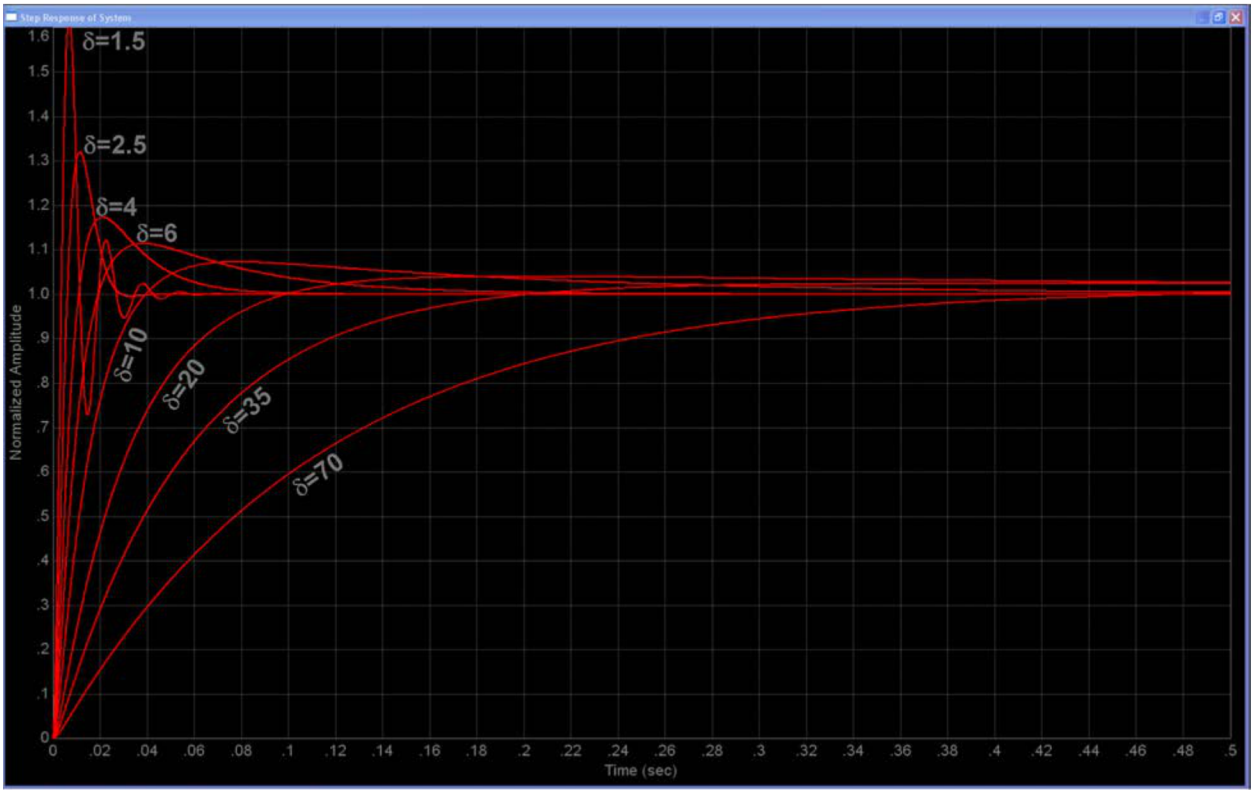


Figure 11-10. Step Response of Speed Controller as a Function of δ

So what should you do if you picked the lowest value you can tolerate for the damping factor, but you still aren't satisfied with the system response times? Your best recourse might be to increase the bandwidth of your current loop. But the problem with this is that it appears to be an iterative approach since you need to determine the current loop bandwidth first before you can determine what speed loop bandwidth is created from a given damping factor. However, we can take advantage of the fact that the frequency curves shown here can be normalized with respect to the current loop bandwidth irrespective of frequency. In other words, if your motor control system has a form similar to the one discussed in this section, irrespective of actual parameter values, you will get frequency curves (and transient step curves) that look like this, with only the frequency scaling (and time scaling) being different. So let's exploit this fact to develop a procedure which will minimize the iterative nature of the design process and allow us to set the current loop bandwidth as a function of the speed loop bandwidth:

- Pick the frequency response (-3 dB cutoff frequency) you want for your speed loop (BW_s).
- Using the shape of the curves in Figure 11-10, find the lowest value of damping factor that will produce a satisfactory response for your speed loop. We have found that it is OK to pick a damping factor with a little more overshoot than you prefer, since integrator clamping will remove most of it anyway. At this point, the scaling of the frequency and time axes is irrelevant.
- Calculate the required current loop bandwidth to support the design requirements using the following formula (obtained by curve fit analysis):

$$BW_c = \frac{K_p^{series}}{L} = BW_s\left(\delta + 2.16 \times e^{-\frac{\delta}{2.8}} - 1.86\right)(rad/sec) \quad (61)$$

Where:

- BW_c is the current controller bandwidth
- K_p^{series} = one of the current loop PI coefficients
- L = the motor inductance
- BW_s is the speed controller bandwidth

Copyright © 2013–2019, Texas Instruments Incorporated

- δ is the damping factor.

Proceed with calculating the four PI coefficients as discussed previously in this section.

EXAMPLE

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

- Rs = 0.4 ohms
- Ls = 0.6 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

The desired speed bandwidth = 800 rad/sec, and we would like a damping factor (δ) of 4. Find the required current loop bandwidth to support the speed loop bandwidth, and then calculate the four PI coefficients.

SOLUTION

The required current bandwidth can be found directly from Equation 61:

$$BW_c = \frac{K_p^{series}}{0.6E-3} = 800\left(4 + 2.16 \times e^{-\frac{4}{2.8}} - 1.86\right) = 2126\,rad/sec \tag{62}$$

From Equation 62, we find

$$K_p^{series} = 1.28 \tag{63}$$

Recall that

$$K_i^{series} = \frac{R}{L} = 667 \tag{64}$$

Also recall that

$$spdK_i^{series} = \frac{K_p^{series}}{\delta^2 \times L} = 133 \tag{65}$$

Finally, recall that

$$K = \frac{3P\lambda_r}{4J} = 162 \tag{66}$$

and,

$$spdK_p^{series} = \frac{K_p^{series}}{L \times \delta \times K} = 3.29 \tag{67}$$

The simulated speed transient step response for this example is shown in Figure 11-11 where the time axis is now scaled appropriately for this design example.

Copyright © 2013–2019, Texas Instruments Incorporated

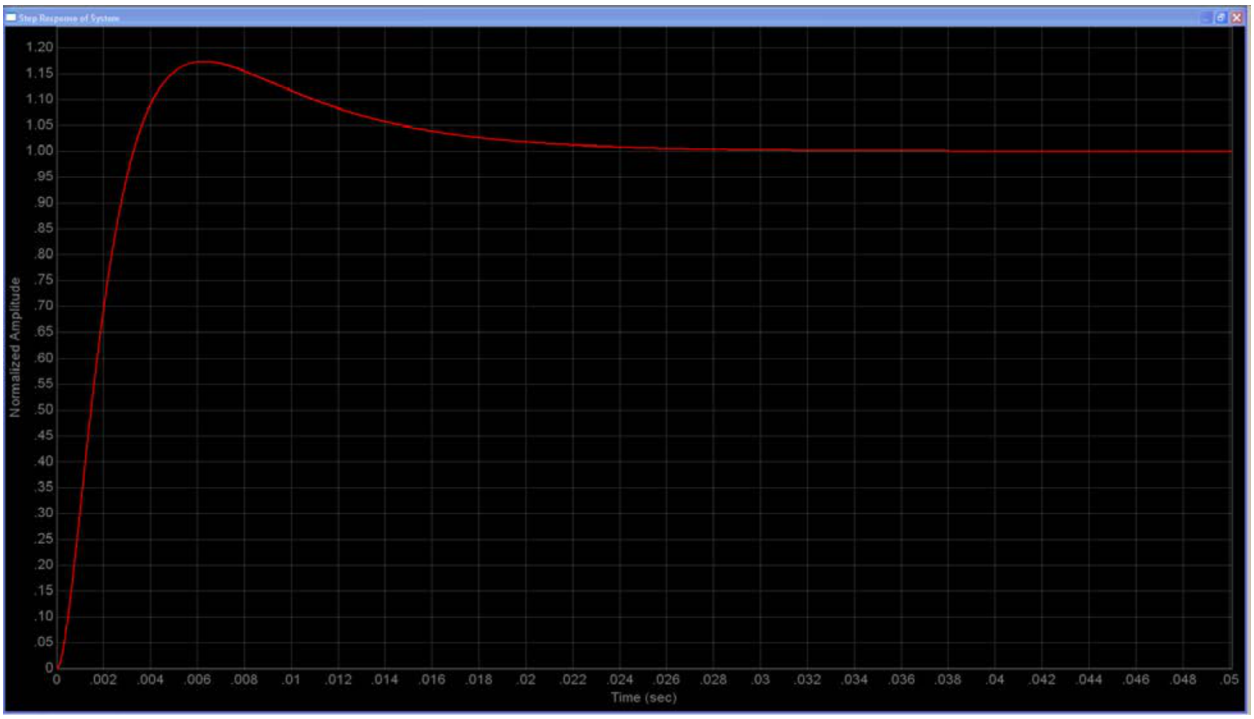


Figure 11-11. Simulated Step Response of Speed Controller Design from the Above Example

Our analysis so far has assumed that the only poles in the speed loop are the two at s = 0, and the one associated with the current controller. But what if other poles exist? For example, the speed feedback signal in many systems is often processed by a low-pass filter. So how does this affect our tuning procedure? This will be covered in the following section.

11.6 Considerations When Adding Poles to the Speed Loop

At the end of last section, we presented a problem that could potentially derail this whole discussion. Throughout the PI tuning sections, we have been discussing a way to find the values for the PI coefficients in a theoretical system where the speed loop contains two poles at "s" equals zero, and a third pole from the current controller. Usually there are one or more additional poles in the transfer function. For example, a very common deviation from this utopian situation is when the speed feedback signal requires filtering, as shown in Figure 11-12.

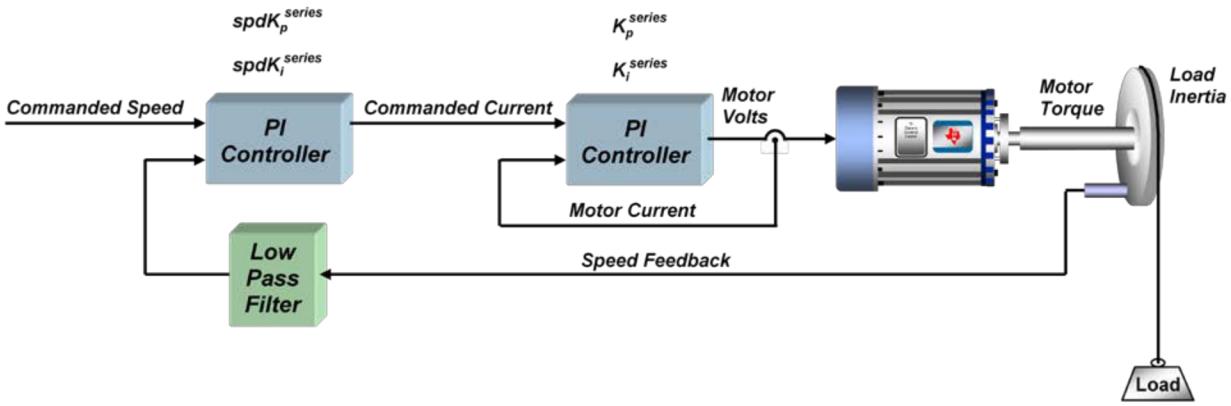


Figure 11-12. Speed Controller with Filtered Speed Feedback

Copyright © 2013–2019, Texas Instruments Incorporated

Creating a good quality high-bandwidth speed signal without spending too much design time and without adding too much system cost can be a real challenge. Techniques have been developed to glean information from the encoder edge transitions, and also using observer technology. But still, the speed signal is usually filtered. This alters the open-loop transfer function of the speed loop to the form shown in Figure 12-9.

$$GH(s) = \frac{K \times spdK_p^{series} \times spdK_i^{series}\left(1 + \frac{s}{spdK_i^{series}}\right)}{s^2\left(1 + \frac{L}{K_p^{series}}s\right) \times \left(1 + \frac{s}{K_{spd_filter}}\right)} \quad (68)$$

Where:

- K is a coefficient that contains several terms related to the motor and load
- $spdK_p^{series}$ and $spdK_i^{series}$ are the PI coefficients for the speed loop
- L is the motor inductance
- K_p^{series} is one of the PI coefficients for the current loop
- Kspd_filter is the pole of the speed feedback filter
- s is the Laplace frequency variable.

So what does this do to the tuning procedure? There are several dimensions to this problem, as well as possible solutions. The selected damping factor and the relative location of the poles all contribute to these challenges. So let's target these challenges one at a time.

The procedure outlined in the last section assumes that a suitable speed bandwidth and damping factor are chosen based on application requirements, and then using the equation presented in step 3, we can calculate the required current controller bandwidth to satisfy these design requirements. But it turns out that the pole calculated in step 3 defines the minimum frequency of any pole which occurs above the unity gain frequency. Armed with this knowledge, we can define a more general expression for $spdK_i^{series}$:

$$spdK_i^{series} = \frac{p}{\delta^2} \quad (69)$$

Where:

p = the lowest value pole above the 0 dB frequency in the speed open-loop frequency response.

The value of p could be set by the current controller, the speed filter, or something else. Since $spdK_p^{series}$ is referenced to $spdK_i^{series}$, then its value will also be potentially affected:

$$spdK_p^{series} = \frac{\delta \times spdK_i^{series}}{K} \quad (70)$$

If you can't meet the required frequency separation between the desired closed-loop speed bandwidth and p as dictated by your chosen value for δ, then something has got to give. It's like a water balloon—you can squeeze one part of the balloon, but it will pop out somewhere else. In this case, you can have your bandwidth at the expense of the damping factor, or vice versa.

The problem is exacerbated when the current controller pole and the speed filter pole are within a half decade of each other and δ is less than 3. With both poles so close to the 0 dB frequency and fighting together to bring down the phase margin, you will get a more underdamped response than you might otherwise expect. For example, Figure 11-13 shows the step response of a system where we used a damping factor of 2.5 to calculate the PI coefficients as described in the last section. The green curve assumes there is no filtering of the speed signal. The red curve shows the addition of a speed feedback filter where the value of the filter's pole equals the current controller's pole. The system is still stable, but the damping is much less than expected for a δ value of 2.5. At this point, we have two options, either increase the damping factor (and consequently lower the speed loop frequency response), or move one of the poles to a higher frequency. The cyan curve shows the first option where we increase the damping factor from 2.5 to 3.8 in order to bring the overshoot down to the original expected value. Unfortunately

Copyright © 2013–2019, Texas Instruments Incorporated

this reduces the bandwidth as indicated by the increased transient time. The yellow curve shows the latter option where we increase the value of one of the poles by a factor of 3 (about half a decade). In this case, the transient time is relatively unaffected, but the damping is still not as good as the green waveform. You can continue to increase the pole value, and at about one decade of separation you get a response that looks pretty close to the green waveform again. But in many cases, moving one of the poles this drastically has other negative effects on your system.

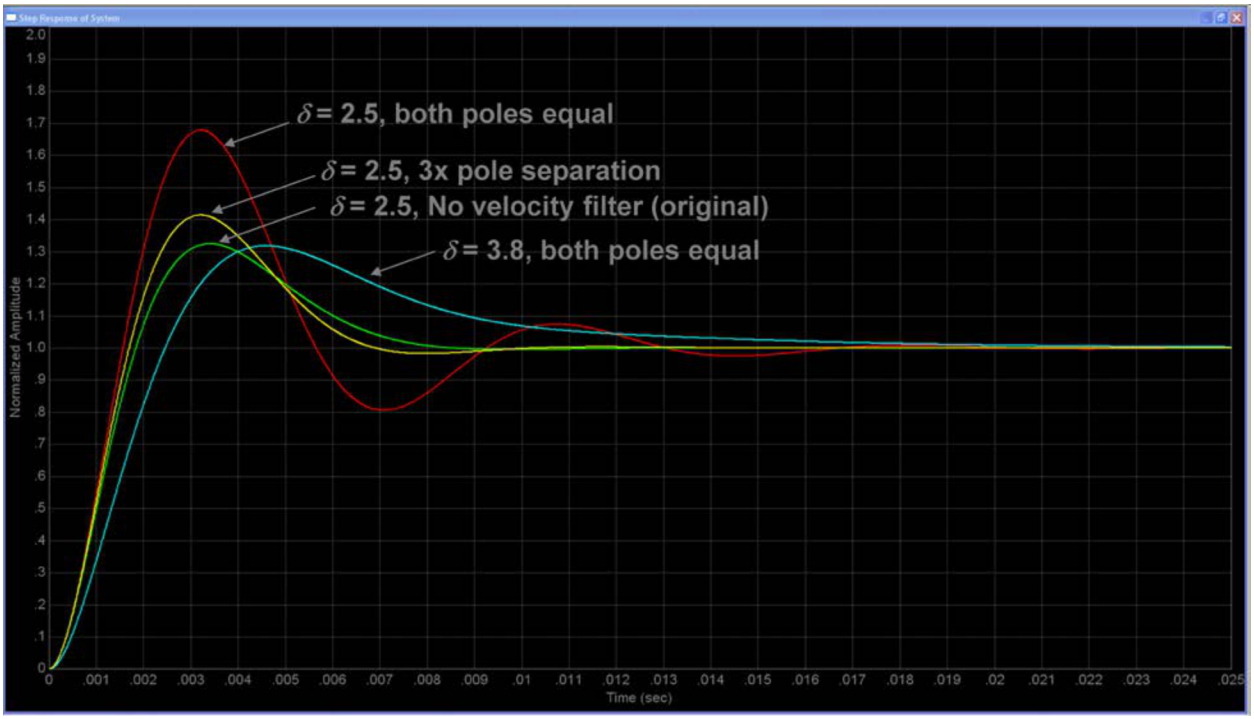


Figure 11-13. Step Response of a System with Variable Damping and Pole Placement

Up to now, we have only dealt with "small signal" conditions (i.e., linear operation with no saturation effects). But in the real world, step transient responses almost always involve saturation of the system's voltage or current levels, which tends to lengthen the response times. When this happens, you can increase the PI gains all you want, but it won't speed up the response. In fact, it will usually just make the overshoot worse, since the integrator is acting on a gained-up error signal, which it will just have to dump eventually. So how do we deal with this problem? Are we doomed to simply using low integrator gains? It turns out that there is another solution which doesn't involve changing your integrator gains, which we will cover in the next section.

11.7 Speed PI Controller Considerations: Current Limits, Clamping and Inertia

Up to now, we have only discussed the tuning problem in the context of a linear system. This is because under steady-state conditions when the system settles out, you will most likely find that you are operating in the linear region, and the AC signal content will be very small. Therefore, performing a small-signal (linear) analysis will tell you how stable your system will be when it is not operating in saturation. But in most real-life scenarios, the system will saturate because of limits on your voltage and/or current, especially under large transient conditions. This saturation effect can play an important role in the PI controller; especially the integrator. Since the maximum torque the motor can produce is limited by your current limit, the acceleration of the system is also limited. But the integrator doesn't know this, and it thinks it can make the motor speed up faster by increasing its output. This increased integrator output can't help the situation since the system is already saturated. All it does is create a very large output that will cause the system to overshoot when it does come out of saturation. For this reason, most PI integrator outputs are clamped to keep them from continuing to integrate needlessly when the system is saturated.

Copyright © 2013–2019, Texas Instruments Incorporated

A simple static clamping scheme is illustrated in Figure 11-14. The most common scenario is to set the clamp values equal to the PI output limit values. For example, the output limit of a PI controller that regulates speed is usually what sets your current limit value since the speed PI output is the reference input signal for the current PI controller. However, there is nothing that says that the integrator limit must equal the PI output limit, and many designs use different clamp values based on the specific application.

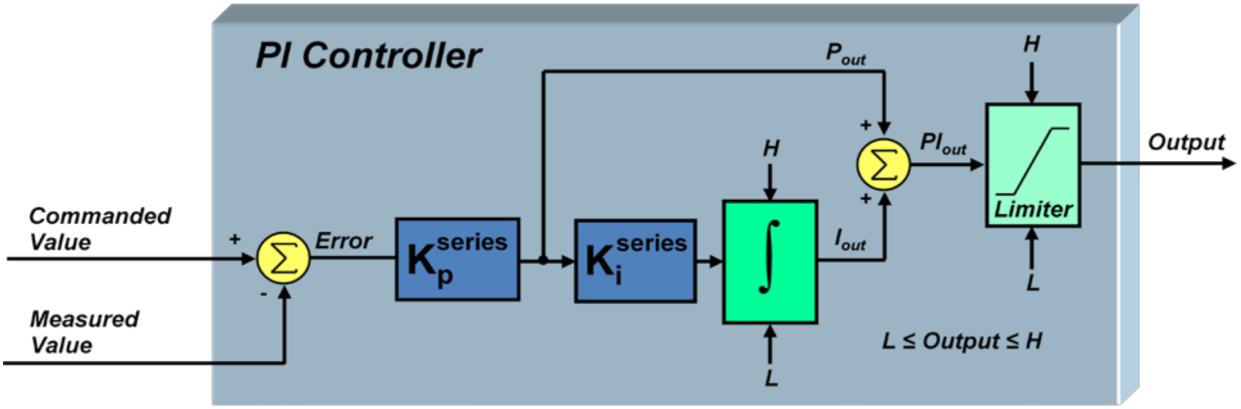
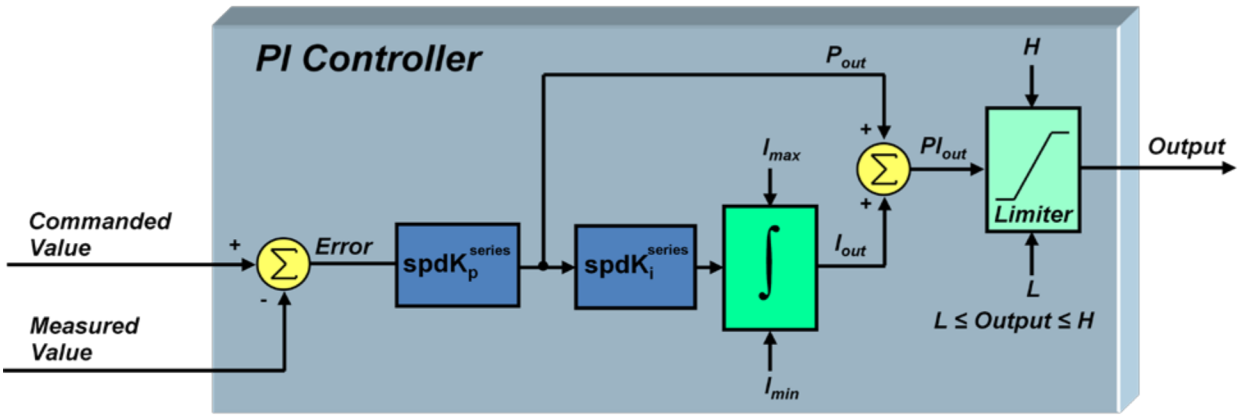


Figure 11-14. PI Controller with Static Integrator Clamping

Figure 11-15 shows a dynamic clamping scheme which provides superior performance over the static scheme. The thinking behind the design of this scheme is based on the rationale that if the system is already saturated by the P gain output, then why continue integrating? Only during conditions where changes in the integrator output would result in changes in the PI controller output is the integrator allowed to continue to integrate error unconstrained.



$$I_{max} = Max(H - P_{out}, 0)$$
$$I_{min} = Min(L - P_{out}, 0)$$

Figure 11-15. PI Controller with Dynamic Integrator Clamping

The effectiveness of integrator clamping can be seen by the simulated curves in Figure 11-16. Let's stimulate the system we designed in Chapter 5 with a commanded speed step from zero to a target speed of 1500 RPM. Shown are the effects of system overshoot under the conditions of no clamping, static clamping where the integrator clamp values equal the output clamp values and finally, dynamic clamping. As you can see, no integrator clamping at all is unacceptable as it results in extremely high overshoot which triggers further system saturation and oscillation. Static integrator clamping dramatically improves this situation. However, dynamic clamping improves performance even further, resulting in a 6 times improvement in the overshoot peak value compared to static clamping in this example.

Copyright © 2013–2019, Texas Instruments Incorporated

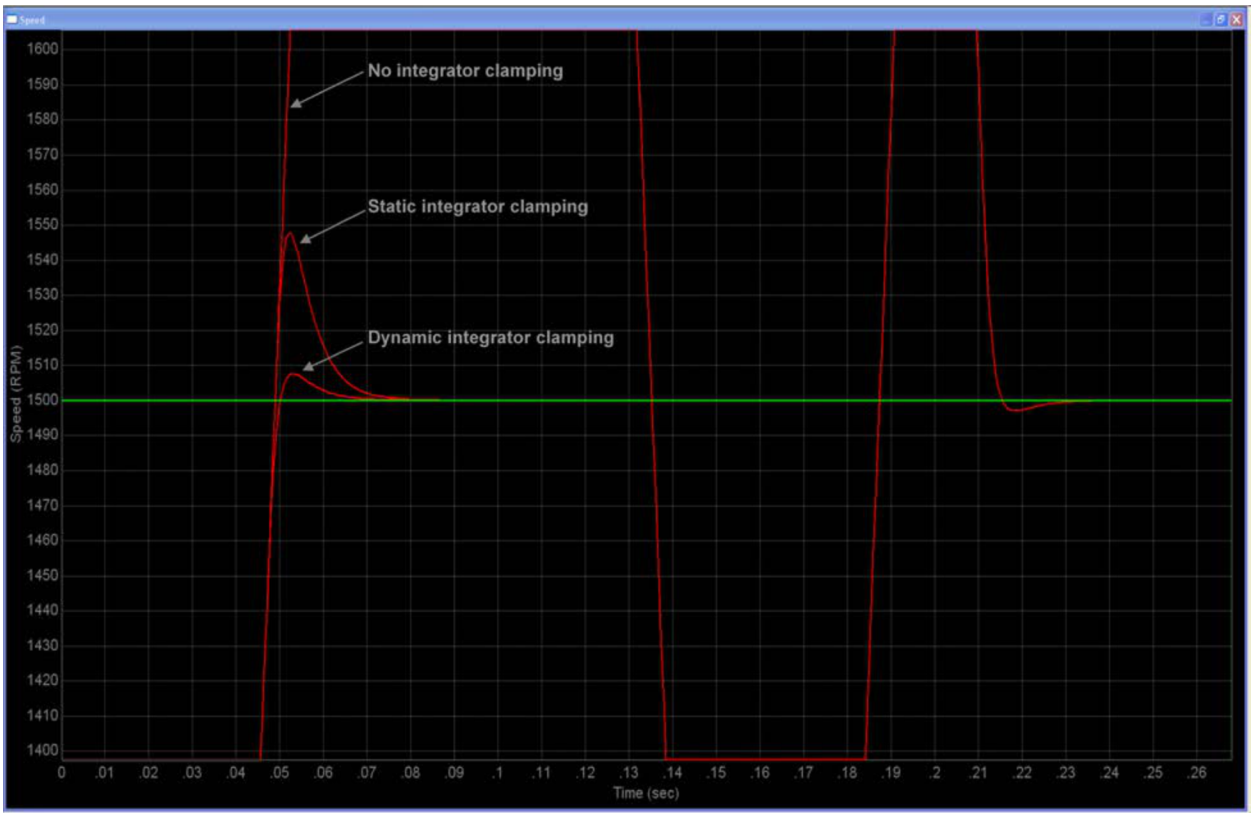


Figure 11-16. Example Comparison of Integrator Clamping Techniques

At this point, let's double-back and talk about a very important part of this whole discussion. Everything we have talked about in these seven sections is not very significant without knowing one critical piece of the system which is the inertia. Without this knowledge, there is no definitive way of stabilizing the speed loop. In many cases, you can calculate the inertia by knowing the form factor and mass distribution of your rotating load. If a gearhead is present on the motor shaft with a big enough gear ratio, the load inertia can often be ignored since transferred inertia is inversely proportional to the square of the turns ratio, and just deal with the motor inertia which is listed on most motor data sheets. If neither of these options is valid, there are several techniques used to measure inertia which usually involve some type of controlled acceleration, deceleration, or both. However, it is not common to see techniques which also take into consideration static torque loading on the motor shaft ("static," loads in this context mean loads which don't change as a function of time, such as friction or an elevator load). The following is a proposed (but at the time of this writing, untested) technique which should yield a better inertia estimate than the techniques mentioned above:

1. Design the current controller using techniques discussed in the PI tuning sections.
2. Set the PI coefficients for the speed loop to conservative values that will just allow spinning the motor up to speed (i.e., having sluggish dynamic response should not be a concern at this point).
3. Spin the motor up to a low speed and allow it to settle (so that inertia torque equals zero). Then take a reading of the average motor torque (Figure 11-17).
4. Repeat step 3 at successively higher speeds, and generate a graph of average torque readings as a function of speed (Graph 1). Record the average current required for the highest speed setting. Then turn off the motor and allow it to stop.
5. Disable the speed loop and using current mode only, apply about 1.2x to 1.5x the current from step 4 to the motor. As the speed hits each speed for which a torque value was recorded in step 4, record the torque again (Graph 2), and also take a time stamp.
6. Subtract graph 1 from graph 2 (this should be the acceleration torque only) (Graph 3).
7. For each point in graph 3, calculate the delta speed and delta time between the points before and after

Copyright © 2013–2019, Texas Instruments Incorporated

the target point. Divide delta speed by delta time to get the local acceleration value for that point.

8. For each torque value in graph 3, divide it by the local acceleration for that point from step 7, to create a graph of inertia (J) as a function of speed.
9. Average the inertia values at different speeds to obtain a single estimate for system inertia.

This process can be done a priori on a bench dynamometer test, or, if there is a way to measure torque in the control algorithm such as the torque output of InstaSPIN-FOC, this can be done as part of the commissioning process of the motor in its target application.

Up to now, we have only discussed PI tuning in generic terms which are independent of the control topology. In the next section, we will focus on some of the subtle points to consider when designing PI controllers for use in a Field-Oriented Control (FOC) system.

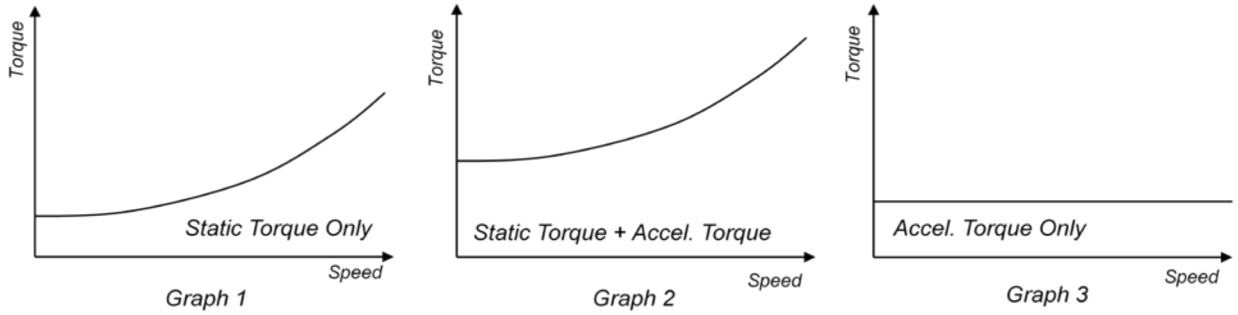


Figure 11-17. Average Motor Torque Readings

11.8 Considerations When Designing PI Controllers for FOC Systems

Let's see how the different PI tuning topics we have discussed so far apply to Field-Oriented Control (FOC) systems. Figure 11-18 shows a typical field oriented system which utilizes three PI controllers: two for controlling the quadrature components of current, and one for controlling the speed.

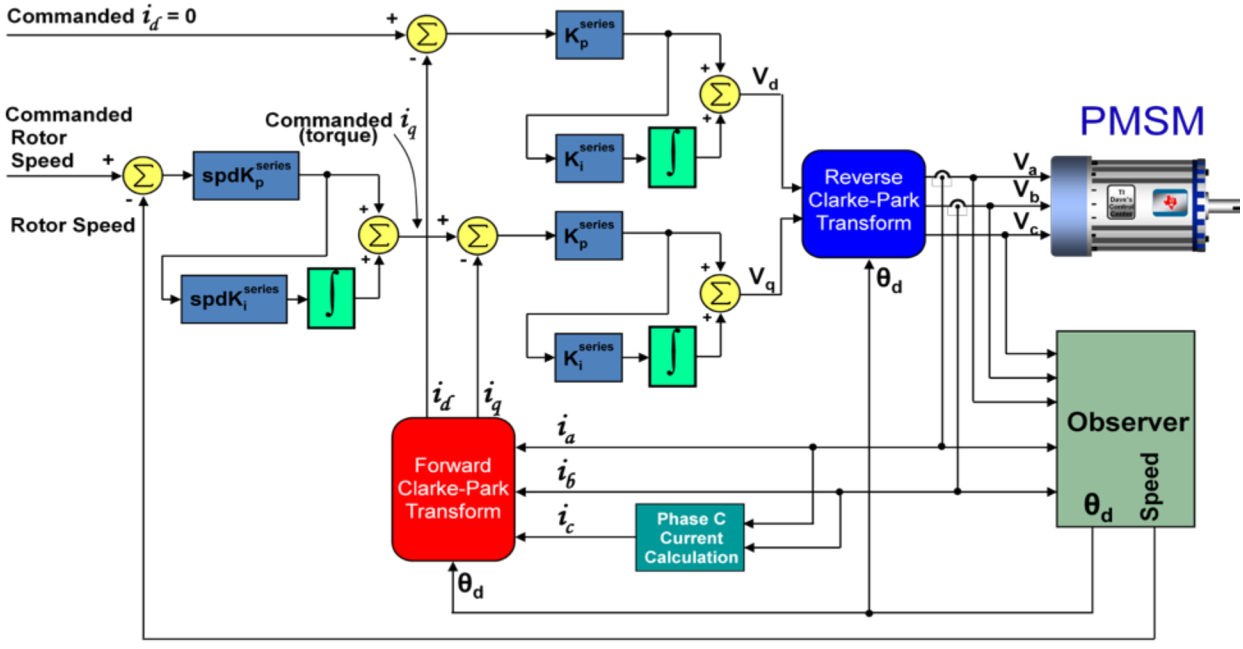


Figure 11-18. Typical FOC Speed Control of a PMSM

Copyright © 2013–2019, Texas Instruments Incorporated

The design of the speed controller doesn't change much in a field oriented system compared to other control algorithms. Just make sure to use the *q-axis* current controller values when calculating the coefficients for the speed controller. But there a subtle differences which affect how the current controllers should be designed, which are covered next.

11.8.1 FOC Differences Between Motor Types

The motor's equivalent RL circuit that is seen by the controller (which determines the PI coefficients) will vary depending on the motor type. For BLDC and Permanent Magnet Synchronous Motors (PMSMs), R is simply the stator resistance, and L is the stator inductance. But with AC Induction Motors (ACIMs), this is not the case. The equivalent inductance value that is needed to use for both axes is not the stator inductance value, but rather the "series" inductance (or sometimes called the "leakage" inductance) which is defined as follows:

$$L = L_s\left(1 - \frac{L_m^2}{L_s L_r}\right) = L_s \times \sigma \tag{71}$$

Where:

- L = the equivalent series inductance
- Ls = the stator inductance
- Lm = the magnetizing inductance
- Lr = the rotor inductance
- σ = the "leakage factor" of the induction motor

Also, the resistor value seen by the current controllers for an ACIM will be different between the d and q axes. For the d-axis controller, the equivalent resistance is simply the stator resistance Rs. However, for the q-axis, the equivalent resistance is the sum of the stator resistance plus the rotor resistance (Rs + Rr).

If these subtleties are not taken into consideration when calculating K_p^{series} and K_i^{series}, you could end up with a PI controller that is incompatible with your motor, resulting in less than optimal control.

11.8.2 Coupling Between Q-Axis and D-Axis

It turns out that the control of the d-axis and q-axis currents are not independent from one another. Within the motor, the q-axis current has an effect on the d-axis current and vice-versa. This is substantiated by the differential equations below for a PMSM.

$$i_d\left(R + DL_s\right) = V_d + \omega L_s i_q \tag{72}$$

$$i_q\left(R + DL_s\right) = V_q - \omega\left(L_s i_d + K_e\right) \tag{73}$$

Where:

- R = the stator resistance
- Ls = the stator inductance
- D = the differential operator
- ω = the electrical frequency
- Ke = the Back EMF constant

From Equation 72 we see that the d-axis current is not only affected by the output voltage of the d-axis current regulator (Vd), but also a voltage which is a function of i_q. From Equation 73, Vq is also competing with the voltage "ω(Ls i_d + Ke)" for control of the i_q level. For both regulators, this cross-coupling effect manifests itself as an unwanted disturbance which is most prominent during transient conditions at high speeds. To correct for this situation, feed-forward decoupling should be applied to each axis which exactly cancels these competing voltage terms. This correction results in each regulator acting on the equivalent of a simple RL circuit, just like we have with a DC motor. The result is the regulator topology shown in Figure 11-19. To judge the effectiveness of this technique, consider the simulation results of Figure 11-20 which show a step response in Q-axis current and how it affect the d-axis current, with and without decoupling compensation.

Copyright © 2013–2019, Texas Instruments Incorporated

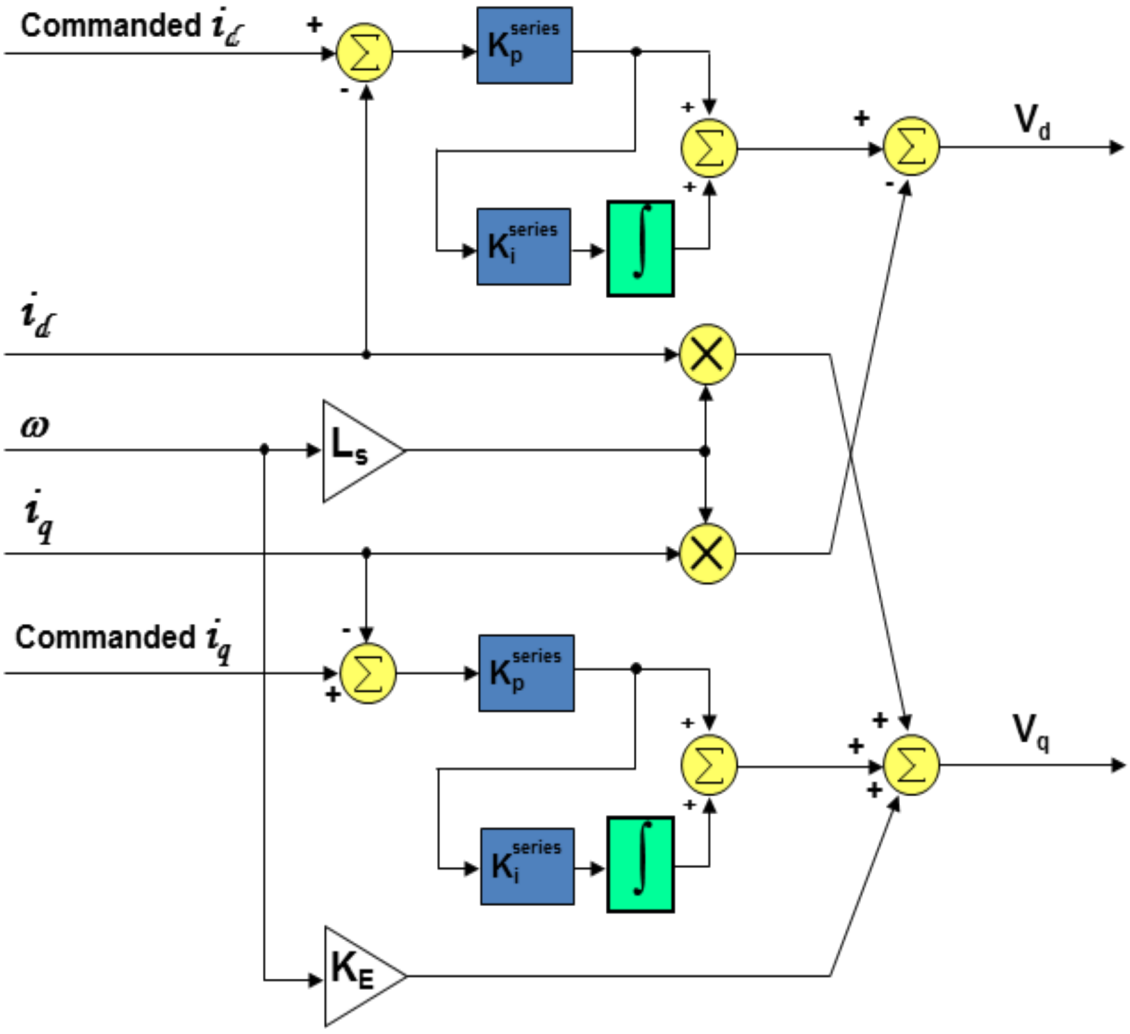


Figure 11-19. Decoupled PI Controllers for a PMSM

Copyright © 2013–2019, Texas Instruments Incorporated

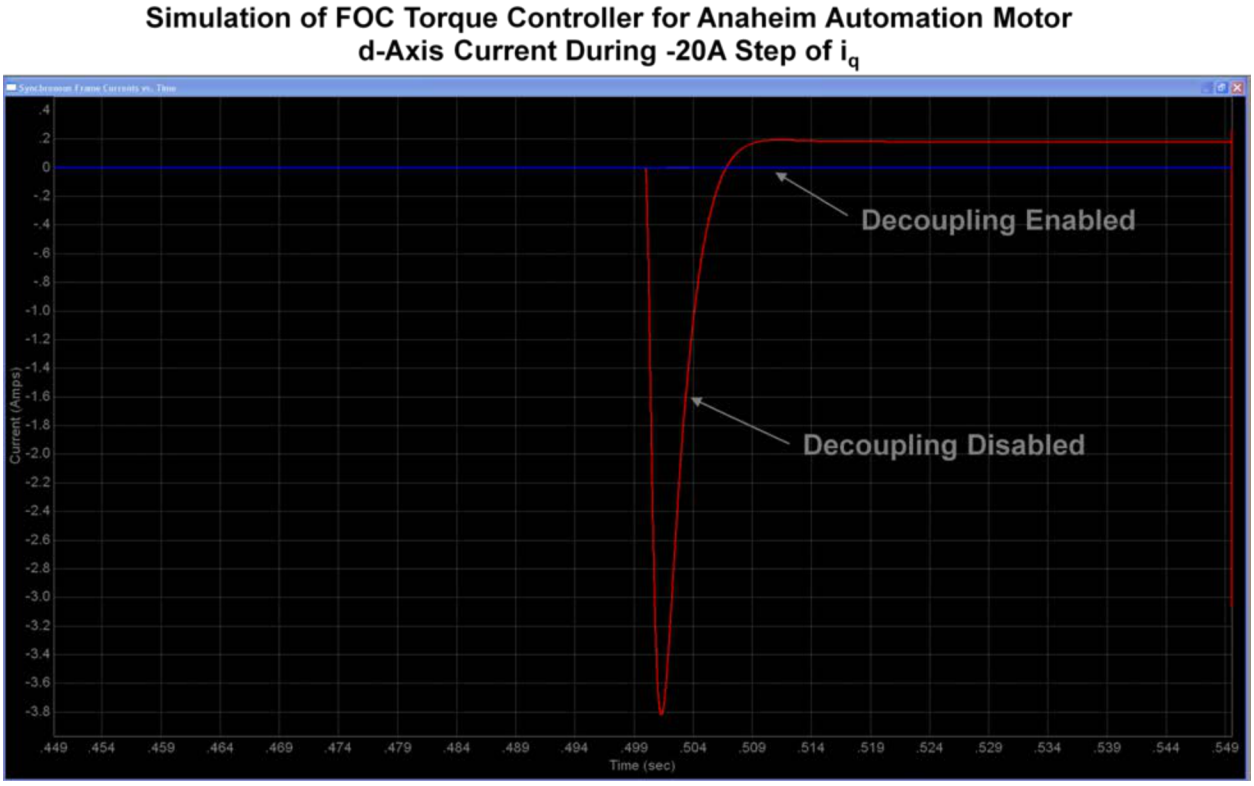


Figure 11-20. Simulated Effectiveness of Current Regulator Decoupling

For AC Induction Motors, the correction becomes a little more complicated. The differential equations defining AC induction motor operation are shown below:

$$i_d\left(R_s + DL_s\sigma\right) = V_d + \omega L_s\sigma i_q - \frac{L_m}{L_r}D\lambda_{rd} \tag{74}$$

$$i_q\left(R_s + DL_s\sigma\right) = V_q - \omega L_s\sigma i_d - \omega\frac{L_m}{L_r}\lambda_{rd} \tag{75}$$

Where:

- Rs = the stator resistance
- Ls = the stator inductance
- σ = the leakage factor defined in Equation 71
- D is the differential operator
- ω = the electrical frequency
- Lm = the magnetizing inductance
- Lr = the rotor inductance
- λrd = the d-axis rotor flux

Similar to the situation with a PMSM machine, we see that there are other voltages besides Vd and Vq competing for control of i_d and i_q respectively. As a result, compensation voltages are added to Vd and Vq which nullify these other voltage terms. This results in each axis acting on the equivalent of a simple RL circuit, again just like we have with a DC motor. But with an ACIM, please remember that the inductance value used for calculating K_p^{series} and K_i^{series} is the stator inductance multiplied by the leakage factor σ, as indicated by Equation 71. The compensation block used to provide correction voltages to the outputs of the I_d and I_q regulators is shown in Figure 11-21.

Copyright © 2013–2019, Texas Instruments Incorporated

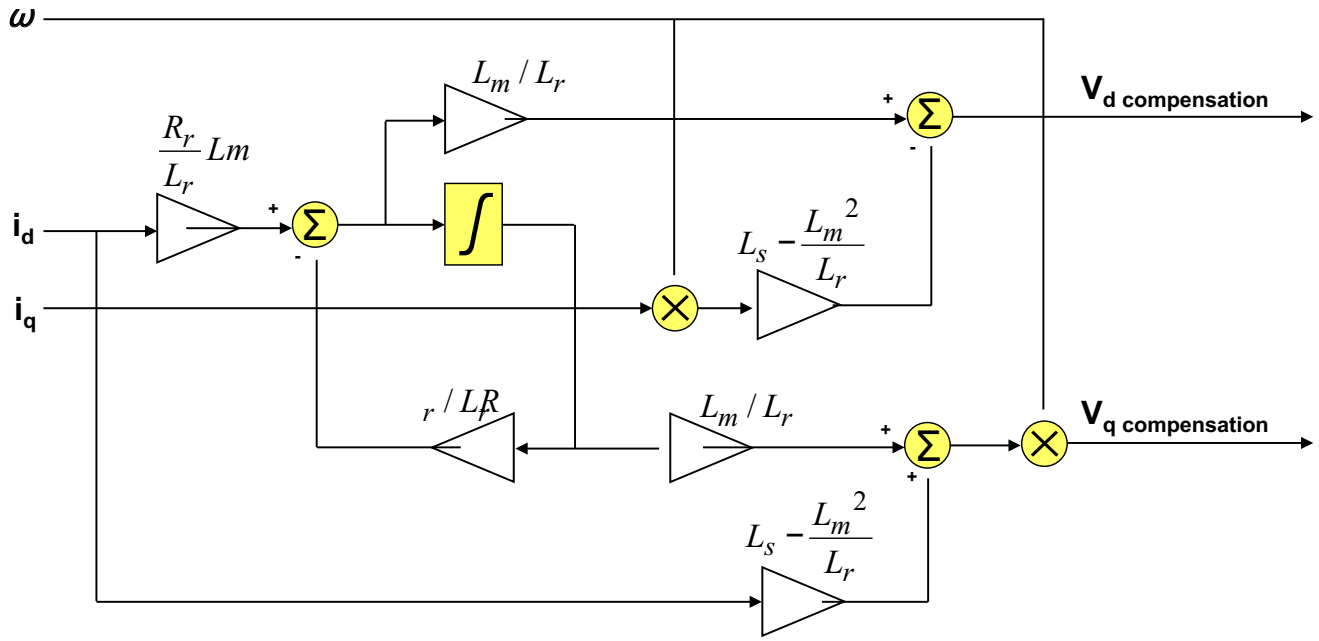


Figure 11-21. Compensation Block Used for Axis Decoupling with ACIMs

11.9 Sampling and Digital Systems Considerations

Throughout the PI tuning sections we have discussed a practical and efficient way to tune the PI controllers in a cascaded speed loop by simply specifying the desired bandwidth of the speed loop, and a factor which determines the desired damping of the system. From these two parameters, plus a rudimentary knowledge of some of the motor and load parameters, the PI coefficients for the speed loop and the inner current loop can be calculated. But nowhere have we discussed what limits are imposed upon the system, especially when dealing with a digital system. Obviously, to get a stiffer response, we would like higher gains, which also translate into higher bandwidth. But how high can we go?

To answer this question, take a look at Figure 11-22, which shows a high level view of a digital FOC based Variable Frequency Drive (VFD). To simplify the discussion, we will assume that the entire control loop is clocked by a common sampling signal, although this limitation is not imposed on real-world applications. In many cases, the speed loop is clocked at a much lower frequency than the current loop, since the frequencies associated with the speed loop are typically much lower.

Copyright © 2013–2019, Texas Instruments Incorporated

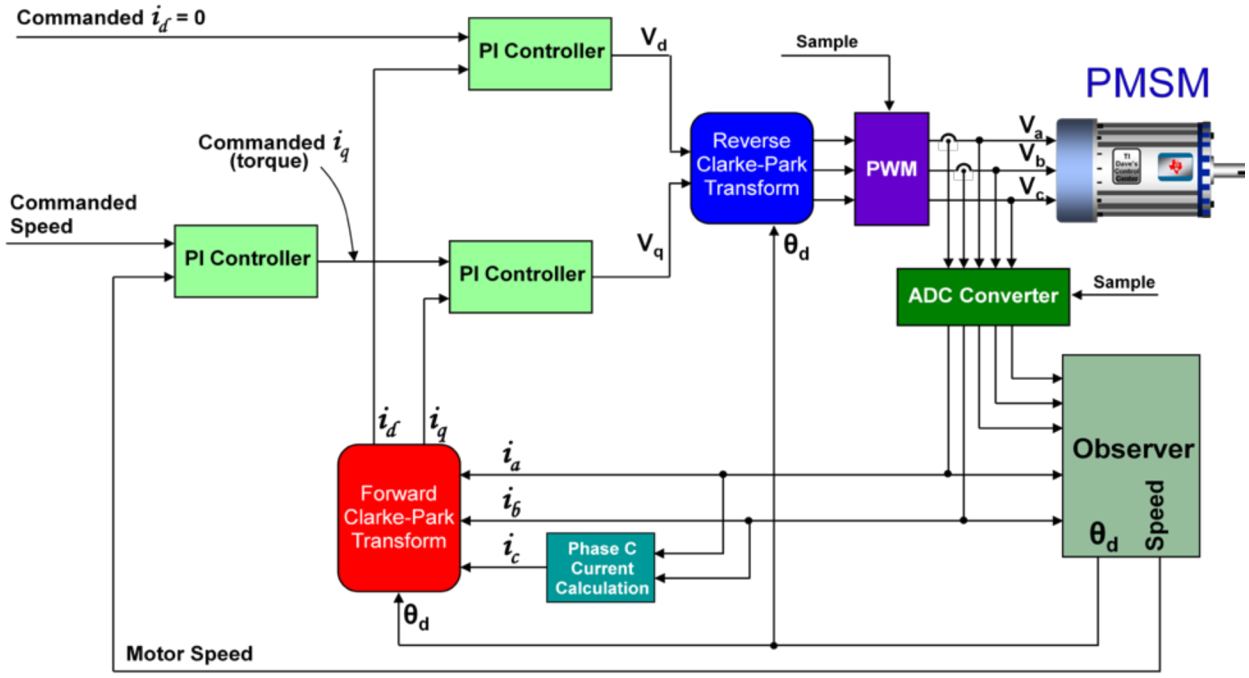


Figure 11-22. Digital Field-Oriented Control System for a PMSM

In an analog system, any change in the motor feedback signals immediately starts having an effect on the output control voltages. But with the digital control system of Figure 11-22, the motor signals are sampled via the ADC at the beginning of the PWM cycle, the control calculations are performed, and the resulting control voltages are deposited into double-buffered PWM registers. These values sit unused in the PWM module until they are clocked to the PWM output at the start of the next PWM cycle. From a system modeling perspective, this looks like a sample-and-hold function with a sampling frequency equal to the PWM update rate frequency. The fixed time delay from the sample-and-hold shows up as a lagging phase angle which gets progressively worse at higher frequencies. Figure 11-23 shows a normalized frequency plot for a sample-and-hold function, where the sampling frequency is assumed to be 1. The phase plot is the most important graph, as it shows that the phase delay from the sample-and-hold can reach down into frequencies much lower than the sampling frequency. For example, even at one tenth the sampling frequency, the S&H is still affecting a phase shift of -18 degrees.

Copyright © 2013–2019, Texas Instruments Incorporated

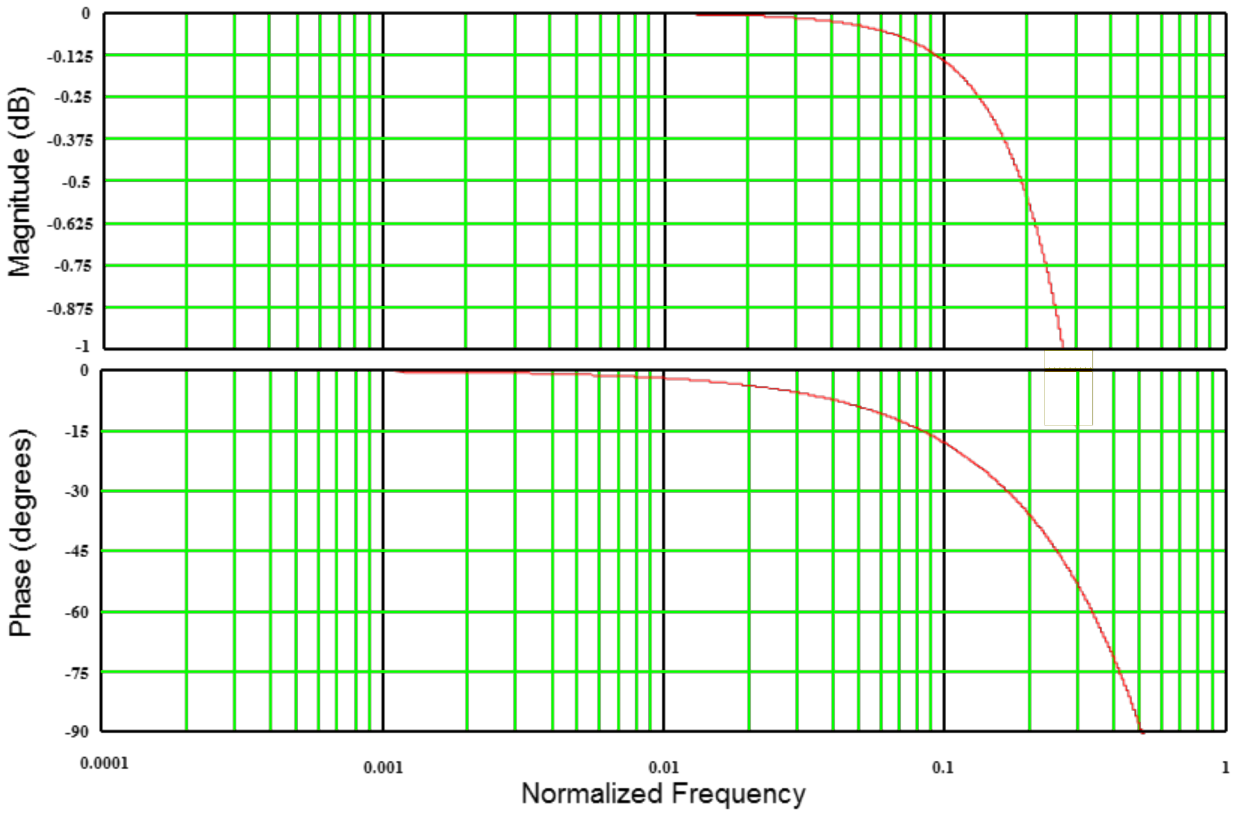


Figure 11-23. Magnitude and Phase Plots for a Sample-and-Hold

Since the current controller processes higher bandwidth signals than the speed loop, it is usually the current loop that suffers most by the S&H effect of the PWM module. Since the S&H is in series with the signal path for the current loop, its magnitude and phase contributions add directly to the open-loop response for the current controller. If we rewrite the equation for the open loop response of the current controller (assuming we have already made the substitutions recommended in the PI tuning sections), we end up with Equation 76.

$$G_{loop}(s) = PI(s) \times \frac{I(s)}{V(s)} = \left(\frac{BW_c}{s}\right) \tag{76}$$

Where:

BW_c is the chosen closed-loop bandwidth for the current controller.

The 0 dB frequency obviously occurs when s = BW_c. The single pole at s = 0 implies that the 0 dB frequency will have a 90 degree phase margin. While there is no magic ratio that exists, it is usually preferred to use the rule of thumb that the sampling frequency should be at least 10 times the bandwidth (BW_c) of the current controller. This ensures that the impact of the S&H's phase delay will only subtract 18 degrees from the phase margin of the current controller, resulting in a very stable 72 degree margin. You can obviously have a higher sampling frequency if desired, but this typically comes at the expense of a more expensive processor with higher MIPS.

Finally, let's look at the other end of the frequency range. At low frequencies, viscous damping may affect your speed loop response times by changing the phase margin at the 0 dB frequency. Recall from Section 11.3, the following transfer function between motor torque and load speed was established:

$$\frac{\omega(s)}{T(s)} = \frac{1}{J}\frac{1}{s} \tag{77}$$

Copyright © 2013–2019, Texas Instruments Incorporated

However, when viscous damping is present, it uses part of the motor's torque to do work to move fluid. Since viscous torque is directly proportional to the speed of the load, we can rewrite Equation 77 to be:

$$\frac{\omega(s)}{T(s)} = \frac{1}{Js + k_v} = \frac{1}{k_v\left(\frac{J}{k_v}s + 1\right)} \tag{78}$$

Where:

kv is the viscous damping factor.

As you can see from Equation 78, adding the viscous damping term moves the pole that was at s = 0 to s = kv/J. Figure 11-24 shows how the addition of viscous damping changes the load's Bode plot.

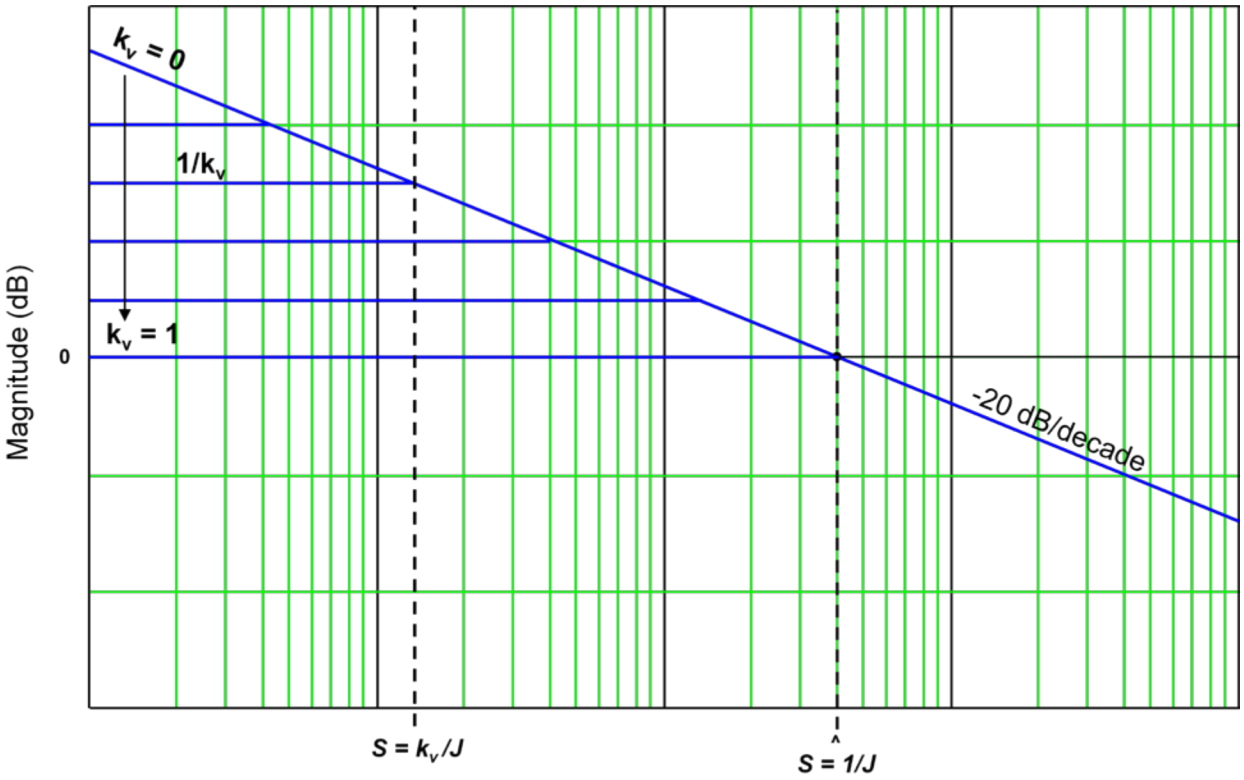


Figure 11-24. Effect of Viscous Damping (kv) on the Load Bode Plot

From Figure 11-24, we see that as viscous damping increases from zero, low frequency gain plateaus to a value of 1/kv. The net effect on the phase plot is to add more phase margin at lower frequencies, since the phase lag of a load with viscous damping will always be less than a load with inertia only. As a result, stability should actually improve for non-zero values of kv. However, the response time may take a hit, depending on where the speed open-loop 0 dB frequency is with respect to the pole frequencies shown in Figure 11-24. So if your system response is sluggish and the motor doesn't seem to put out as much torque as it is rated for, you could have excessive viscous damping in your system.

Before closing out this series of PI Tuning sections, there is one more topic to be discussed. In many cases an engineer correctly calculates PI coefficients. After using those coefficients in their code, the motor spins out of control, or sit there and does nothing. So what happened? It is most likely the fault of one of the following situations:

Copyright © 2013–2019, Texas Instruments Incorporated

11.9.1 Sampling Period Considerations in the Integral Gain

It was forgotten to take into consideration the sampling frequency effect on the I gain term. Figure 11-25 shows how to implement a typical integrator for a PI controller. To scale the output to match what an analog integrator would provide, we must multiply the signal by the sampling period "T". In order to avoid two separate multiply operations, most code examples simply lump T together with the I gain term. If T is not accounted for, the integrator gain will be much larger than you anticipated.

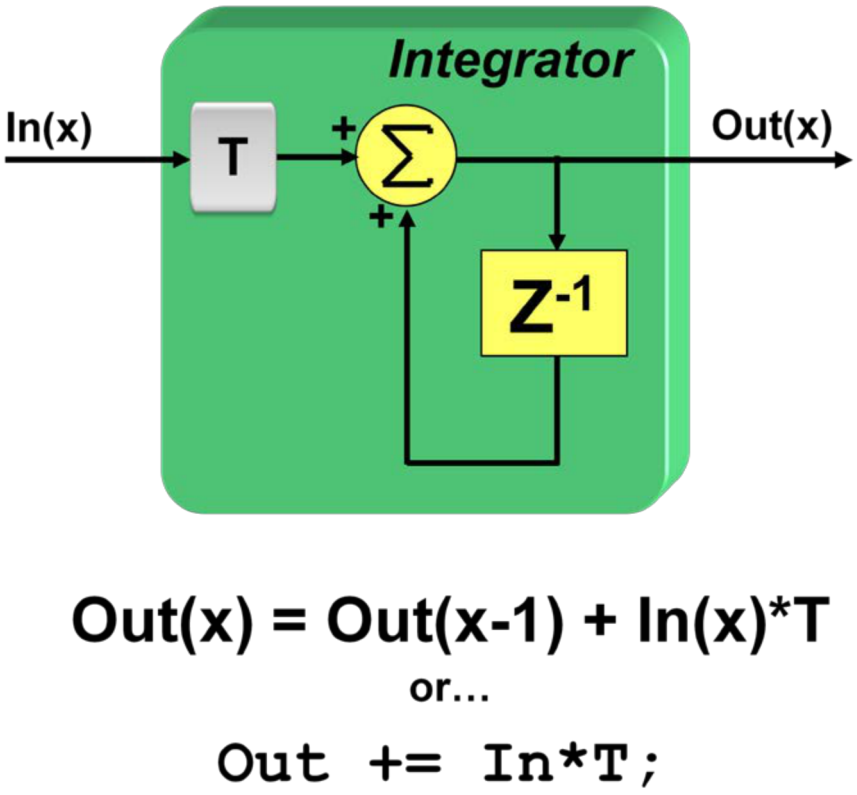


Figure 11-25. Typical Implementation of a Digital Integrator

11.9.2 Number Format Considerations

Up until now we have assumed there are no limitations on the number format itself. If a floating-point processor is used, then there is no need to worry about the fractional component of the PI terms. But most motor control applications are implemented on fixed-point machines for cost reasons. The good news is that TI has developed a linkable library which is in ROM of most C2000 processors which solves this problem. It is called the "IQ Math" library which stands for "Integer Quotient". This allows the user to handle floating point values with ease on a fixed-point machine without suffering from the performance hit you typically get with a full floating-point support device. IQ math creates a new variable type in the code which is designated by an "IQ" followed by a number. For example, say you have a 32 bit variable which is typed as an "IQ24" variable type. This means that any variable of this type is assumed to have a 24-bit fractional component, and an 8-bit integer part. But what occasionally happens is that someone copies TI code into their design without realizing that the coefficients are represented in IQ format. For example, if the I-gain was calculated to be 10,000 (0x00002710 in IQ0 format), but it is not realized that the code assumed the variable to be in IQ24 format, you will end up with an integrator gain of 0.596E-3 instead of 10,000. The two values are obviously very different. If the same mistake is made for all of the PI coefficients, the motor will most likely just sit there and do nothing since all the gains are way too low. So, it is advised to make sure that the numerical format your coefficients are in is well known.

Copyright © 2013–2019, Texas Instruments Incorporated

11.9.3 PI Coefficients Scaling Considerations

Finally, the scaling of the PI coefficients throughout this series of PI Tuning sections has been done assuming we want to represent real system values throughout the signal chain. For example, the output of the speed PI controller equals the actual input reference current in amperes for the current controller. The output of the current controller equals the actual voltage applied to the motor windings. But in many designs, the PI controller outputs are normalized to per unit scaling where a value of 1 represents the maximum value possible, and a value of -1 represents the minimum value possible. For example, a current regulator's output might be scaled in such a way that 1 corresponds to 100% PWM, and -1 corresponds to 0% PWM. In these cases, it is required to know the exact scale factor between the PI output and the actual system parameter you are controlling so that you can adjust the PI coefficients accordingly.

Copyright © 2013–2019, Texas Instruments Incorporated

InstaSPIN-MOTION Controllers

For most motion systems the speed and/or position of that system requires regulation. The industry standard speed controller is a PI controller, as discussed in Chapter 11. There are a number of inherent deficiencies with a PI controller.

- It has multiple parameters that need to be adjusted in order to tune the control for a specific speed and load operating point. These multiple parameters produce a multidimensional solution set, and the gains are usually determined experimentally, which makes tuning difficult.
- The range of speed and load that work for that specific tuning can be very small. If your system is highly dynamic and has many different speed and load operating points, you might need to tune a PI controller for each point.

The SpinTAC speed controller solves these challenges. SpinTAC provides advanced speed and position control and features Active Disturbance Rejection Control (ADRC), which reduces all gains to a single tuning parameter. ADRC accommodates for high degree of model uncertainties, which means that it is robust against system variations.

Disturbance is defined as any undesired behavior in the system. Error resulting from unmodeled dynamics and disturbances are estimated and compensated by a SpinTAC controller. The controller is unique in that it treats any undesired behavior of the system as disturbance that can be estimated and rejected. This allows SpinTAC controllers to control a wide range of positions, speeds and loads with a single tuning parameter.

The single tuning parameter, called bandwidth, determines the stiffness of the system and dictates how aggressively the system will reject disturbances. This single parameter makes it very easy to adjust the tuning of the SpinTAC speed controller.

The major considerations in tuning a controller for dynamic systems are stability and performance.

Topic **Page**

12.1 Stability **422**
12.2 Software Configuration for the SpinTAC Velocity Control **428**
12.3 Optimal Performance in Speed Control **431**
12.4 Software Configuration for SpinTAC Position Control **440**
12.5 Optimal Performance in Position Control **443**

Copyright © 2013–2019, Texas Instruments Incorporated

12.1 Stability

Stability of a control system is a safety issue in engineering systems. There are multiple definitions of system stability: Lyapunov stability, bounded-input bounded-output stability, and input-to-state stability. For simplicity, in this document, the criterion of stability is Lyapunov asymptotic stability, which means that the system has the nature to converge to the equilibrium point asymptotically.

In speed control, the equilibrium point is the target speed of a step response, or the speed trajectory while tracking a changing reference speed. In position control, the equilibrium point is either the end position of a step response or the changing position reference during a transition.

One simple way to assess the stability of the system is to see if the step response eventually converges to the setpoint. Typical step responses of stable and unstable systems are illustrated in Figure 12-1.

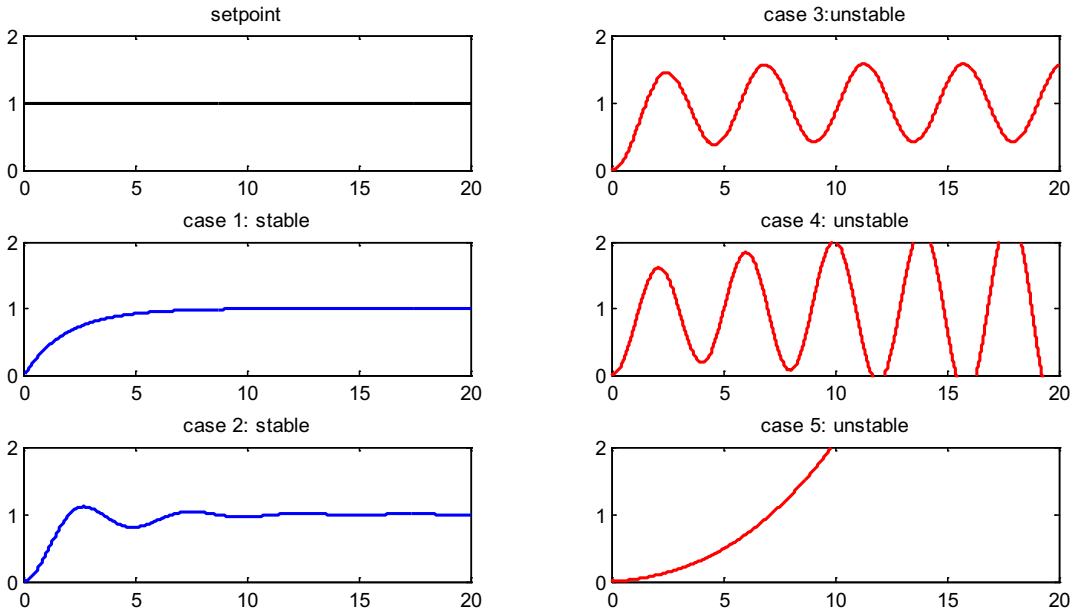


Figure 12-1. Typical Step Responses of Stable and Unstable Systems

The figures show the system responses when a unit step input is applied. Top left is the set point input signal. Cases 1 and 2 are stable systems; Cases 3, 4, and 5 are unstable systems. Case 3 is defined as marginally stable in some instances since the response is bounded oscillation.

12.1.1 Quantifying Stability

Classic control design models the system and derives the linear expressions close to the operating point, and uses Bode analysis to assess the stability through the gain margin and phase margin. Gain margin is the negative of the magnitude curve value at the frequency where the phase curve crosses -180°. Phase margin is the phase curve value above -180° at the frequency where the magnitude curve crosses 0 dB.

In order to allow the system to tolerate some non-linearity and model mismatch, typically 6- to 12-dB gain margin and 30- to 45-degree phase margin are needed.

12.1.1.1 SpinTAC Velocity Control Stability

Given a motor's speed loop dynamics without consideration of uncertainties such as resonant mode, sample time, and output saturation, the open loop Bode of a speed loop controlled by SpinTAC is always stable, which can be illustrated by the open loop Bode in Figure 12-2. The phase curve does not cross -180°, which means there is no limit on the gain margin. The phase margin is always positive.

Copyright © 2013–2019, Texas Instruments Incorporated

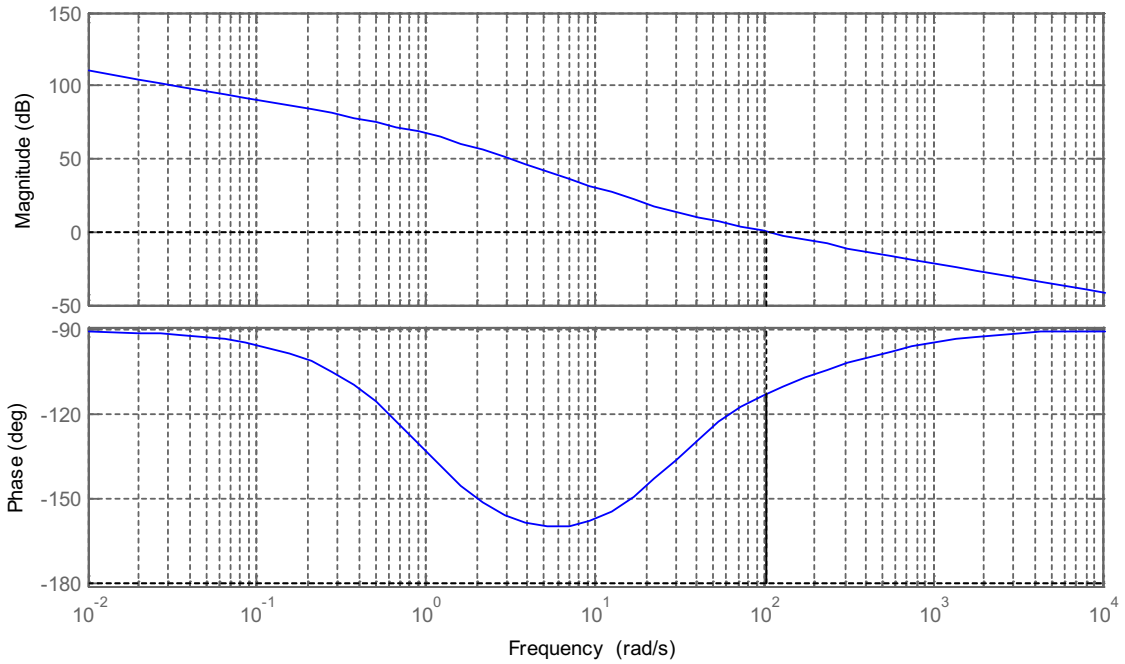


Figure 12-2. Typical Open Loop Bode of SpinTAC Velocity Control

However, most mechanical systems have resonant mode, typically at high frequency, which will cause 180° change in the phase curve and spikes in the magnitude curve.

The output saturation and sample time will also limit the adjustable range of the controller gains in order to keep system stable.

12.1.1.2 SpinTAC Position Control Stability

SpinTAC Position Control controls both the position loop and the speed loop.

Position control is more complex than speed control. There is a -180° cross-over in the phase curve, where the gain margin is measured. SpinTAC Position Control gives a fairly good negative gain margin to tolerate system changes. Typically, this negative gain margin represents the degree to which the configured inertia value can be greater than the real system inertia if the inertia is measured incorrectly (see Figure 12-3).

Copyright © 2013–2019, Texas Instruments Incorporated

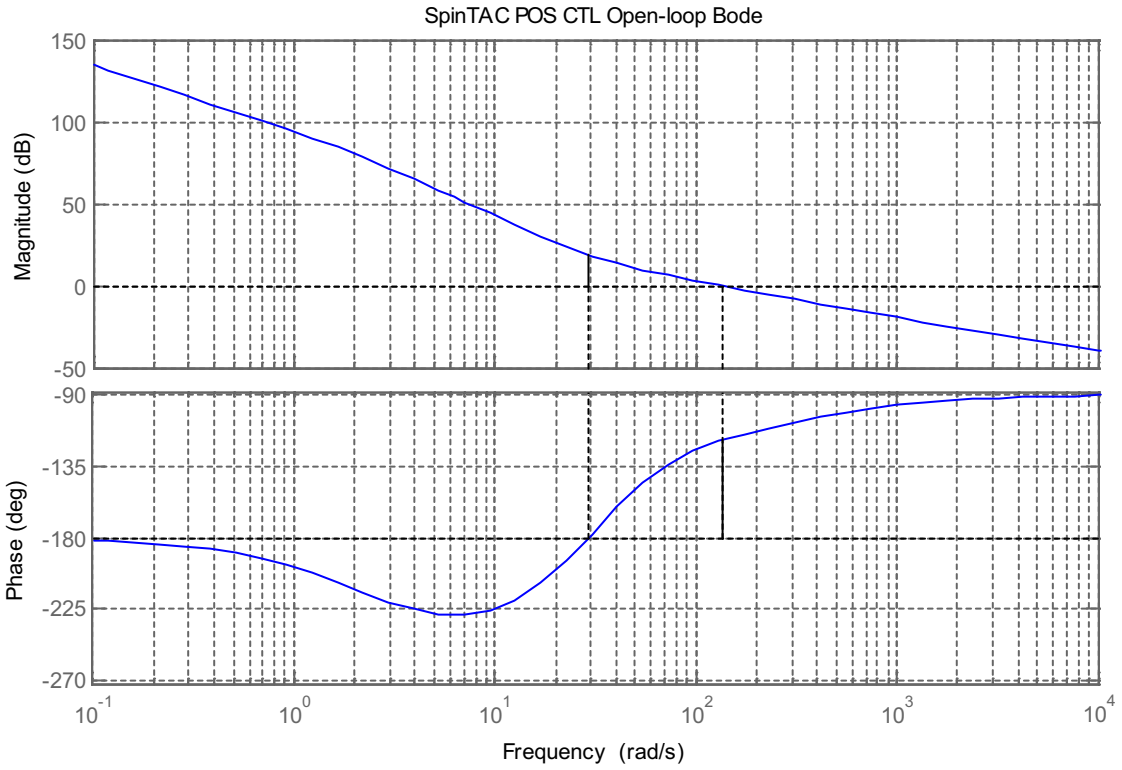


Figure 12-3. Typical Open Loop Bode of SpinTAC Position Control

Like speed control, position control is also subject to high-frequency resonant mode and noise, which limits the adjustable range of the controller gains in order to keep the system stable. These are general control design considerations for any type of controller.

12.1.2 Performance

The performance of a controller is usually evaluated by two criteria: reference tracking performance and disturbance rejection performance.

Reference tracking performance shows how closely the system can follow the desired trajectory. In cases where the setpoint changes, it shows how fast the system can reach a new setpoint with reasonable overshoot.

Disturbance rejection performance shows how little deviation the system can have when a disturbance is applied to the system and how fast the system can compensate for it.

The performance of a controller can be evaluated in time domain and frequency domain.

12.1.2.1 Frequency Domain Analysis

If the approximate linear model of the system is achieved, the performance can be evaluated in frequency domain. The purpose of this section is to visually illustrate the SpinTAC Velocity Control and SpinTAC Position Control performance in engineering language (Bode analysis) not to ask users to do the analysis of a given system.

The SpinTAC Velocity Control is designed to optimize disturbance rejection performance and trajectory tracking performance together and tune the control with one single parameter: the bandwidth. The typical reference tracking performance Bode and disturbance rejection performance Bode are illustrated in Figure 12-4.

Copyright © 2013–2019, Texas Instruments Incorporated

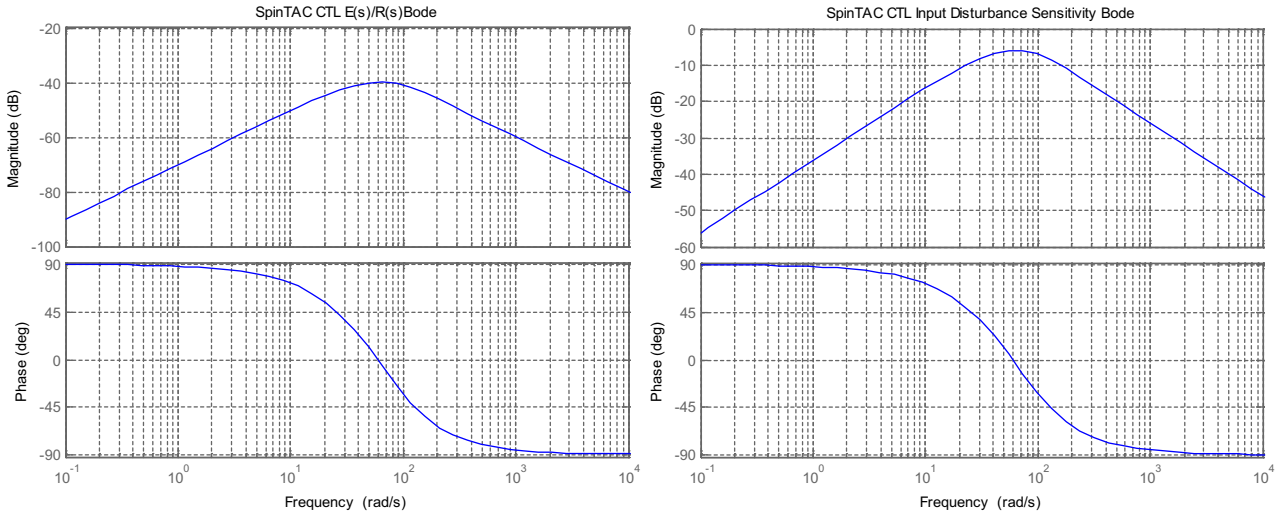


Figure 12-4. Typical Performance Bode of SpinTAC Velocity Control

As shown in Figure 12-4, the magnitude of Error/Reference Bode and Input Disturbance Sensitivity Bode are negative values with the unit dB. The more negative values on the magnitude curves, the better the system performance.

The performance analysis of SpinTAC Position Control is similar to SpinTAC Velocity Control. The typical performance Bodes are shown in Figure 12-5.

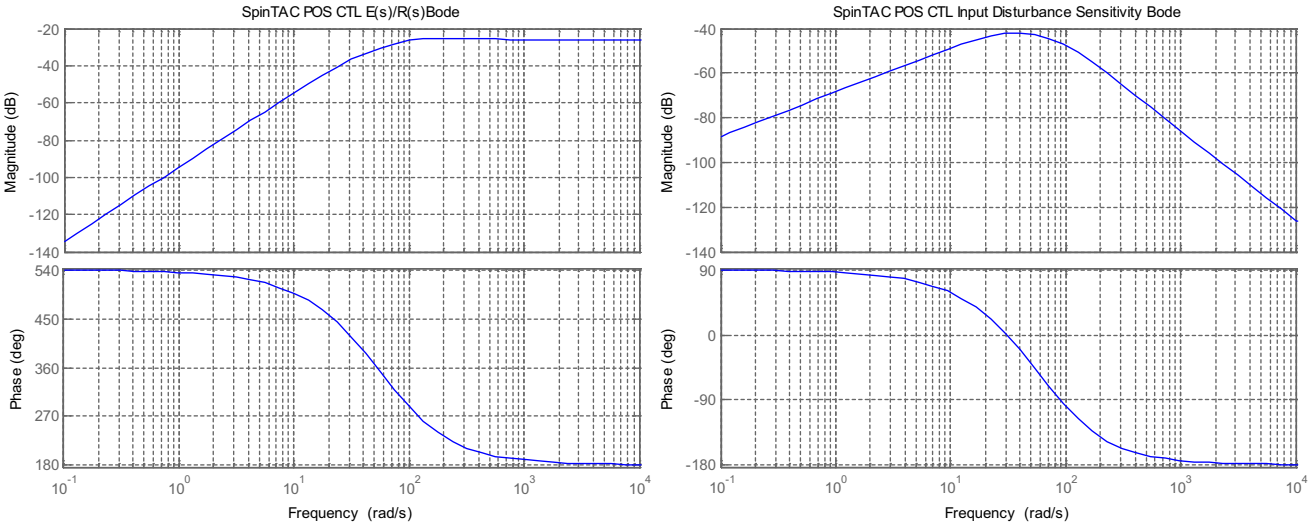


Figure 12-5. Typical Performance Bode of SpinTAC Position Control

12.1.2.2 Time Domain Analysis

The performance can be easily evaluated in the time domain. The time domain common criteria are listed in Table 12-1.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 12-1. Time Domain Common Criteria

Performance Criteria	Description
Overshoot	Maximum value deviated from the setpoint after first crossing of the setpoint
Settling Time	The time from start to finally entering a defined percentage (typically 2% ~ 5%) range around the step setpoint,
Maximum Absolute Error (MAE)	Maximum absolute value of deviation from the setpoint, indicate the worst case value
Integral Absolute Error (IAE)	Integrated absolute value of deviation from the setpoint, indicates the deviation over time

12.1.3 Trade-Off Between Stability and Performance

Typically there is a trade-off between system stability and performance. Real systems always have noise and uncertainties (high-order unknown dynamics such as resonant mode). Aggressively-tuned controllers exhibit better performance. This is true until the controller reaches the level that the system approaches unstable conditions or the noise allowed into the system is too high and degrades the performance.

12.1.4 Tuning the SpinTAC Controller

With single coefficient tuning, SpinTAC allows you to quickly test and tune your speed or position 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 time. Multi-variable PID based systems often require a dozen or more speed and load tuned coefficient sets to handle all possible dynamic conditions.

12.1.4.1 Considerations

Noise, resonant mode, and sample time are considered in tuning the close-loop controllers. SpinTAC simplifies the tuning process with one tuning parameter. By adjusting the bandwidth, it is straightforward to find desired performance while keeping the stability margins. As the bandwidth is increased, there is less deviation from the setpoint with the same amount of disturbance load applied. However as the bandwidth increases, this increases the noise that the controller is using to determine the output. Finding the correct tuning is often a balance between disturbance rejection and resultant noise that is generated.

The sample time is taken into account automatically by the SpinTAC controller. It uses the sample time of your system as a limit on the available bandwidth. This limit prevents your system from going unstable by preventing it from having a very high bandwidth and a very small sample time which can cause some instability in your system.

12.1.4.2 Tuning the InstaSPIN-MOTION Controller

SpinTAC controllers are tuned via a single parameter. This parameter is called bandwidth. The bandwidth of the SpinTAC controller is adjusted via the configuration parameter Bandwidth Scale. This value should be used to adjust the value of the bandwidth in the controller to meet the control requirements of your application. For position control applications, a single bandwidth is used to set the gain for both position and speed.

Figure 12-6 shows the SpinTAC Velocity Control response to a torque disturbance as the bandwidth is increased. As the bandwidth is increased, the response to the torque disturbance becomes faster and has less overshoot. When the bandwidth is increased too much it begins to oscillate around the goal speed when the torque disturbance is removed from the system. This indicates that the bandwidth has been set too high. In this example the ideal bandwidth is 40 rad/s. This is due to the response having minimal oscillation around the goal speed when the torque is removed.

Copyright © 2013–2019, Texas Instruments Incorporated

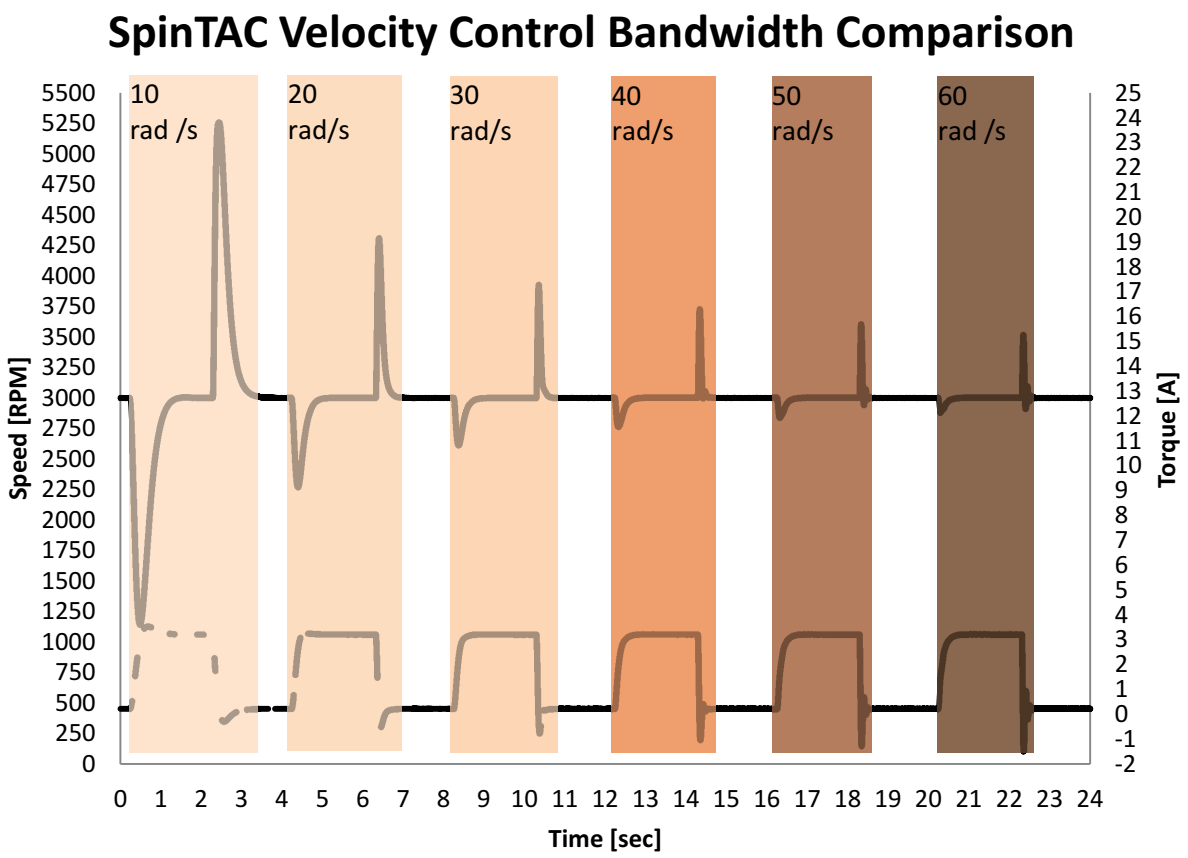


Figure 12-6. Comparison of Bandwidths for SpinTAC Velocity Control

Figure 12-7 shows SpinTAC Position Control response to a torque disturbance. As the bandwidth is increased, the response to the torque disturbance becomes faster and has less overshoot. When the bandwidth has been increased too much the motor begins to hum and oscillate. This indicates that the output of SpinTAC Position Control has begun to oscillate. In this example the ideal bandwidth is 50 rad/s. This is because it has a very good response to the disturbances while having a minimal output oscillation.

Copyright © 2013–2019, Texas Instruments Incorporated

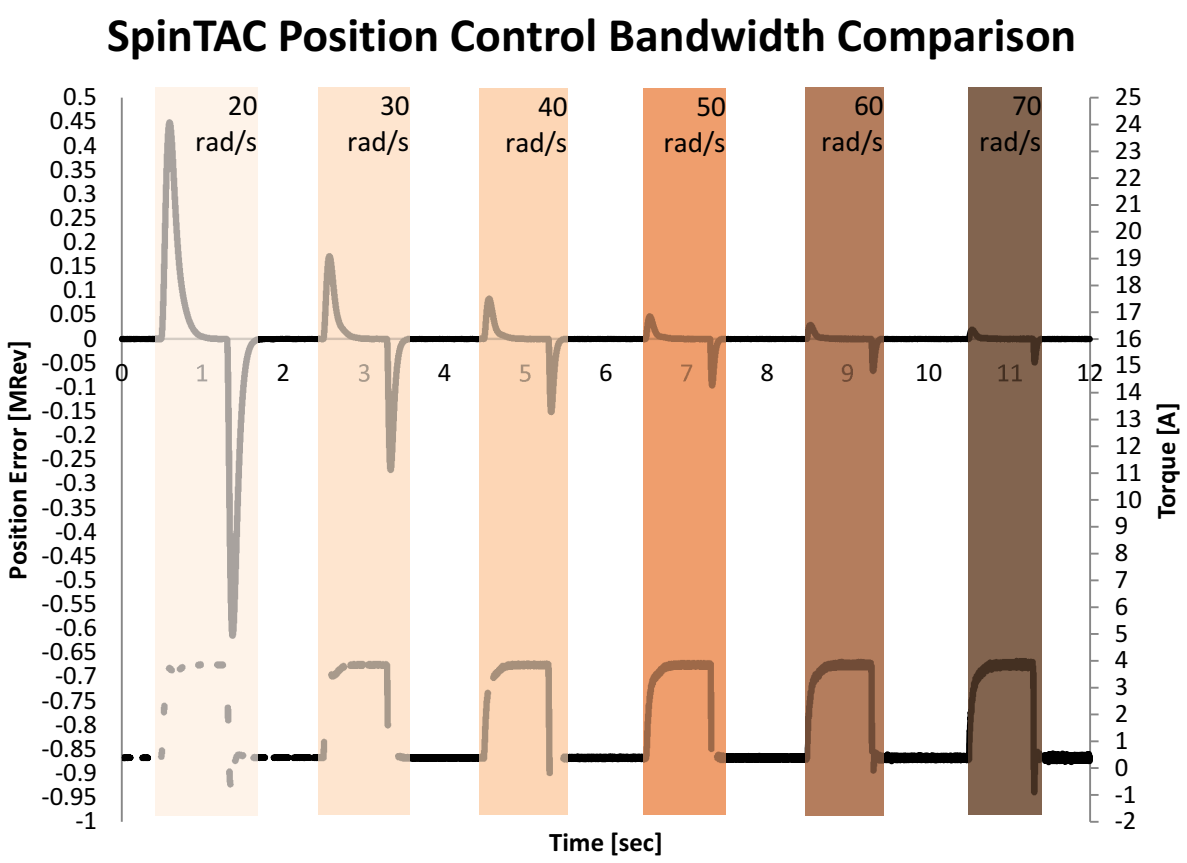


Figure 12-7. Comparison of Bandwidths for SpinTAC Position Control

To tune a SpinTAC controller, the first step is to set the speed reference to zero. 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" or "gMotorVars.SpinTAC.PosCtlBwScale" in steps of 1, continuing to feel how tightly the motor is holding zero speed. For motors where the shaft is not accessible, give the motor a reference step. Change the reference and monitor how aggressively the controller tries to achieve the new setpoint. Once the SpinTAC controller is tightly holding zero the bandwidth scale has been tuned for zero hold. At this point it is important to run the motor in order to ensure that this bandwidth will work across your appliction's operating range

12.2 Software Configuration for the SpinTAC Velocity Control

Configuring SpinTAC Velocity Control requires four steps. Lab 5d — InstaSPIN-MOTION Speed Controller — is an example project that implements the steps required to use SpinTAC Velocity Control. The header file spintac_velocity.h, included in MotorWare, allows you to quickly include the SpinTAC components in your project.

12.2.1 Include the Header File

This should be done with the rest of the module header file includes. In the Lab 5d example project, this file is included in the spintac_velocity.h header file. For your project, this step can be completed by including spintac_velocity.h

```
#include "sw/modules/spintac/src/32b/spintac_vel_ctl.h"
```


Copyright © 2013–2019, Texas Instruments Incorporated

12.2.2 Declare the Global Structure

This should be done with the global variable declarations in the main source file. In the Lab 5d project, this structure is included in the ST_Obj structure that is declared as part of the spintac_velocity.h header file.

```
ST_Obj st_obj;      // The SpinTAC Object
ST_Handle stHandle; // The SpinTAC Handle
```

This example is if you do not wish to use the ST_Obj structure that is declared in the spintac_velocity.h header file.

```
ST_VelCtl_t      stVelCtl;       // The SpinTAC Speed Controller object
ST_VELCTL_Handle stVelCtlHandle; // The SpinTAC Speed Controller Handle
```

12.2.3 Initialize the Configuration Variables

This should be done in the main function of the project ahead of the forever loop. This will load all of the default values into the SpinTAC Velocity Control. This step can be completed by running the functions ST_init and ST_setupVelCtl that are declared in the spintac_velocity.h header file. If you do not wish to use these two functions, the code example below can be used to configure SpinTAC Velocity Control component. This configuration of SpinTAC Velocity Control represents the typical configuration that should work for most motors.

```
// Initialize the SpinTAC Speed Controller Component
stVelCtlHandle = STVELCTL_init(&stVelCtl, sizeof(stVelCtl));

// Setup the maximum current in PU
_iq maxCurrent_PU = _IQ(USER_MOTOR_MAX_CURRENT / USER_IQ_FULL_SCALE_CURRENT_A);

// Instance of the velocity controller
STVELCTL_setAxis(stVelCtlHandle, ST_AXIS0);
// Sample time [s], (0, 1)
STVELCTL_setSampleTime_sec(stVelCtlHandle, _IQ(ST_SPEED_SAMPLE_TIME));
// System inertia upper (0, 127.9999] and lower (0, SgiMax] limits [PU/(pu/s^2)]
STVELCTL_setInertiaMaximums(stVelCtlHandle, _IQ(10.0), _IQ(0.001));
// System control signal high (0, OutMax] & low [OutMin, 0) limits [PU]
STVELCTL_setOutputMaximums(stVelCtlHandle, maxCurrent_PU, -maxCurrent_PU);
// System maximum (0, 1.0] and minimum [-1.0, 0) velocity [pu/s]
STVELCTL_setVelocityMaximums(stVelCtlHandle, _IQ(1.0), _IQ(-1.0));
// System upper (0, 0.2/(T*20)] and lower [0, BwScaleMax] limits for bandwidth scale
STVELCTL_setBandwidthScaleMaximums(stVelCtlHandle,
                                   _IQ24((0.2) / (ST_SPEED_SAMPLE_TIME * 20.0)), _IQ24(0.01));
// System inertia [PU/(pu/s^2)], [SgiMin, SgiMax]
STVELCTL_setInertia(stVelCtlHandle, _IQ(USER_SYSTEM_INERTIA));
// Controller bandwidth scale [BwMin, BwMax]
STVELCTL_setBandwidthScale(stVelCtlHandle, _IQ24(USER_SYSTEM_BANDWIDTH_SCALE));
// Initially ST_VelCtl is not enabled
STVELCTL_setEnable(stVelCtlHandle, false);
// Initially ST_VelCtl is not in reset
STVELCTL_setReset(stVelCtlHandle, false);
```


Copyright © 2013–2019, Texas Instruments Incorporated

12.2.4 Call SpinTAC Velocity Control

This should be done in the main ISR. This function needs to be called at the proper decimation rate for this component. The decimation rate is established by ISR_TICKS_PER_SPINTAC_TICK; for more information, see Section 4.7.1.4. Before calling the SpinTAC Velocity Control function the speed reference, acceleration reference and speed feedback must be updated. This example uses SpinTAC Velocity Move to provide the references to SpinTAC Velocity Control. For more information on SpinTAC Velocity Move, see Chapter 13.

```
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;    // Get pointer to CTRL object
// Get the mechanical speed in pu/s
_iq speedFeedback = EST_getFm_pu(obj->estHandle);    // Get the mechanical speed in pu/s
// Update the Velocity Reference
STVELCTL_setVelocityReference(stVelCtlHandle,
                                  STVELMOVE_getVelocityReference(stVelMoveHandle));
//Update the Acceleration Reference
STVELCTL_setAccelerationReference(stVelCtlHandle,
                                 STVELMOVE_getAccelerationReference(stVelMoveHandle));
//Update the Velocity Feedback
STVELCTL_setVelocityFeedback(stVelCtlHandle, speedFeedback);
// Run the SpinTAC Speed Controller
STVELCTL_run(stVelCtlHandle);

// Get the Torque Reference from the SpinTAC Speed Controller
iqReference = STVELCTL_getTorqueReference(stVelCtlHandle);

// Set the Iq reference that came out of SpinTAC Velocity Control
CTRL_setIq_ref_pu(ctrlHandle, iqReference);
```

12.2.5 Troubleshooting SpinTAC Velocity Control

12.2.5.1 ERR_ID

ERR_ID provides an error code for users. A list of errors defined for SpinTAC Velocity Control and the solutions for these errors are shown in Table 12-2.

Table 12-2. SpinTAC Velocity Control ERR_ID Code

ERR_ID	Problem	Solution
1	Invalid sample time value	Set cfg.T_sec within (0, 1]
2	Invalid reference maximum value	Set cfg.VelMax within (0, 1]
3	Invalid reference minimum value	Set cfg.VelMin within (0, 1]
4	Invalid control signal maximum value	Set cfg.OutMax within (0, 1]
5	Invalid control signal minimum value	Set cfg.OutMin within [-1, 0)
16	Invalid inertia maximum value	Set cfg.InertiaMax as a positive _iq24 value
17	Invalid inertia minimum value	Set cfg.InertiaMin within (0, cfg.InertiaMax]
18	Invalid bandwidth maximum value	Set cfg.BwMax within [0, min(2000, 0.2/cfg.T)]
19	Invalid bandwidth minimum value	Set cfg.BwMin within [0, cfg.BwMax]
32	Invalid axis ID	Set cfg.Axis within {ST_AXIS0, ST_AXIS1}
1012	Invalid inertia value	No action. Inertia will be saturated by the bound [cfg.InertiaMin, cfg.InertiaMax]
1014	Bandwidth × Inertia is greater than 2000	No action. The actual bandwidth is saturated by the value of 2000/Inertia
1016	Friction is out of bounds	No action. The friction will be saturated by the adjusted friction bounds [0, 5]
4001	Invalid SpinTAC license	Use a chip with a valid license
4003	Invalid ROM version	Use a chip with a valid ROM version or use the SpinTAC library that is compatible with the current ROM version.

Copyright © 2013–2019, Texas Instruments Incorporated

12.3 Optimal Performance in Speed Control

12.3.1 Introduction

Getting the best possible performance out of your motion system is important. A poorly tuned regulator can result in wasted energy, wasted material, or an unstable system. It is important that for any speed controller the performance is evaluated at many different speed and load operating points in order to determine how well it works in your application.

12.3.2 Comparing Speed Controllers

Speed controllers can be compared on a number of different factors. However, two metrics - disturbance rejection and profile tracking - can be used to test performance and determine how well your controller is tuned for your application.

12.3.3 Disturbance Rejection

Disturbance rejection tests a controller's resistance to external disturbances, which will impact the motor speed. Disturbance rejection is measured using the maximum speed error and settling time. The maximum speed error shows the deviation from the goal speed, and is an indication of how aggressively your controller is tuned. Aggressive tuning will produce a low maximum error.

Settling time refers to the amount of time from the point when the disturbance happens until the speed returns to a fixed band around the goal speed. This is also an indication of how aggressively your control loop is tuned. If the controller is tuned too aggressively it will have a long settling time because it will oscillate around the goal speed before settling.

Figure 12-8 and Figure 12-9 show the difference between poor tuning and optimal tuning of the same controller. As you can see by tuning the speed controller, when torque is applied or removed from a motor system the tuned controller greatly reduces the maximum error and settling time. This is an exaggerated example, but it is used to highlight the importance of getting a good tuning for your system.

Copyright © 2013–2019, Texas Instruments Incorporated

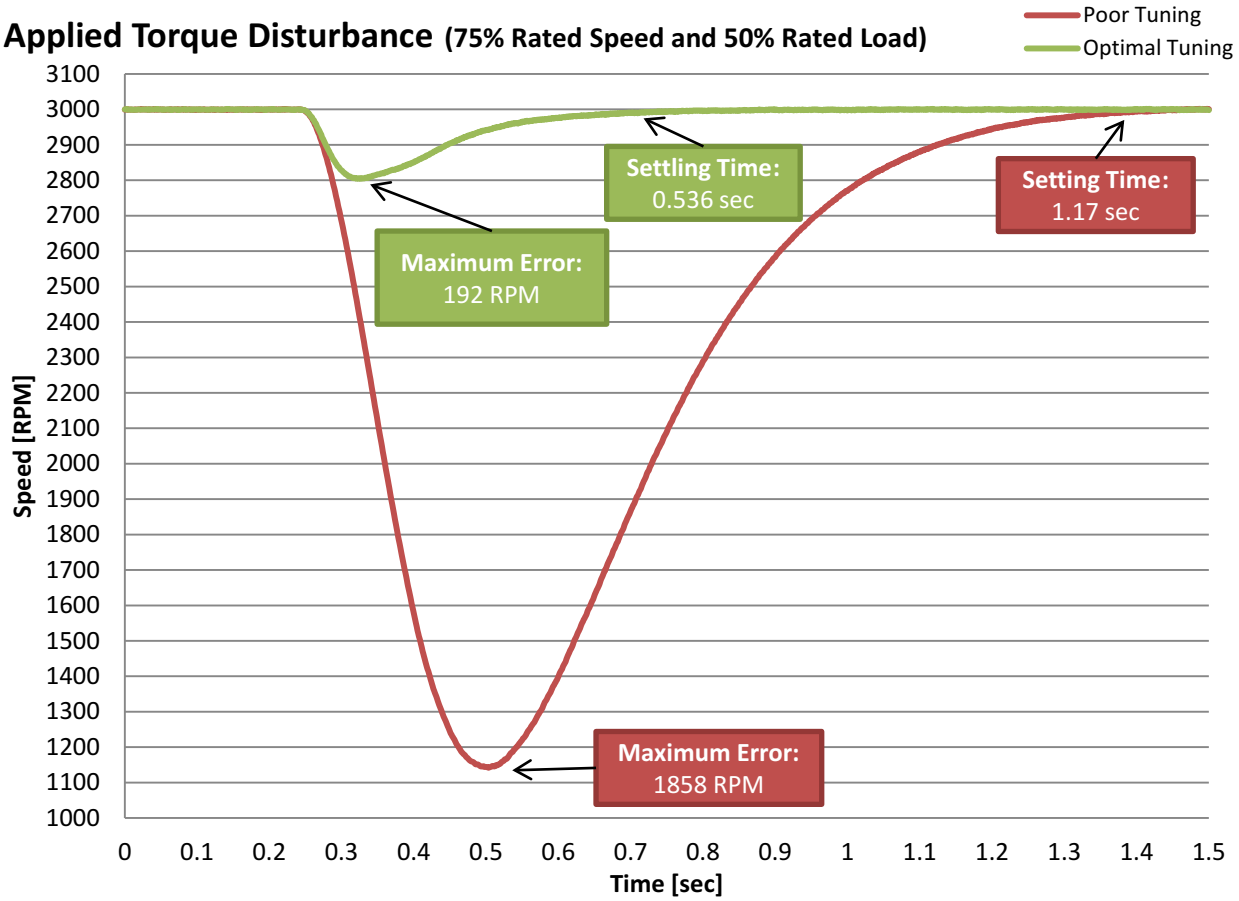


Figure 12-8. Velocity Tuning Comparison for Applied Torque Disturbance

Copyright © 2013–2019, Texas Instruments Incorporated

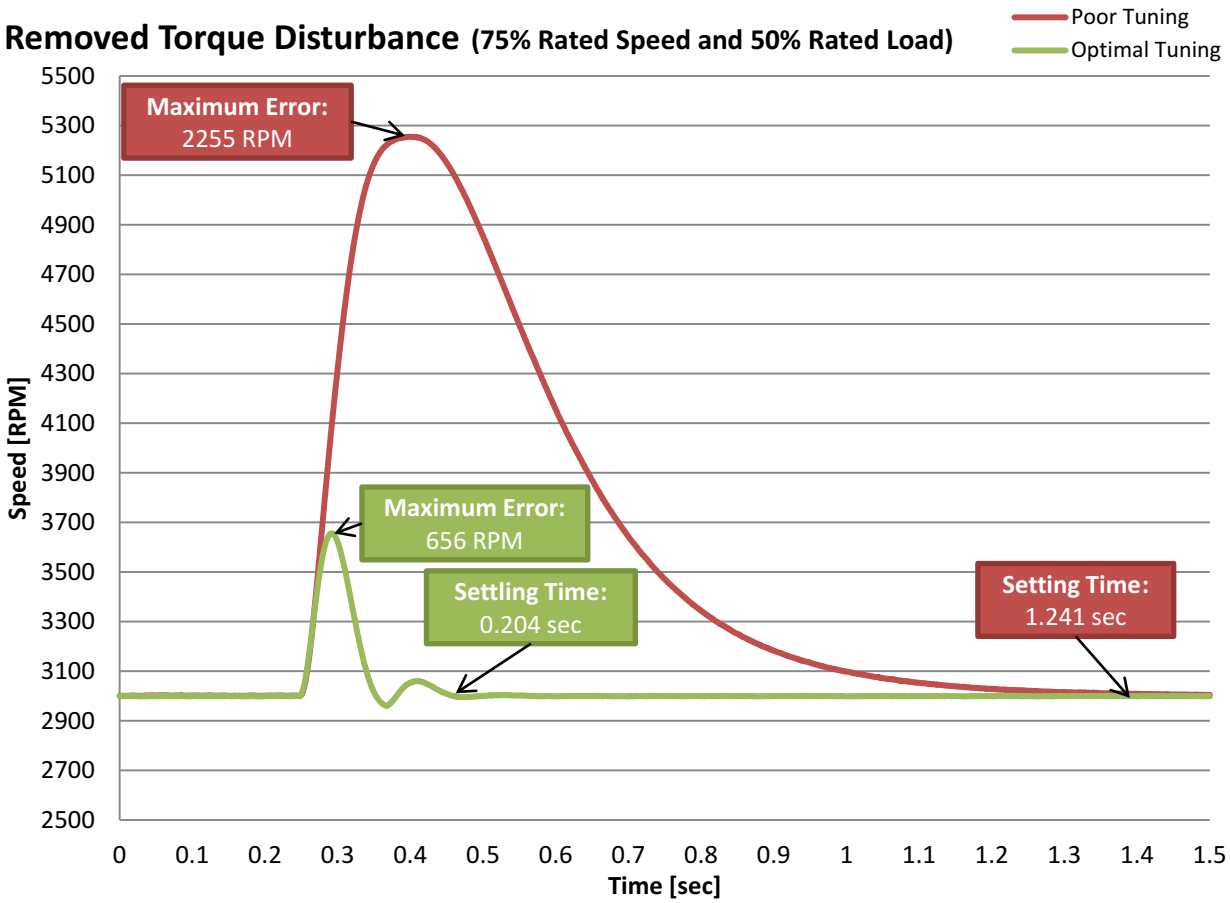


Figure 12-9. Velocity Tuning Comparison for Removed Torque Disturbance

When doing disturbance rejection testing it is important to test at multiple speed and load combinations. Speed controllers have different performance characteristics when placed into different situations. In order to properly evaluate the effectiveness of your speed controller, tests should be done across the entire application range. The test results will indicate whether the controller will meet the application specifications, or whether the controller needs to be tuned multiple times for different operating points.

It is also important to be able to create repeatable disturbances. This can be accomplished using a dynamometer or a disturbance motor. Creating repeatable disturbance is an important factor when evaluating multiple controllers. If test conditions cannot be replicated, it is difficult to adequately compare the responses of two controllers.

12.3.4 Profile Tracking

Profile tracking tests how well the controller follows a changing target speed. The two metrics to evaluate in this testing are the maximum error and the absolute average error. The maximum speed error shows how much the controller overshoots while changing speeds. This is an indication of how aggressively your controller is tuned. If you controller is not tuned aggressively enough, the speed will overshoot the target, and will take a long time to recover. If the controller is tuned too aggressively it will overshoot, and then oscillate as it settles on the goal speed. If the controller is correctly tuned, it will overshoot and then smoothly return to the goal speed.

Absolute average error is an average of the absolute value of the instantaneous speed error over the entire profile. This measure shows the amount of deviation throughout the entire profile. It takes into account all of the little errors as the motor is running. If the controller is tuned too aggressively it will result in larger absolute average error because the controller will be oscillating throughout the profile. If the controller is not tuned aggressively enough, it will result in a larger absolute average error because it is continuously falling behind what the profile is commanding the motor to do.

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 12-10 shows the difference between the default tuning and the optimal tuning of the same controller. As you can see by tuning the speed controller, you are able to make your motion system much more closely track the reference. By tuning the controller, it greatly reduces the maximum error, the absolute average error, and the maximum overshoot.

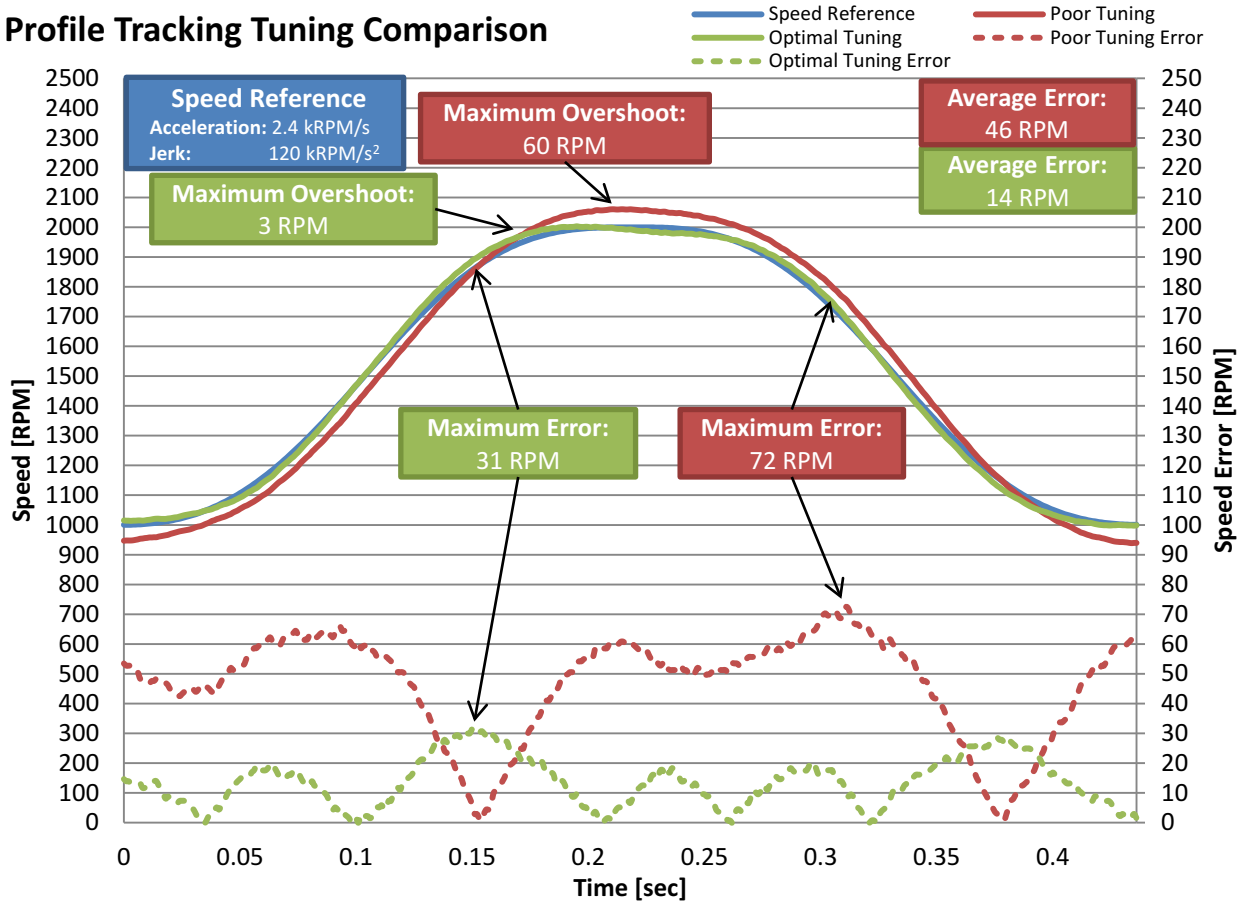


Figure 12-10. Velocity Tuning Comparison for Profile Tracking

It is important to test multiple speeds and accelerations in your profile as well as multiple different loads. Speed controllers have different performance characteristics when placed into different situations. In order to properly evaluate the effectiveness of your speed controller, tests should be conducted across the entire application range. This includes when you design the profile for testing. Care needs to be taken to ensure that the application speeds and accelerations are built into the profile. The results of these tests will inform you if your controller will meet the application specifications or if your controller needs to be tuned multiple times for different operating points.

It is also important to be able to create repeatable profiles and loads. Creating a repeatable profile can be done using SpinTAC Velocity Move and SpinTAC Velocity Plan; for more information, see Chapter 13. Repeatable profiles are required so that all controllers will be tested using the same reference in the same order and for the same length of time. This ensures that test conditions are as identical as possible. When applying load during a profile tracking test it is important to create repeatable disturbances. This can be accomplished using a dynamometer or a disturbance motor. Creating a repeatable disturbance is an important factor when evaluating controllers. If test conditions cannot be replicated, it is difficult to adequately compare the responses of two controllers.

Copyright © 2013–2019, Texas Instruments Incorporated

12.3.5 InstaSPIN-MOTION Velocity Control Advantage

12.3.5.1 Single Parameter Tuning

InstaSPIN-MOTION presents numerous advantages in achieving optimal performance for your application. The SpinTAC Velocity Control helps you achieve optimal performance by offering single parameter tuning over a wide operating range. Having a single tuning parameter allows you to quickly zero in on the right tuning settings for your application. The Active Disturbance Rejection Control (ADRC) at the core of the SpinTAC Velocity Control allows that single tuning parameter to work across a very wide operating range. In most cases a single tuning setting can work across the entire operating range of an application. Compare that with a PI controller that requires at least two tuning parameters and both of those tuning parameters need to be tuned for multiple different operating points in an application. This results in a much longer amount of time spent tuning your application than with the SpinTAC Velocity Control.

To compare the differences between the SpinTAC Velocity Control and a traditional PI speed controller, we attached an Anaheim Automation BLY172S motor that comes with the DRV8312 Rev D evaluation kit to a Magtrol HD-400 dynamometer. The PI tuning parameters were arrived at using the example in Section 11.5. The following characteristics were used to calculate the Speed PI gains:

- Rs = 0.4 ohms
- Ls = 0.6mH
- Back-EMF = 0.0054 V-sec / radians
- Inertia = 335E-4 kg-m2 (To include the Inertia of the motor and the dynamometer)
- Rotor poles = 8
- Speed bandwidth = 800 rad / sec
- Damping factor = 4

This resulted in the following Speed PI gains:

- spdKpseries = 5.495
- spdKiseries = 132.88

The SpinTAC Velocity Control was tuned experimentally by the method outlined in Section 12.1.4. This tuning resulted in the following gain:

- Bandwidth = 45 radians / sec

The above gains for the PI speed controller and the SpinTAC Velocity Control were using for the following tests.

12.3.5.2 Disturbance Rejection Test

The ADRC technology in the SpinTAC Velocity Control has excellent disturbance rejection. It is actively estimating disturbances in real-time and compensating for these disturbances (see Figure 12-11 and Figure 12-12). When the SpinTAC Velocity Control detects a disturbance in the system, it applies a correction to quickly and smoothly bring the speed of the motor back in line with the target speed.

Copyright © 2013–2019, Texas Instruments Incorporated

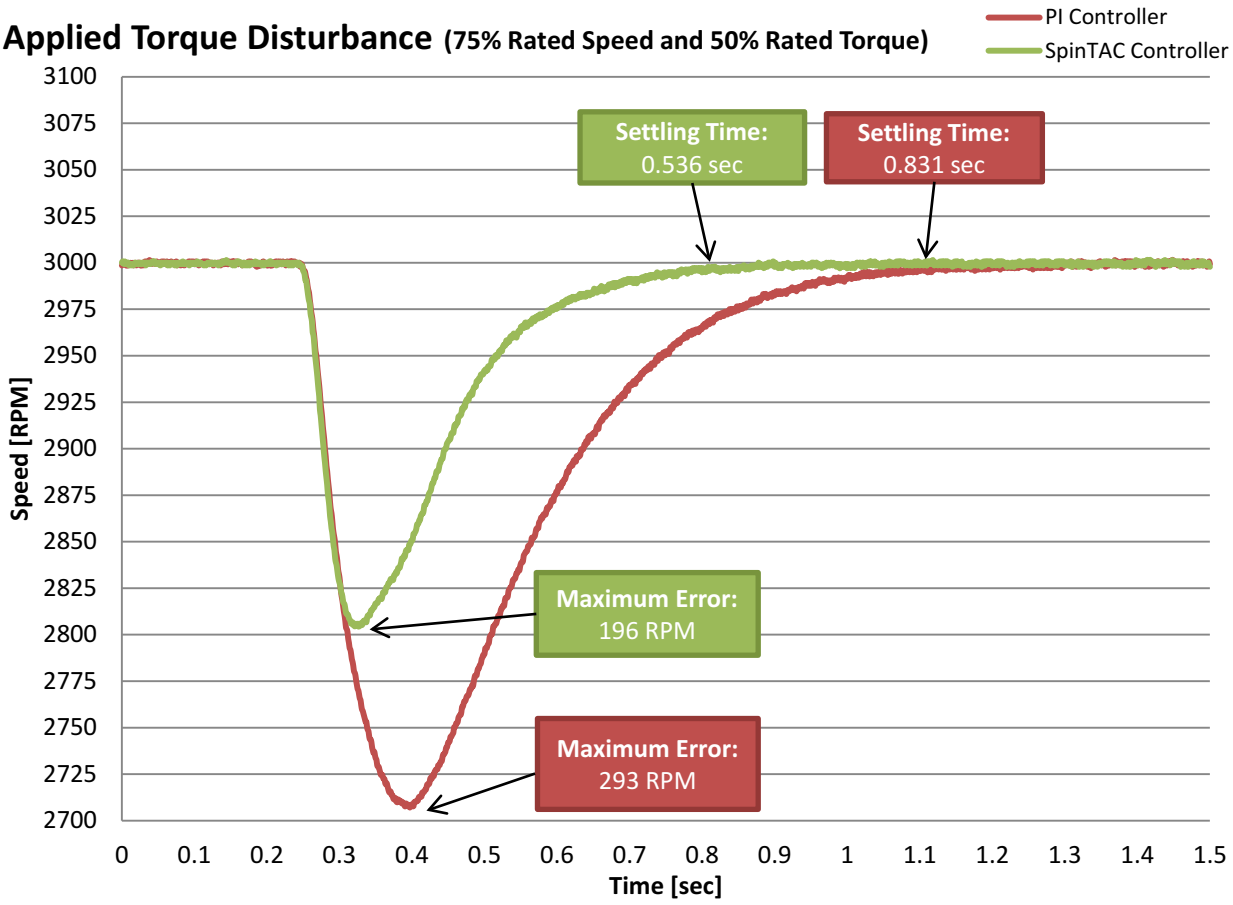


Figure 12-11. PI and SpinTAC Comparison for Applied Torque Disturbance Velocity Control

Copyright © 2013–2019, Texas Instruments Incorporated

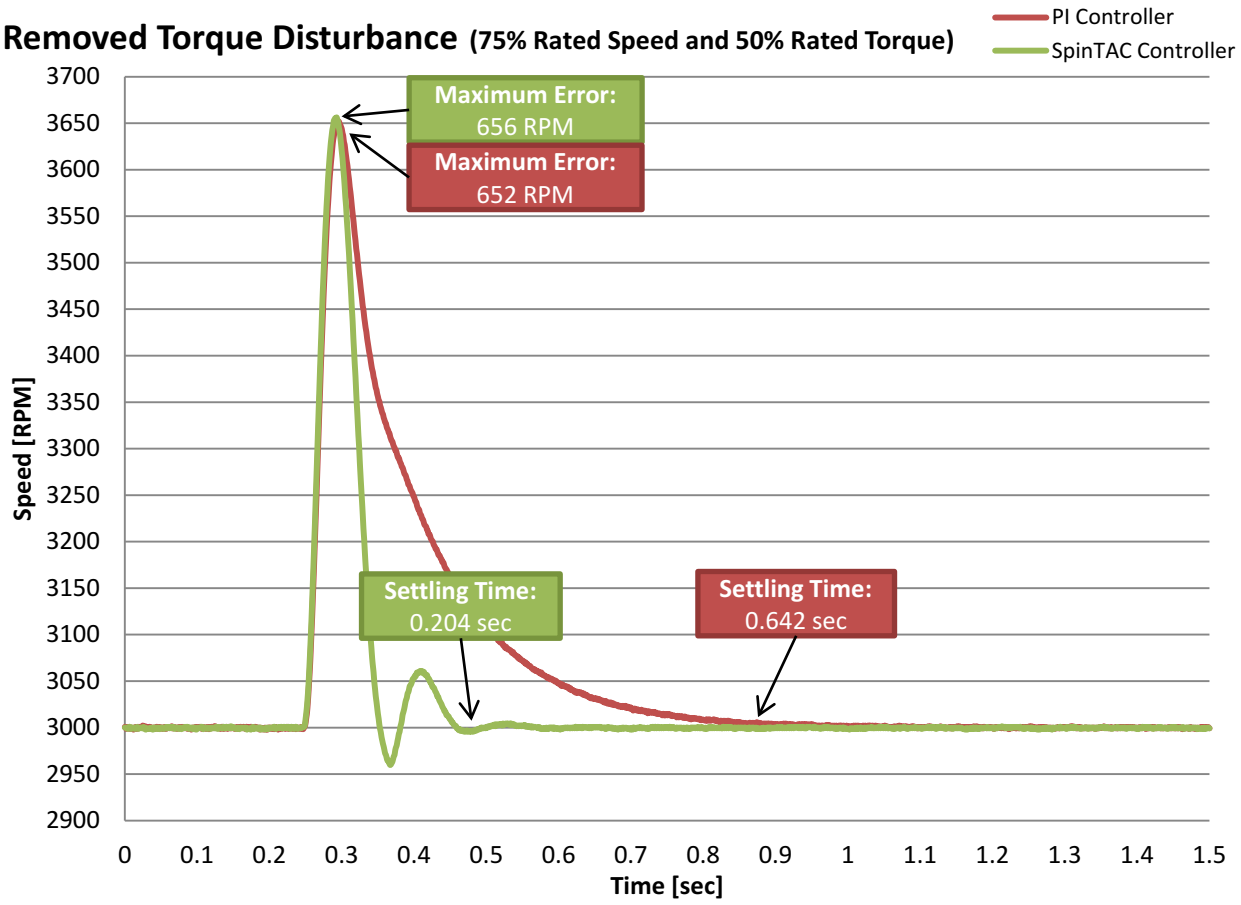


Figure 12-12. PI and SpinTAC Comparison for Removed Torque Disturbance Velocity Control

The SpinTAC Velocity Control features a much faster recovery time and smaller maximum error than a traditional PI controller. This results in your application experiencing fewer speed fluctuations. This reduction in speed fluctuation results in your application running at a more consistent speed and can lead to less power consumed in your application.

12.3.5.3 Feedforward

The SpinTAC Velocity Control also features feedforward. This allows for excellent profile tracking (see Figure 12-13). Feedforward tells the SpinTAC Velocity Control how fast it should be accelerating or decelerating. This allows the SpinTAC Velocity Control to react to profile changes much quicker than a PI controller. It results in less maximum error and less absolute average error.

Copyright © 2013–2019, Texas Instruments Incorporated

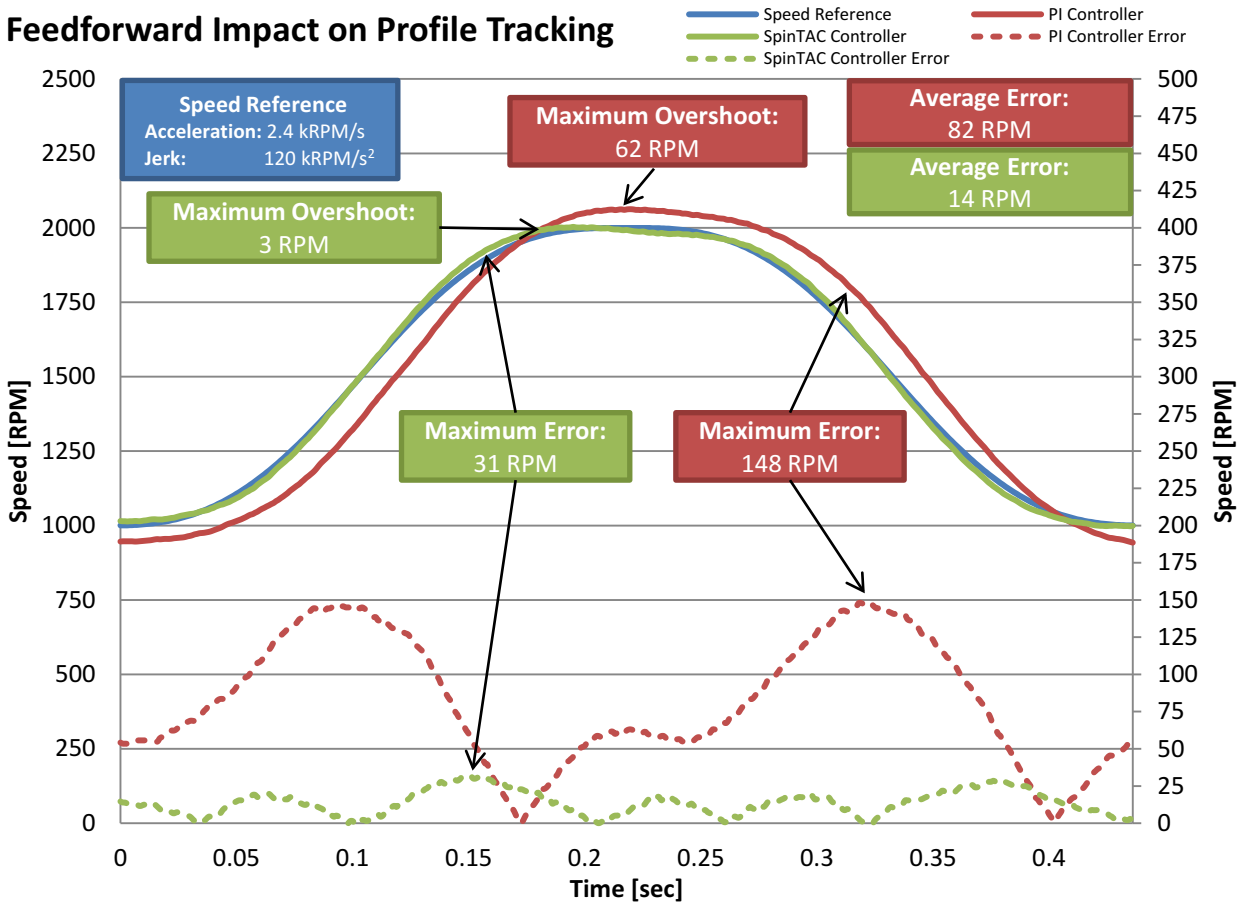


Figure 12-13. PI and SpinTAC Comparison for Feedforward Impact on Velocity Profile Tracking

The SpinTAC Velocity Control features reduced maximum error and absolute average error. This results in much improved tracking performance over a traditional PI controller. This results in less wasted motion and wasted energy while the controller is attempting and overacting to try and track the changing reference. This feature becomes even more important when combined with the disturbance rejection capabilities discussed in Section 12.3.5.2. If your system encounters a disturbance while tracking a changing speed reference, that could result in a large amount of overshoot and wasted energy or material.

12.3.5.4 No Integrator Windup

Integrator windup is an issue where the integrator component of a standard PI controller has built up a large reserve of error. This happens when the controller goes into saturation and there is a steady state error in the speed. This steady state error will continue to build up the value in the integrator and when the condition causing the saturation is removed, this error will cause the speed to drastically overshoot the speed reference. The SpinTAC Velocity Control does not have this issue. The ADRC technology is estimating the system error in real-time and does not rely on an integrator that can cause integrator windup issues.

Figure 12-14 shows a case where a traditional PI controller experiences integrator windup. In this case the motor could not overcome the torque disturbance and it was forced to run at the speed slower than the setpoint. This placed the controller into saturation where the PI controller's integrator term built up over time. Once this torque disturbance was removed, the integrator term of the PI caused it to have a very large overshoot and take a much longer time to settle back to the speed setpoint. The SpinTAC controller did not see any of these ill effects since it does not contain an integrator term that can build up and cause integrator windup issues. It is also interesting to compare Figure 12-14 with Figure 12-12. You should notice that SpinTAC's response to removing the torque disturbance is very similar for the 50% rated torque disturbance and the 80% rated torque disturbance.

Copyright © 2013–2019, Texas Instruments Incorporated

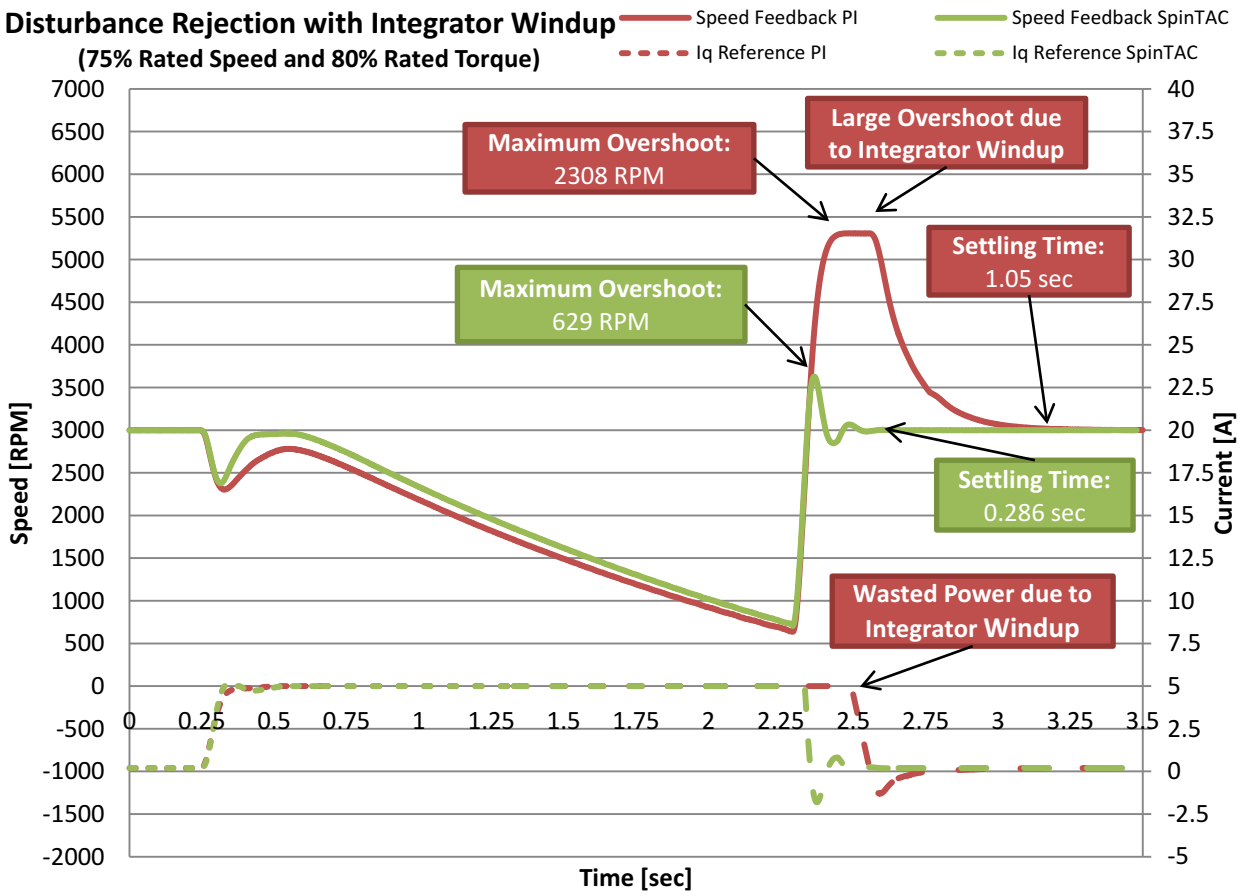


Figure 12-14. PI and SpinTAC Comparison for Integrator Windup During Disturbance Rejection Velocity Control

12.3.5.5 Minimum Startup Overshoot

InstaSPIN-MOTION features a controller that produces minimum overshoot during startup. This results in your application spending less energy when starting the motor and less time to recover and run at the target speed. For compressors that cycle on and off this can be a critical point to save energy.

Figure 12-15 compares the step response of the SpinTAC Velocity Control with a traditional PI speed controller. From the plot you can see that the traditional PI speed controller has a much greater overshoot and settling time than the SpinTAC Velocity Control. The large overshoot by the PI speed controller results in wasted power and wasted motion. It is not the ideal response for your system. The SpinTAC Velocity Control has a much better response and results in less overshoot and a faster settling time.

Copyright © 2013–2019, Texas Instruments Incorporated

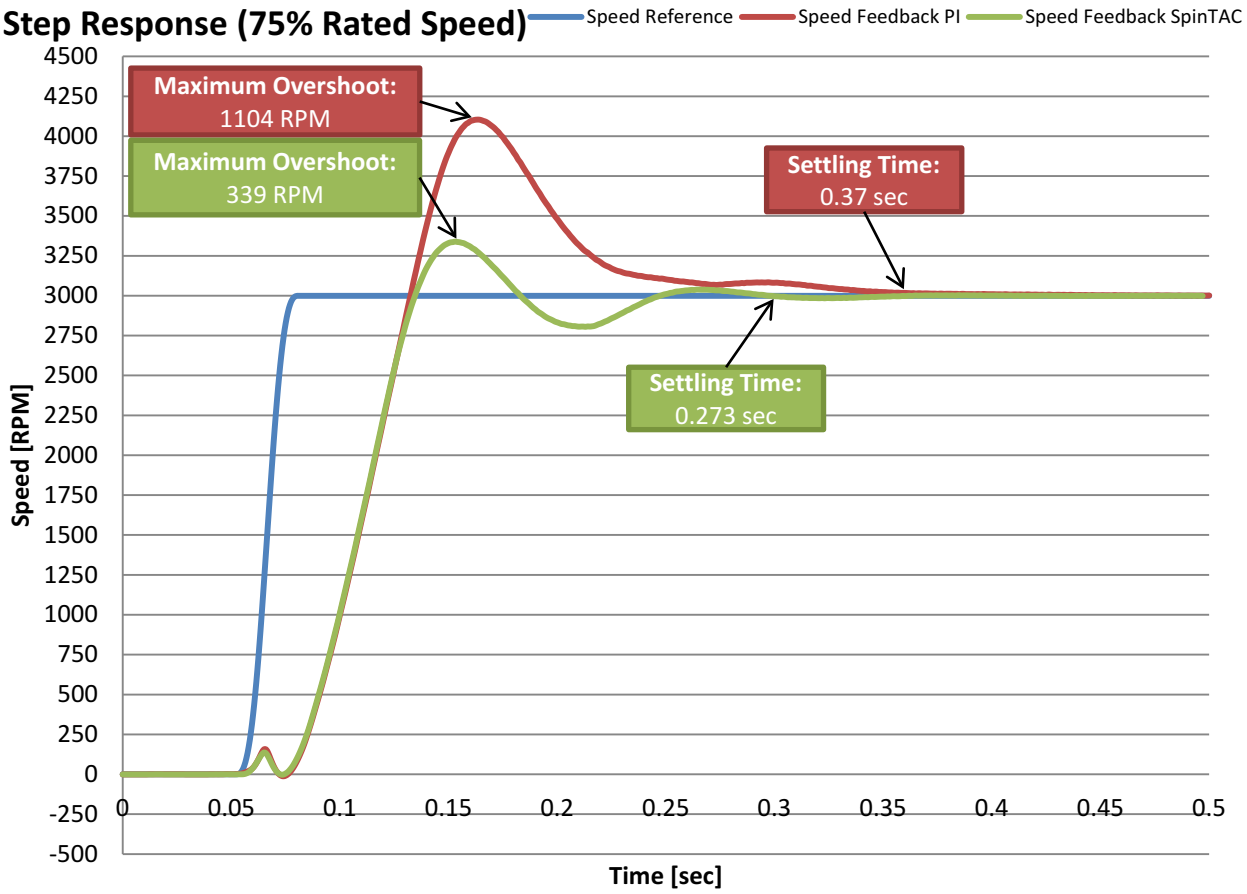


Figure 12-15. PI and SpinTAC Comparison for Step Response Velocity Control

12.3.5.6 Conclusions

The SpinTAC Velocity Control included in InstaSPIN-MOTION directly replaces the traditional PI speed controller. It results in better performance across the entire range of your application. It is less complex to tune than a PI controller, since it features a single tuning parameter. This results in less development effort being spent tuning the speed controller, allowing you to focus on the rest of your application. It results in less overshoot and faster settling time. It also features better profile tracking. All of these features combine to reduce the energy consumed by your application due to wasted motion.

12.4 Software Configuration for SpinTAC Position Control

Configuring SpinTAC Position Control requires four steps. Lab 13a — Tuning the InstaSPIN-MOTION Position Controller — is an example project that implements the steps required to use SpinTAC Position Control. The header file spintac.h, included in MotorWare, allows you to quickly include the SpinTAC components in your project.

12.4.1 Include the Header File

This should be done with the rest of the module header file includes. In the Lab 13a example project this file is included in the spintac_position.h header file. For your project, this step can be completed by including spintac_position.h.

```
#include "sw/modules/spintac/src/32b/spintac_pos_ctl.h"
```


Copyright © 2013–2019, Texas Instruments Incorporated

12.4.2 Declare the Global Structure

This should be done with the global variable declarations in the main source file. In the Lab 13a project this structure is included in the ST_Obj structure that is declared as part of the spintac_position.h header file.

```
ST_Obj       st_obj;    // The SpinTAC Object
ST_Handle stHandle;  // The SpinTAC Handle
```

This example is if you do not wish to use the ST_Obj structure that is declared in the spintac_position.h header file.

```
ST_PosCtl_t       stPosCtl;       // The SpinTAC Position Controller object
ST_POSCTL_Handle stPosCtlHandle; // The SpinTAC Position Controller Handle
```

12.4.3 Initialize the Configuration Variables

This should be done in the main function of the project ahead of the forever loop. This will load all of the default values into SpinTAC Position Control. This step can be completed by running the functions ST_init and ST_setupPosCtl that are declared in the spintac_position.h header file. If you do not wish to use these two functions, the code example below can be used to configure SpinTAC Position Control component. This configuration of SpinTAC Position Control represents the typical configuration that should work for most motors.

```
// Initialize SpinTAC Position Control
stPosCtlHandle = STPOSCTL_init(&stPosCtl, sizeof(stPosCtl));

// Setup the maximum current in PU
_iq maxCurrent_PU = _IQ(USER_MOTOR_MAX_CURRENT / USER_IQ_FULL_SCALE_CURRENT_A);

// Instance of the position controller
STPOSCTL_setAxis(stPosCtlHandle, ST_AXIS0);
// Sample time [s], (0, 1)
STPOSCTL_setSampleTime_sec(stPosCtlHandle, _IQ(ST_SAMPLE_TIME));
// System inertia upper (0, 127.9999] and lower (0, InertiaMax] limits [PU/(pu/s^2)]
STPOSCTL_setInertiaMaximums(stPosCtlHandle, _IQ(10.0), _IQ(0.001));
// System velocity limit (0, 1.0] [pu/s]
STPOSCTL_setVelocityMaximum(obj->;posCtlHandle, _IQ24(1.0));
// System control signal high (0, 1] & low [-1, 0) limits [PU]
STPOSCTL_setOutputMaximums(stPosCtlHandle, maxCurrent_PU, -maxCurrent_PU);
// System maximum (0, 1.0] and minimum [-1.0, 0) velocity [pu/s]
STPOSCTL_setVelocityMaximums(stPosCtlHandle, _IQ(1.0), _IQ(-1.0));
// System maximum value for mechanical revolutions [MRev]
STPOSCTL_setPositionRolloverMaximum_mrev(stPosCtlHandle, _IQ24(ST_MREV_ROLLOVER));
// Sets the values used for converting between pu and MRev
STPOSCTL_setUnitConversion(stPosCtlHandle, USER_IQ_FULL_SCALE_FREQ_Hz,
                                                    USER_MOTOR_NUM_POLE_PAIRS);
// System maximum allowable error [MRev]
STPOSCTL_setPositionErrorMaximum_mrev(stPosCtlHandle,
                                                _IQ24(ST_POS_ERROR_MAXIMUM_MREV));
// Disturbance type {true: Ramp; false: Square}
STPOSCTL_setRampDisturbanceFlag(stPosCtlHandle, false);
// System upper (0, 0.1/(cfg.T_sec*20)] and lower [0, BwScaleMax] limits for BWScale
STPOSCTL_setBandwidthScaleMaximums(stPosCtlHandle,
                                _IQ24((0.1) / (ST_SAMPLE_TIME * 20.0)), _IQ24(0.01));
// Enables the feedback filter
STPOSCTL_setFilterEnableFlag(stPosCtlHandle, true);
// System inertia [PU/(pu/s^2)], [InertiaMin, InertiaMax]
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
STPOSCTL_setInertia(stPosCtlHandle, _IQ(USER_MOTOR_INERTIA));
// Controller bandwidth scale [BwScaleMin, BwScaleMax]
STPOSCTL_setBandwidthScale(stPosCtlHandle, _IQ24(USER_SYSTEM_BANDWIDTH_SCALE));
// Initially ST_PosCtl is not enabled
STPOSCTL_setEnable(stPosCtlHandle, false);
// Initially ST_PosCtl is not in reset
STPOSCTL_setReset(stPosCtlHandle, false);
```

12.4.4 Call SpinTAC Position Control

This should be done in the main ISR. This function needs to be called at the proper decimation rate for this component. The decimation rate is established by ISR_TICKS_PER_SPINTAC_TICK; for more information, see Section 4.7.1.4. Before calling SpinTAC Position Control function the position reference, speed reference, acceleration reference, and position feedback must be updated. This example uses SpinTAC Position Move to provide the references to the SpinTAC Position Control. For more information on SpinTAC Position Move, see Chapter 13.

```
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;    // Get pointer to CTRL object
// Get the mechanical speed in pu/s
_iq speedFeedback = EST_getFm_pu(obj->estHandle);
STPOSCTL_setPositionReference_mrev(stPosCtlHandle,
                                   STPOSMOVE_getPositionReference_mrev(stPosMoveHandle));
// Update the Velocity Reference
STPOSCTL_setVelocityReference(stPosCtlHandle,
                                      STPOSMOVE_getVelocityReference(stPosMoveHandle));
// Update the Acceleration Reference
STPOSCTL_setAccelerationReference(stPosCtlHandle,
                                  STPOSMOVE_getAccelerationReference(stPosMoveHandle));
// Update the Position Feedback
STPOSCTL_setPositionFeedback_mrev(stObj->posCtlHandle,
                                        STPOSCONV_getPosition_mrev(stPosConvHandle));
// Run SpinTAC Position Control
STPOSCTL_run(stPosCtlHandle);

// Get the Torque Reference from SpinTAC Position Control
iqReference = STPOSCTL_getTorqueReference(stPosCtlHandle);

// Set the Iq reference that came out of SpinTAC Position Control
CTRL_setIq_ref_pu(ctrlHandle, iqReference);
```

12.4.5 Troubleshooting SpinTAC Position Control

12.4.5.1 ERR_ID

ERR_ID provides an error code for users. A list of errors defined for SpinTAC Position Control and the solutions for these errors are shown in Table 12-3.

Table 12-3. SpinTAC Position Control ERR_ID Code

ERR_ID	Problem	Solution
1	Invalid sample time value	Set cfg.T_sec within (0, 1]
2	Invalid velocity reference maximum value	Set cfg.VelMax within (0, 1]
4	Invalid control signal maximum value	Set cfg.OutMax within (0, 1]
5	Invalid control signal minimum value	Set cfg.OutMin within [-1, 0)
13	Invalid position rollover bound value	Set cfg.ROMax_mrev within [2, 100]
14	Invalid mechanical revolution [MRev] to [pu] ratio value	Set cfg.mrev_TO_pu within [0.002, 1]

Copyright © 2013–2019, Texas Instruments Incorporated

Table 12-3. SpinTAC Position Control ERR_ID Code (continued)

ERR_ID	Problem	Solution
15	Invalid position error maximum value	Set cfg.PosErrMax_mrev within [0, cfg.ROMax_mrev/2]
16	Invalid inertia maximum value	Set cfg.InertiaMax as a positive _iq24 value
17	Invalid inertia minimum value	Set cfg.InertiaMin within (0, cfg.InertiaMax]
18	Invalid bandwidth maximum value	Set cfg.BwMax within [0, min(2000, 0.2/cfg.T)]
19	Invalid bandwidth minimum value	Set cfg.BwMin within [0, cfg.BwMax]
20	Invalid value for disturbance specification	Set cfg.RampDist within {false, true}
32	Invalid axis ID	Set cfg.Axis within {ST_AXIS0, ST_AXIS1}
35	Invalid value for filter enable	Set cfg.FiltEN within {false, true}
1012	Invalid inertia value	No action. Inertia will be saturated by the bound [cfg.InertiaMin, cfg.InertiaMax]
1014	Bandwidth × Inertia is greater than 2000	No action. The actual bandwidth is saturated by the value of 2000/Inertia
1016	Friction is out of bounds	No action. The friction will be saturated by the adjusted friction bounds [0, 5]
2002	Position error exceeds maximum	Increase bandwidth to get less position error or make profile velocity, acceleration, and jerk slower
4001	Invalid SpinTAC license	Use a chip with a valid license
4003	Invalid ROM version	Use a chip with a valid ROM version or use the SpinTAC library that is compatible with the current ROM version.

12.5 Optimal Performance in Position Control

12.5.1 Introduction

Getting the best possible performance out of your motion system is important. A poorly tuned regulator can result in wasted energy, wasted material, or an unstable system. It is important that for any position controller the performance is evaluated at many different operating points in order to determine how well it works in your application.

12.5.2 Comparing Position Controllers

Position controllers can be compared on a number of different factors. However, two metrics - disturbance rejection and profile tracking - can be used to test performance and determine how well your controller is tuned for your application.

12.5.3 Disturbance Rejection

Disturbance rejection tests a controller's resistance to external disturbances, which will impact the motor speed and position. Disturbance rejection is measured using the maximum error and settling time. The maximum error shows the deviation from the goal position, and is an indication of how aggressively your controller is tuned. Aggressive tuning will produce a low maximum error.

Settling time refers to the amount of time from the point when the disturbance happens until the position returns to a fixed band around the goal position. This is also an indication of how aggressively your control loop is tuned. If the controller is tuned too aggressively it will have a long settling time because it will oscillate around the goal position before settling.

Figure 12-16 and Figure 12-17 show the difference between poor tuning and optimal tuning of the same controller. As you can see by tuning the position controller, when torque is applied or removed from a motor system the tuned controller greatly reduces the maximum error and settling time. This is an exaggerated example, but it is used to highlight the importance of getting a good tuning for your system.

Copyright © 2013–2019, Texas Instruments Incorporated

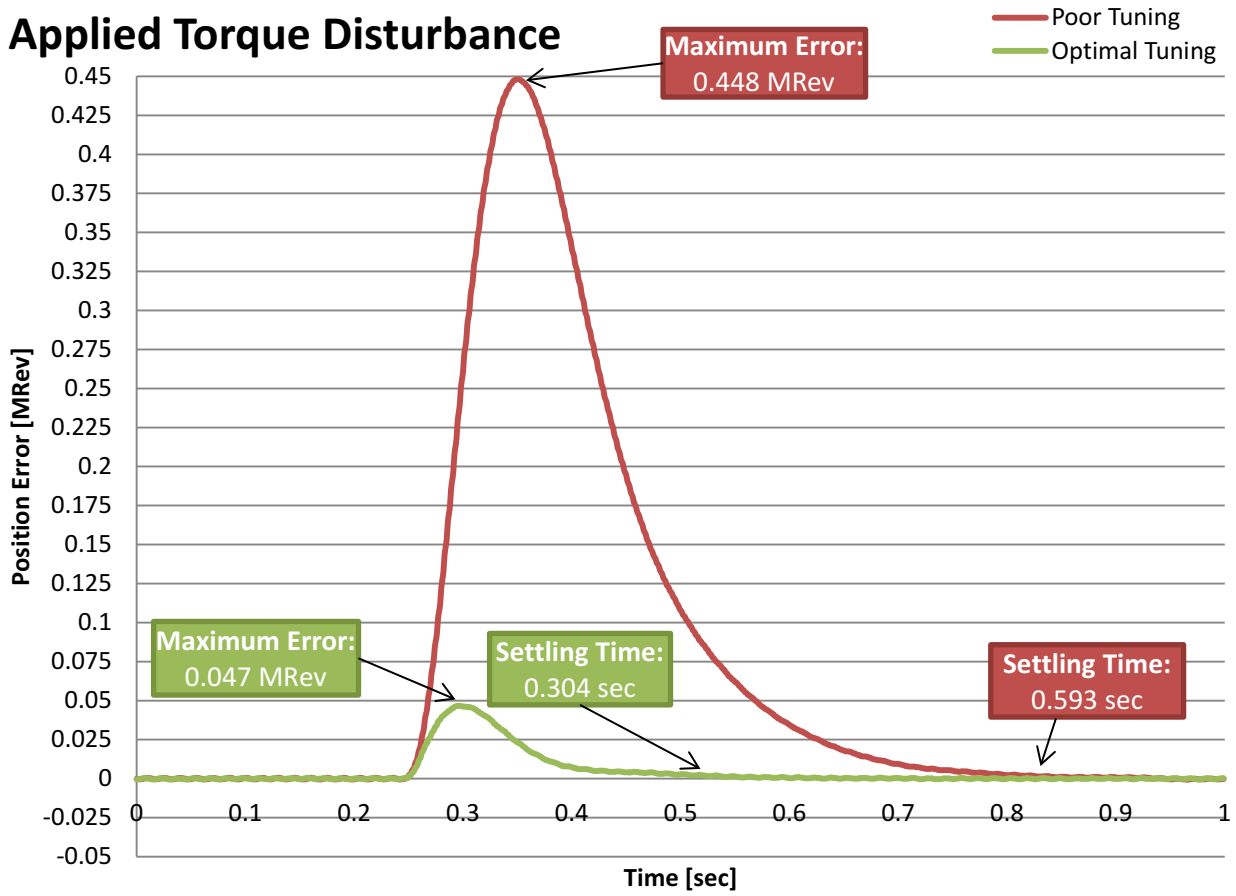


Figure 12-16. Position Tuning Comparison for Applied Torque Disturbance

Copyright © 2013–2019, Texas Instruments Incorporated

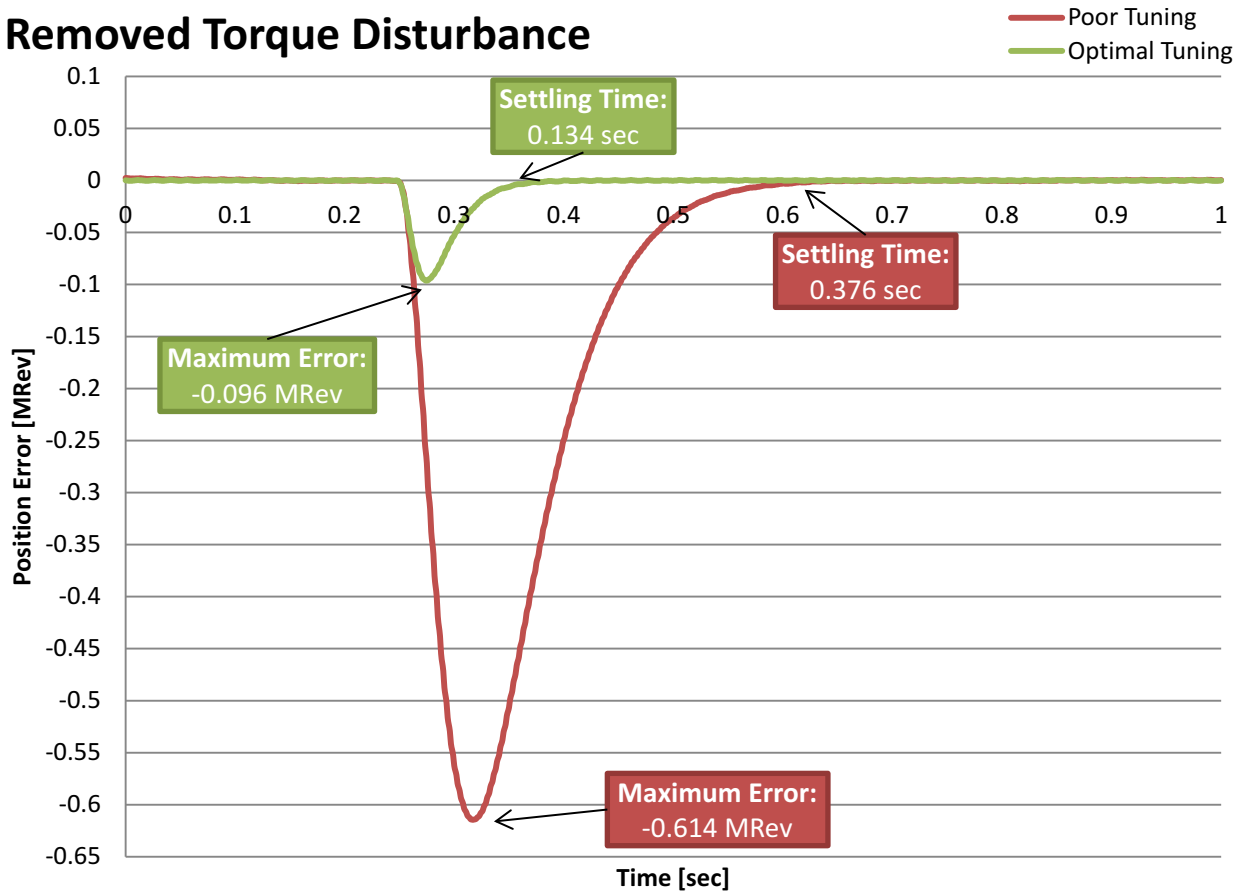


Figure 12-17. Position Tuning Comparison for Removed Torque Disturbance

When doing disturbance rejection testing it is important to test at multiple speed and load combinations. Position controllers have different performance characteristics when placed into different situations. In order to properly evaluate the effectiveness of your position controller, tests should be done across the entire application range. The test results will indicate whether the controller will meet the application specifications, or whether the controller needs to be tuned multiple times for different operating points.

It is also important to be able to create repeatable disturbances. This can be accomplished using a dynamometer or a disturbance motor. Creating repeatable disturbance is an important factor when evaluating multiple controllers. If test conditions cannot be replicated, it is difficult to adequately compare the responses of two controllers.

12.5.4 Profile Tracking

Profile tracking tests how well the controller follows a changing target position. The two metrics to evaluate in this testing are the maximum error and the absolute average error. The maximum error shows how much the controller overshoots while changing positions. This is an indication of how aggressively your controller is tuned. If you controller is not tuned aggressively enough, the position will overshoot the target, and will take a long time to recover. If the controller is tuned too aggressively it will overshoot, and then oscillate as it settles on the goal position. If the controller is correctly tuned, it will overshoot and then smoothly return to the goal position.

Absolute average error is an average of the absolute value of the instantaneous error over the entire profile. This measure shows the amount of deviation throughout the entire profile. It takes into account all of the little errors as the motor is running. If the controller is tuned too aggressively it will result in larger absolute average error because the controller will be oscillating throughout the profile. If the controller is not tuned aggressively enough, it will result in a larger absolute average error because it is continuously falling behind what the profile is commanding the motor to do.

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 12-18 shows the difference between the default tuning and the optimal tuning of the same controller. As you can see by tuning the position controller, you are able to make your motion system much more closely track the reference. By tuning the controller, it greatly reduces the maximum error, the absolute average error, and the maximum overshoot.

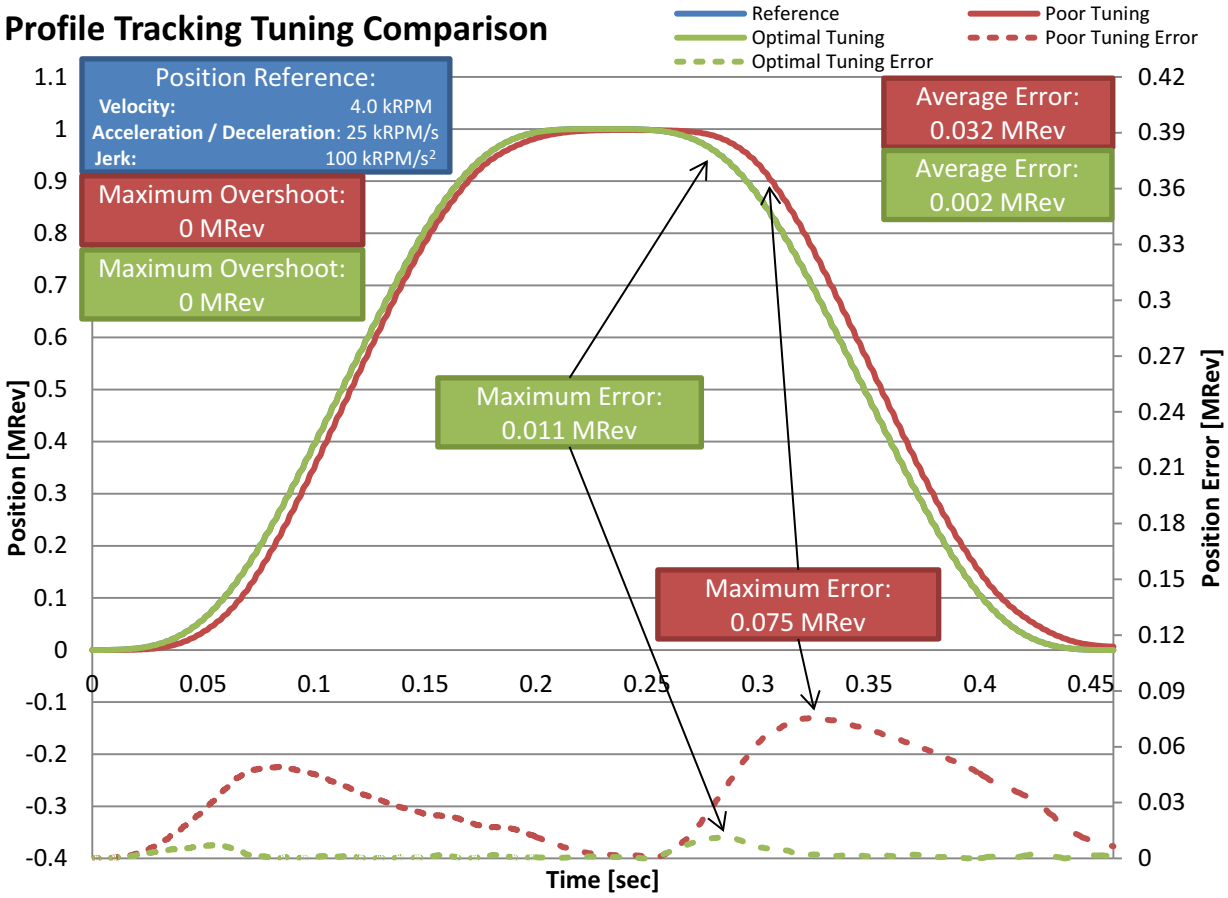


Figure 12-18. Position Tuning Comparison for Profile Tracking

It is important to test multiple speeds and accelerations in your profile as well as multiple different loads. Position controllers have different performance characteristics when placed into different situations. In order to properly evaluate the effectiveness of your position controller, tests should be conducted across the entire application range. This includes when you design the profile for testing. Care needs to be taken to ensure that the application speeds and accelerations are built into the profile. The results of these tests will inform you if your controller will meet the application specifications or if your controller needs to be tuned multiple times for different operating points.

It is also important to be able to create repeatable profiles and loads. Creating a repeatable profile can be done using SpinTAC Position Move and SpinTAC Position Plan; for more information, see Chapter 13. Repeatable profiles are required so that all controllers will be tested using the same reference in the same order and for the same length of time. This ensures that test conditions are as identical as possible. When applying load during a profile tracking test it is important to create repeatable disturbances. This can be accomplished using a dynamometer or a disturbance motor. Creating a repeatable disturbance is an important factor when evaluating controllers. If test conditions cannot be replicated, it is difficult to adequately compare the responses of two controllers.

Copyright © 2013–2019, Texas Instruments Incorporated

12.5.5 InstaSPIN-MOTION Position Control Advantage

12.5.5.1 Single Parameter Tuning

InstaSPIN-MOTION presents numerous advantages in achieving optimal performance for your application. Traditional PI position control requires three cascaded control loops — one for current, one for speed, and one for position — while SpinTAC Position Control requires two loops — one for current and one combined position-velocity loop (see Table 12-4). Because of these cascaded control loops, the PI controllers for velocity and position require at least four tuning parameters, all of which need to be tuned for each operating point in the application.

Table 12-4. InstaSPIN-MOTION Position Control Advantage

Control Loop	Traditional PI Control	SpinTAC Position Control
Current	Automatically identified during parameter identification	Automatically identified during parameter identification
Velocity	Suggested starting values are provided, but require adjustments and testing to validate. Calculations are provided in Section 11.5.	Tuned via a single parameter and is effective across the operating range. Single parameter tunes position and speed, and is effective across the operating range.
Position	No suggested starting values. No calculations provided.	

SpinTAC Position Control helps you achieve optimal performance by offering single parameter tuning for both position and velocity. Having a single tuning parameter allows you to quickly zero in on the right tuning settings for your application. The Active Disturbance Rejection Control (ADRC) at the core of SpinTAC Position Control allows that single tuning parameter to work across a very wide operating range. SpinTAC Position Control reduces the time and complexity required to optimize your application.

To compare the differences between SpinTAC Position Control and a traditional PI control system, the Teknic M2310PLN04K motor (available in the TI eStore) was coupled with a Magtrol HD-400 dynamometer. The PI tuning parameters determined from the example tuning in Section 11.5 were used as a starting point. In order to tune the position PI regulator, the velocity PI regulator had to be re-tuned. This was an iterative process. Each time the velocity gains were modified, the impact on the position gains was evaluated.

SpinTAC Position Control was tuned experimentally by the method outlined in Section 12.4. Prior to tuning SpinTAC Position Control, the system inertia was identified by the procedure outlined in Lab 05c.

12.5.5.2 Disturbance Rejection

The ADRC technology in SpinTAC Position Control has excellent disturbance rejection. It actively estimates and compensates for disturbances in real-time (see Figure 12-19 and Figure 12-20). When SpinTAC Position Control detects a disturbance in the system; it applies a correction to quickly and smoothly bring the motor position back to the target.

Copyright © 2013–2019, Texas Instruments Incorporated

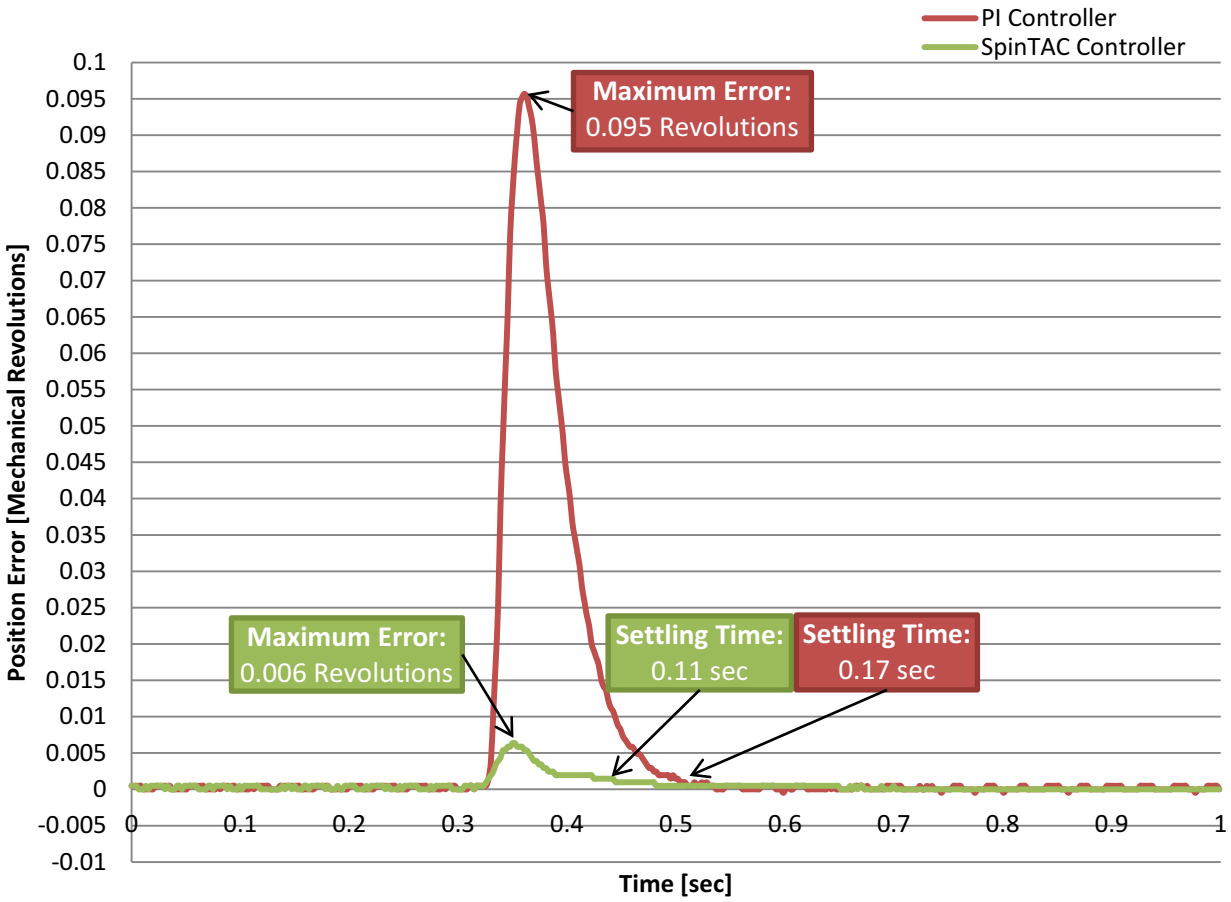


Figure 12-19. PI and SpinTAC Position Control Comparison for Applied Torque Disturbance

Copyright © 2013–2019, Texas Instruments Incorporated

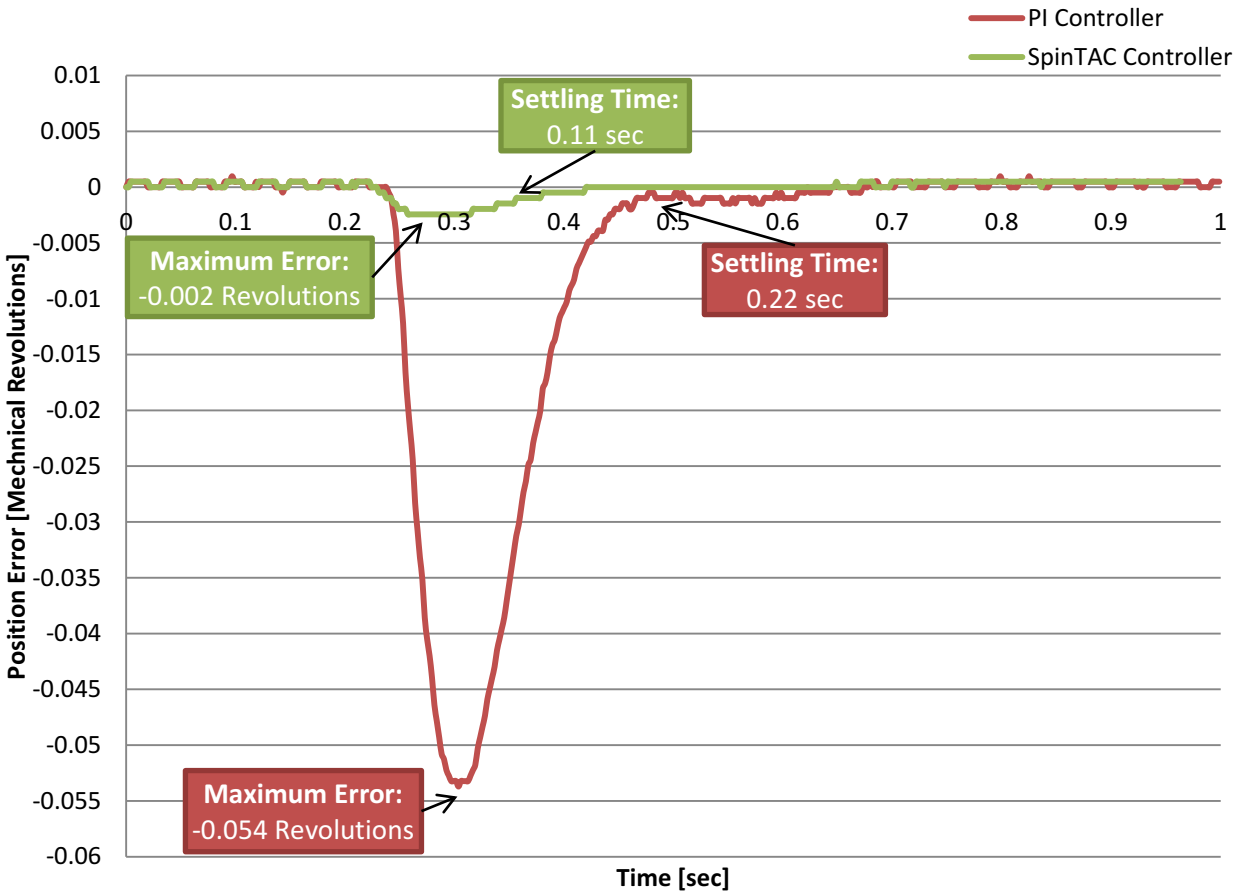


Figure 12-20. PI and SpinTAC Position Control Comparison for Removed Torque Disturbance

SpinTAC Position Control features a much faster recovery time and smaller maximum error than a traditional PI controller. This results in fewer position fluctuations, more consistent performance, and reduced power consumption.

12.5.5.3 Feedforward

SpinTAC Position Control also features feedforward. This allows for excellent profile tracking (see Figure 12-21). Feedforward tells SpinTAC Position Control how fast it should be accelerating or decelerating toward the position target. This allows SpinTAC Position Control to react to profile changes much quicker than a PI controller. It results in less maximum error and less absolute average error.

Copyright © 2013–2019, Texas Instruments Incorporated

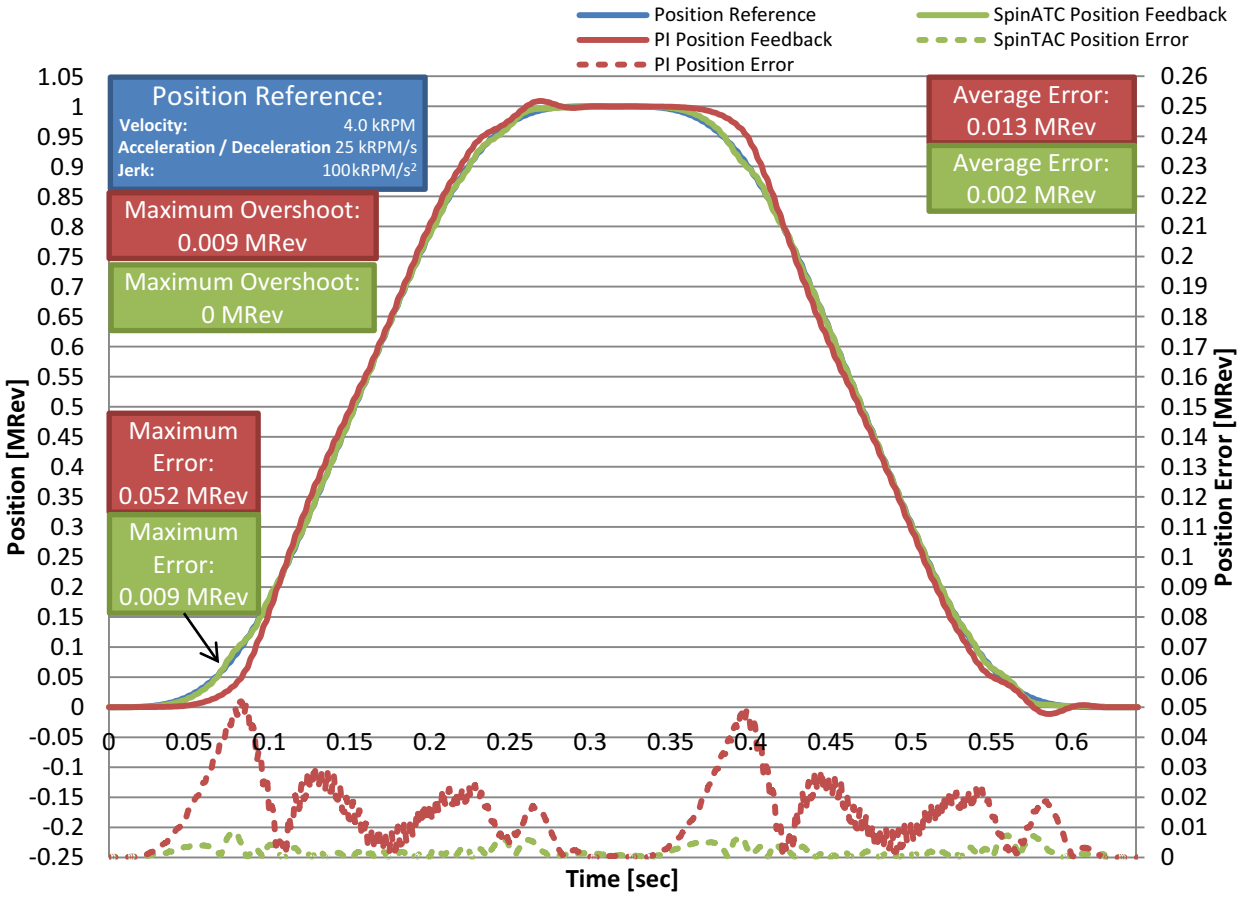


Figure 12-21. PI and SpinTAC Comparison for Feedforward Impact on Position Profile Tracking

SpinTAC Position Control features reduced maximum error and absolute average error. This results in much improved tracking performance over a traditional PI controller. This results in less wasted motion and wasted energy during a reference change.

12.5.5.4 Low-Speed Operation and Smooth Startup

Some applications, such as high-end security and conference room cameras operate at very low speeds (for example, 0.1 rpm) and require accurate and smooth position control to pan, tilt, and zoom. The motors that drive these cameras are difficult to tune for low speed and they usually require a minimum of 4 tuning sets to control both position and speed.

It can be difficult to overcome the system inertia at low speeds, which results in choppy movement at startup, and a shaky or unfocused picture. Figure 12-22 is an example of a very small position movement at a very low speed. SpinTAC is able to more accurately track the reference position resulting in smoother motion than the PI controller. SpinTAC Position Control is equally effective at overcoming system inertia at low speeds and high speeds, and results in very smooth low speed movements.

Copyright © 2013–2019, Texas Instruments Incorporated

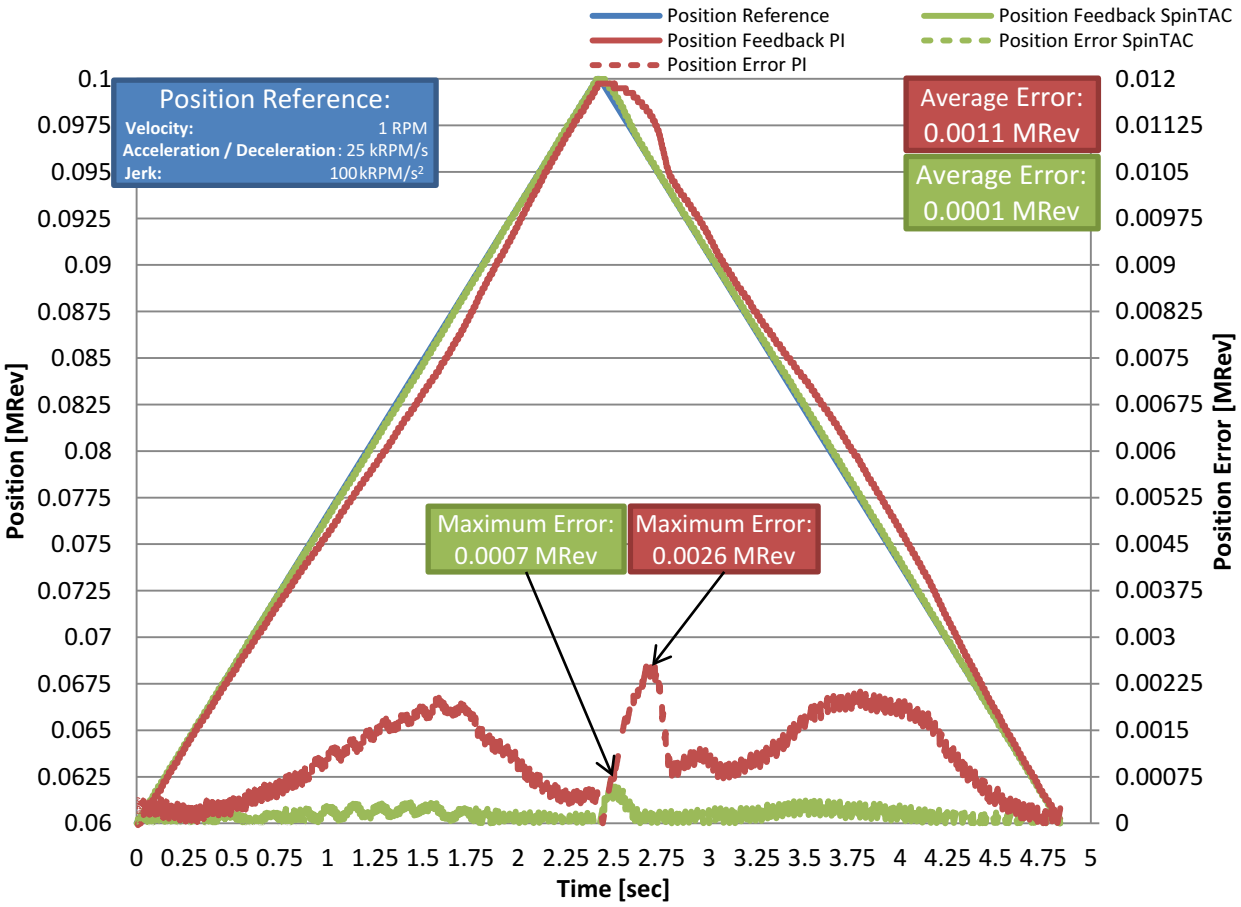


Figure 12-22. PI and SpinTAC Comparison for Low Speed Position Profile Tracking

12.5.5.5 Minimum Step Response Settling Time

SpinTAC Position Control features less settling time for step responses. This results in the system being more responsive to control changes. The system will spend more time at the goal position, which results in less delay. Both controllers have been tuned with zero overshoot, but in situations with changing dynamics, SpinTAC will respond better and will continue to have minimal overshoot when compared with a PI controller.

Figure 12-23 compares the step response of SpinTAC Position Control with a traditional PI position control system. From the plot you can see that the traditional PI position control system has a much longer settling time than SpinTAC Position Control. Longer setting time means that it will take longer for the application to reach the goal position.

Copyright © 2013–2019, Texas Instruments Incorporated

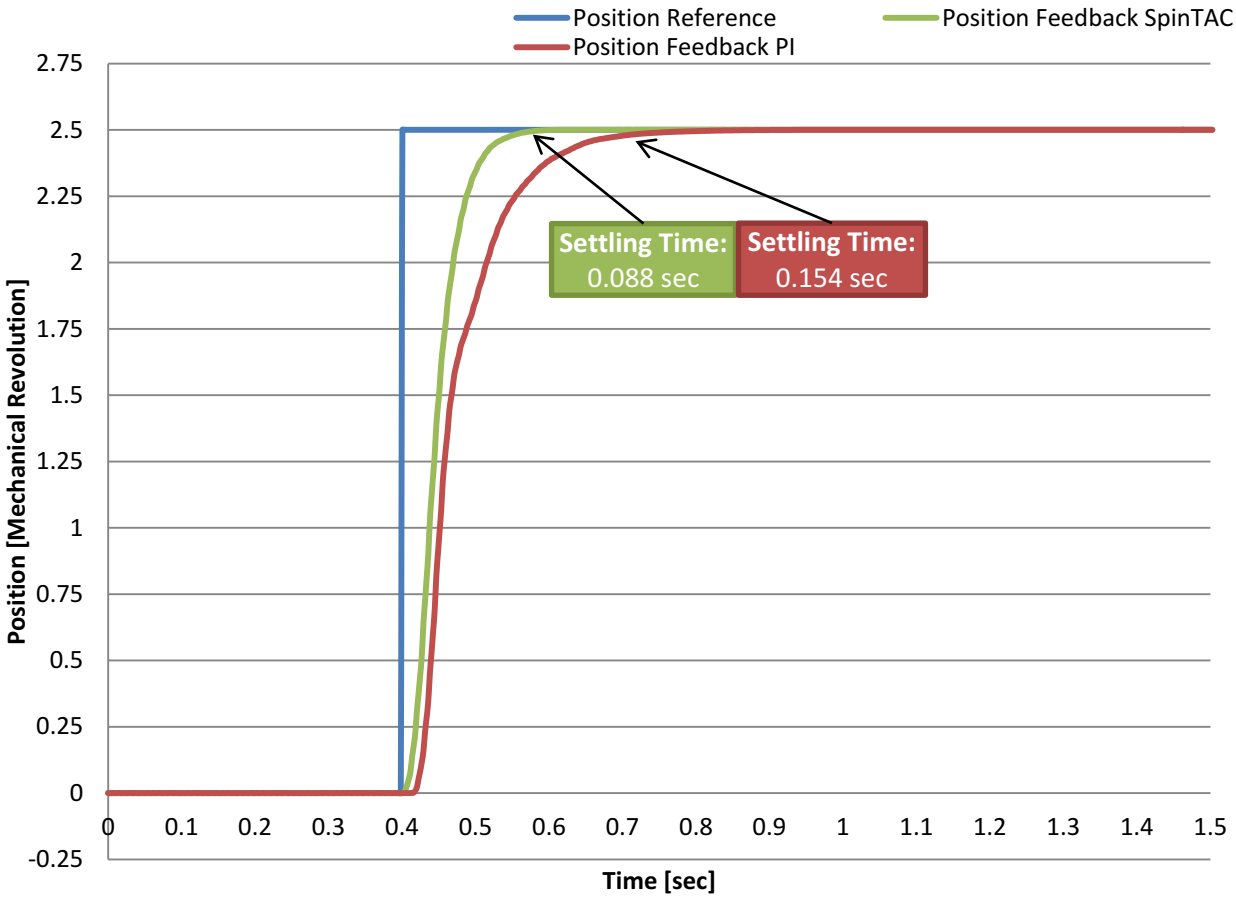


Figure 12-23. PI and SpinTAC Position Control Comparison for Step Response

12.5.5.6 Conclusions

SpinTAC Position Control included with InstaSPIN-MOTION replaces the traditional PI controller for speed and position. It results in better performance across the entire range of your application. It is less complex to tune than a PI controller, since it features a single tuning parameter. This reduces development effort spent tuning the position controller, allowing you to focus on the rest of your application. It results in less overshoot and faster settling time. It also features better profile tracking. All of these features combine to reduce the energy consumed by your application due to wasted motion.

Copyright © 2013–2019, Texas Instruments Incorporated

Trajectory Planning

Controlling the speed or position of a motor is the first step to establishing a motion system. The next step is to establish a method to transition between different speeds and positions, then to sequence the motion of the motor to accomplish the application tasks. InstaSPIN-MOTION allows you to quickly build complex motion sequences with logic-based state transitions.

Topic **Page**

13.1 InstaSPIN-MOTION Profile Generation 454
13.2 Software Configuration for SpinTAC Velocity Move 455
13.3 Software Configuration for SpinTAC Position Move 458
13.4 InstaSPIN-MOTION Sequence Planning 461
13.5 Software Configuration for SpinTAC Velocity Plan 467
13.6 Troubleshooting SpinTAC Velocity Plan 471
13.7 Software Configuration for SpinTAC Position Plan 474
13.8 Troubleshooting SpinTAC Position Plan 478
13.9 Conclusion 481

Copyright © 2013–2019, Texas Instruments Incorporated

13.1 InstaSPIN-MOTION Profile Generation

SpinTAC Move is a constraint-based, time-optimal profile generator. This profile generator calculates the motion profile during run-time without using FLASH lookup tables. This results in a small memory footprint. The user provides constraints (velocity limit [Position Only], acceleration/deceleration limit, and jerk limit) and SpinTAC Move calculates the optimal profile between the current setpoint and the target setpoint. This allows you greater flexibility in designing your application motion profiles.

In addition to the industry standard trapezoidal and s-curve profiles, SpinTAC Move provided the Linestream proprietary st-curve. This curve provides smoother motion changes than either trapezoidal or s-curve profiles. The main feature of the st-curve is the continuous jerk.

Figure 13-1 compares the different curve types available in SpinTAC Position Move. The most notable difference between s-curve and st-curve is the bottom plot, the jerk plot, which shows how the st-curve continuously adjusts the jerk in order to provide even smoother motion than the s-curve. For speed transitions, only the lower three graphs in Figure 13-1 need to be considered

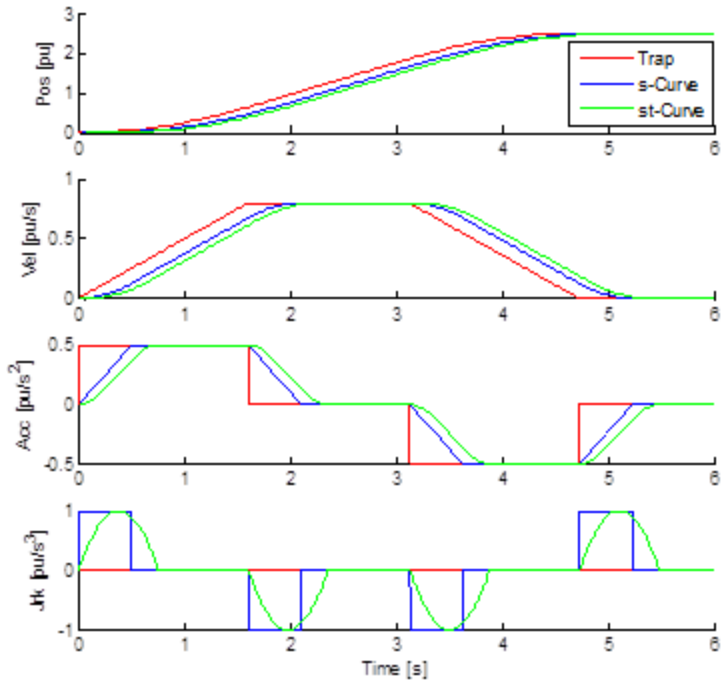


Figure 13-1. Comparison of Curves Provided by SpinTAC Position Move

SpinTAC Move uses sample time based profile generation. This aligns the motion profile with the speed sample time and guarantees that the time to complete a profile will always be a multiple of the sample time. This determinism ensures that for a given set of constraints, SpinTAC Move will always generate an identical profile.

13.1.1 Jerk Impact on System Performance

Jerk represents the rate of change of acceleration. So a larger jerk will allow the acceleration to increase at a faster rate. Jerk is an important factor to consider in applications where fragile objects can only tolerate a limited amount of acceleration changes. Jerk is also critical in applications where rapid changes in acceleration of a cutting tool can lead to premature tool wear or result in uneven cuts. For applications where the system jerk needs to be considered, using SpinTAC Move with the st-curve is essential. Jerk will also have an impact on the amount of current the motor consumes when changing speeds. Lower jerk will cause the motor to consume less current when changing speeds. This is due to the smaller jerk reducing the rate of acceleration increase. For applications where the jerk does not directly need to be considered, it can still have an impact on system performance.

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 13-2 compares three trajectory curves. They have the same start and end velocity and the same acceleration. The jerk for each of these curves has been modified. As the jerk increases the curve reaches the goal speed faster. A consequence of this faster movement is that the motor consumed more current while it was executing the trajectory curve. The maximum current is displayed on the graph. This test was done without any load in the system. If there were a load attached to the system, the increased jerk would have an even more dramatic impact on the maximum current.

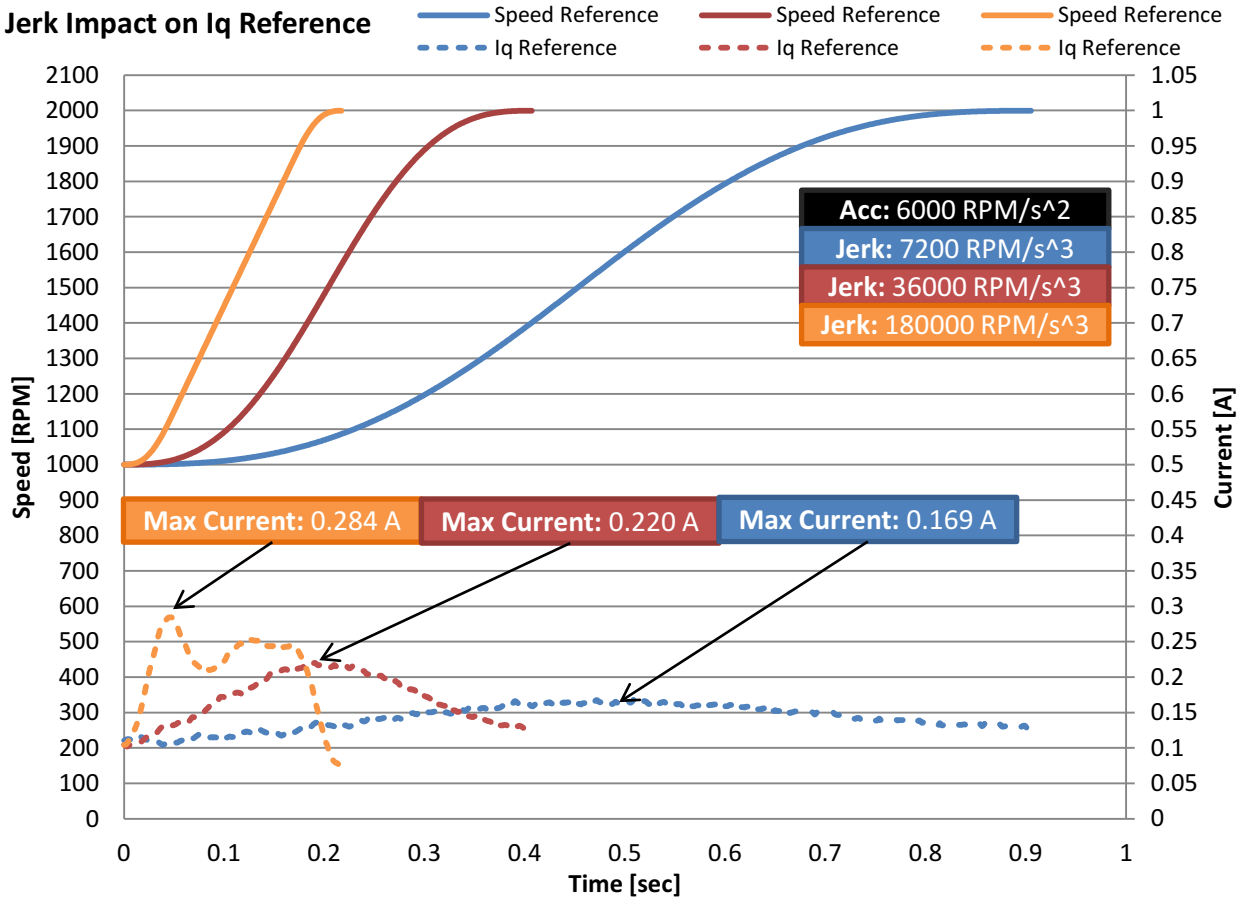


Figure 13-2. Impact of Jerk on Iq Reference

13.2 Software Configuration for SpinTAC Velocity Move

Configuring SpinTAC Velocity Move requires four steps. Lab 6a — Smooth system movement with SpinTAC Move — is an example project that implements the steps required to use SpinTAC Velocity Move to generate trajectory changes. The header file spintac.h, included in MotorWare, allows you to quickly include the SpinTAC components in your project.

13.2.1 Include the Header File

This should be done with the rest of the module header file includes. In the Lab 6a example project, this file is included in the spintac_velocity.h header file. For your project, this step can be completed by including spintac_velocity.h.

```
#include    "sw/modules/spintac/src/32b/spintac_vel_move.h"
```


Copyright © 2013–2019, Texas Instruments Incorporated

13.2.2 Declare the Global Structure

This should be done with the global variable declarations in the main source file. In the Lab 8a project, this structure is included in the ST_Obj structure that is declared as part of the spintac_velocity.h header file.

```
ST_Obj    st_obj;   // The SpinTAC Object
ST_Handle stHandle; // The SpinTAC Handle
```

This example is if you do not wish to use the ST_Obj structure that is declared in the spintac_velocity.h header file.

```
ST_VelMove_t      stVelMove;       // The SpinTAC Velocity Move object
ST_VELMOVE_Handle stVelMoveHandle; // The SpinTAC Velocity Move Handle
```

13.2.3 Initialize the Configuration Variables

This should be done in the main function of the project ahead of the forever loop. This will load all of the default values into SpinTAC Velocity Move. This step can be completed by running the functions ST_init and ST_setupVelMove that are declared in the spintac_velocity.h header file. If you do not wish to use these two functions, the code example below can be used to configure the SpinTAC Velocity Move component. This configuration of SpinTAC Velocity Move represents the typical configuration that should work for most motors.

```
// Initialize the SpinTAC Velocity Move Component
stVelMoveHandle = STVELMOVE_init(&stVelMove, sizeof(ST_VelMove_t));

// Instance of SpinTAC Velocity Move
STVELMOVE_setAxis(stVelMoveHandle, ST_AXIS0);
// Sample time [s], (0, 1]
STVELMOVE_setSampleTime_sec(stVelMoveHandle, _IQ24(ST_SAMPLE_TIME));
// System maximum limit for:
//  speed [pu/s] [IQ24(0.001), _IQ24(1)],
//  acceleration [pu/s^2] [IQ24(0.002), _IQ24(120)],
//  jerk references [pu/s^3] [_IQ20(0.0005), IQ20(2000)]
STVELMOVE_setProfileMaximums(stVelMoveHandle, _IQ24(1.0), _IQ24(10.0), _IQ20(62.5));
// Acceleration limit for the profile [pu/s^2] [IQ24(0.001), _IQ24(120)]
STVELMOVE_setAccelerationLimit(stVelMoveHandle, _IQ24(0.4));
// Jerk limit for the profile [pu/s^3] [_IQ20(0.0005), _IQ20(2000)]
STVELMOVE_setJerkLimit(stVelMoveHandle, _IQ20(1.0));
// Set profile curve type { ST_MOVE_CUR_TRAP, ST_MOVE_CUR_SCRV, ST_MOVE_CUR_STCRV }
STVELMOVE_setCurveType(stVelMoveHandle, ST_MOVE_CUR_STCRV);
// ST_VelMove is not in test mode
STVELMOVE_setTest(stVelMoveHandle, FALSE);
// Initially ST_VelMove is not enabled
STVELMOVE_setEnable(stVelMoveHandle, FALSE);
// Initially ST_VelMove is not in reset
STVELMOVE_setReset(stVelMoveHandle, FALSE);
```

13.2.4 Call SpinTAC Velocity Move

This should be done in the main ISR. This function needs to be called at the proper decimation rate for this component. The decimation rate is established by ST_ISR_TICKS_PER_SPINTAC_TICK; for more information, see Section 4.7.1.4. Before calling SpinTAC Velocity Move function the speed target, acceleration limit, jerk limit, and curve type need to be updated.

```
// If we are not in reset, and the SpeedRef_krpm has been modified
if((STVELMOVE_getReset(stVelMoveHandle) == FALSE)
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
   && (_IQmpy(gMotorVars.SpeedRef_krpm, _IQ24(ST_SPEED_PU_PER_KRPM))
       != STVELMOVE_getVelocityEnd(stVelMoveHandle)))
{
   // Get the configuration for SpinTAC Velocity Move
   STVELMOVE_setCurveType(stVelMoveHandle, gMotorVars.SpinTAC.VelMoveCurveType);
   STVELMOVE_setVelocityEnd(stVelMoveHandle,
                         _IQmpy(gMotorVars.SpeedRef_krpm, _IQ24(ST_SPEED_PU_PER_KRPM)));
   STVELMOVE_setAccelerationLimit(stVelMoveHandle,
                       _IQmpy(gMotorVars.MaxAccel_krpmps, _IQ24(ST_SPEED_PU_PER_KRPM)));
   STVELMOVE_setJerkLimit(stVelMoveHandle,
                    _IQ20mpy(gMotorVars.MaxJrk_krpmps2, _IQ20(ST_SPEED_PU_PER_KRPM)));
   // Enable SpinTAC Move
   STVELMOVE_setEnable(stVelMoveHandle, TRUE);
   //If starting from zero speed, enable ForceAngle, otherwise disable ForceAngle
   if(_IQabs(STVELMOVE_getVelocityStart(stVelMoveHandle)) < _IQ24(ST_MIN_ID_SPEED_PU))
   {
     EST_setFlag_enableForceAngle(ctrlObj->estHandle, TRUE);
     gMotorVars.Flag_enableForceAngle = TRUE;
   }
   else
   {
     EST_setFlag_enableForceAngle(ctrlObj->estHandle, FALSE);
     gMotorVars.Flag_enableForceAngle = FALSE;
   }
}

// Run SpinTAC Move
STVELMOVE_run(stVelMoveHandle);
```

13.2.5 Troubleshooting SpinTAC Velocity Move

13.2.5.1 ERR_ID

ERR_ID provides an error code for users. A list of errors defined in SpinTAc Velocity Move and the solutions for these errors are shown in Table 13-1.

Table 13-1. SpinTAC Velocity Move ERR_ID Code

ERR_ID	Problem	Solution
1	Invalid sample time value	Set cfg.T_sec within (0, 0.01]
2	Invalid system maximum velocity value	Set cfg.VelMax within (0, 1]
10	Invalid system maximum acceleration value	Set cfg.AccMax within [0.001, 120]
12	Invalid system maximum jerk value	Set cfg.JrkMax within [0.0005, 1000]
32	Invalid axis ID	Set cfg.Axis within {ST_AXIS0, ST_AXIS1}
1002	Invalid acceleration limit value	Set AccLim within [0.002, cfg.AccMax]
1004	Invalid jerk limit value	Set JrkLim within [0.001, cfg.JrkMax]
1005	Invalid curve type	Set cfg.CUR_MOD within {ST_PRO_TRAP, ST_PRO_SCRV, ST_PRO_STCRV}
1006	Invalid velocity start value	Set cfg.VelStart within [-cfg.VelMax, cfg.VelMax]
1007	Invalid velocity end value	Set VelEnd within [-cfg.VelMax, cfg.VelMax]
4001	Invalid SpinTAC license	Use the chip with valid license
4003	Invalid ROM Version	Use a chip with a valid ROM version or use the SpinTAC library that is compatible with the current ROM version.

Copyright © 2013–2019, Texas Instruments Incorporated

13.3 Software Configuration for SpinTAC Position Move

Configuring SpinTAC Position Move requires four steps. Lab 13b — Position Transitions with SpinTAC Move — is an example project that implements the steps required to use SpinTAC Position Move to generate trajectory changes. The header file spintac_position.h, included in MotorWare, allows you to quickly include the SpinTAC components in your project.

13.3.1 Include the Header File

This should be done with the rest of the module header file includes. In the Lab 6a example project, this file is included in the spintac_position.h header file. For your project, this step can be completed by including spintac_position.h.

```
#include "sw/modules/spintac/src/32b/spintac_pos_move.h"
```

13.3.2 Declare the Global Structure

This should be done with the global variable declarations in the main source file. In the Lab 13b project this structure is included in the ST_Obj structure that is declared as part of the spintac_position.h header file.

```
ST_Obj       st_obj;    // The SpinTAC Object
ST_Handle stHandle;  // The SpinTAC Handle
```

This example is if you do not wish to use the ST_Obj structure that is declared in the spintac_position.h header file.

```
ST_PosMove_t      stPosMove;       // The SpinTAC Position Move object
ST_POSMOVE_Handle stPosMoveHandle; // The SpinTAC Position Move Handle
```

13.3.3 Initialize the Configuration Variables

This should be done in the main function of the project ahead of the forever loop. This will load all of the default values into SpinTAC Position Move. This step can be completed by running the functions ST_init and ST_setupPosMove that are declared in the spintac_position.h header file. If you do not wish to use these two functions, the code example below can be used to configure the SpinTAC Position Move component. This configuration of SpinTAC Position Move represents the typical configuration that should work for most motors.

```
// Initialize the SpinTAC Speed Controller Component
stPosMoveHandle = STPOSMOVE_init(&stPosMove, sizeof(stPosMove));

// Instance of SpinTAC Move
STPOSMOVE_setAxis(stPosMoveHandle, ST_AXIS0);
// Sample time [s], (0, 1]
STPOSMOVE_setSampleTime_sec(stPosMoveHandle, _IQ24(ST_SAMPLE_TIME));
// Set the type of profile to generate {ST_POS_MOVE_VEL_TYPE , ST_POS_MOVE_POS_TYPE}
STPOSMOVE_setProfileType(stPosMoveHandle, ST_POS_MOVE_POS_TYPE);
// Set the maximum value for mechanical revolutions before rollover [MRev]
STPOSMOVE_setMRevMaximum_mrev(stPosMoveHandle, _IQ24(10.0));
// Set the unit conversion values, this will convert between Mrev and pu
STPOSMOVE_setUnitConversion(stPosMoveHandle, USER_IQ_FULL_SCALE_FREQ_Hz,
                                                    USER_MOTOR_NUM_POLE_PAIRS);
// System maximum limit for:
//  speed [pu/s] [IQ24(0.002), _IQ24(1)]
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
//  acceleration [pu/s^2] [IQ24(0.001), _IQ24(120)]
//  deceleration [pu/s^2] [IQ24(0.001), _IQ24(120)]
//  jerk references [pu/s^3] [_IQ20(0.0005), _IQ20(2000)]
STPOSMOVE_setProfileMaximums(obj->posMoveHandle,
                             _IQ24(USER_MOTOR_MAX_SPEED_KRPM * ST_SPEED_PU_PER_KRPM),
                                                  _IQ24(10), _IQ24(10), _IQ20(62.5));
// Velocity limit for the profile [pu/s] [IQ24(0.002), _IQ24(1)]
STPOSMOVE_setVelocityLimit(obj->posMoveHandle,
                           _IQ24(USER_MOTOR_MAX_SPEED_KRPM * ST_SPEED_PU_PER_KRPM));
// Acceleration limit for the profile [pu/s^2] [IQ24(0.001), _IQ24(120)]
STPOSMOVE_setAccelerationLimit(stPosMoveHandle, _IQ24(0.4));
// Deceleration limit for the profile [pu/s^2] [IQ24(0.001), _IQ24(120)]
STPOSMOVE_setDecelerationLimit(obj->posMoveHandle, _IQ24(0.4));
// Jerk limit for the profile [pu/s^3] [_IQ20(0.0005), _IQ20(2000)]
STPOSMOVE_setJerkLimit(stPosMoveHandle, _IQ20(1.0));
// Set profile curve type { ST_MOVE_CUR_TRAP, ST_MOVE_CUR_SCRV, ST_MOVE_CUR_STCRV }
STPOSMOVE_setCurveType(stPosMoveHandle, ST_MOVE_CUR_STCRV);
// ST_PosMove is not in test mode
STPOSMOVE_setTest(stPosMoveHandle, false);
// Initially ST_PosMove is not enabled
STPOSMOVE_setEnable(stPosMoveHandle, false);
// Initially ST_PosMove is not in reset
STPOSMOVE_setReset(stPosMoveHandle, false);
```

13.3.4 Call SpinTAC Position Move

This should be done in the main ISR. This function needs to be called at the proper decimation rate for this component. The decimation rate is established by ST_ISR_TICKS_PER_SPINTAC_TICK; for more information, see Section 4.7.1.4. Before calling SpinTAC Position Move function the speed target, acceleration limit, jerk limit, and curve type need to be updated.

```
// If we are not running a profile, and the PosStep_MRev has been modified
if((STPOSMOVE_getStatus(stObj->posMoveHandle) == ST_MOVE_IDLE)
&& (gMotorVars.PosStepInt_MRev != 0 || gMotorVars.PosStepFrac_MRev != 0)) {
    // Get the configuration for SpinTAC Position Move
   STPOSMOVE_setCurveType(stObj->posMoveHandle, gMotorVars.SpinTAC.PosMoveCurveType);
   STPOSMOVE_setPositionStep_mrev(stObj->posMoveHandle, gMotorVars.PosStepInt_MRev,
                                                        gMotorVars.PosStepFrac_MRev);
   STPOSMOVE_setVelocityLimit(stObj->posMoveHandle,
                        _IQmpy(gMotorVars.MaxVel_krpm, _IQ24(ST_SPEED_PU_PER_KRPM)));
   STPOSMOVE_setAccelerationLimit(stObj->posMoveHandle,
                     _IQmpy(gMotorVars.MaxAccel_krpmps, _IQ24(ST_SPEED_PU_PER_KRPM)));
   STPOSMOVE_setDecelerationLimit(stObj->posMoveHandle,
                     _IQmpy(gMotorVars.MaxDecel_krpmps, _IQ24(ST_SPEED_PU_PER_KRPM)));
   STPOSMOVE_setJerkLimit(stObj->posMoveHandle,
                    _IQ20mpy(gMotorVars.MaxJrk_krpmps2, _IQ20(ST_SPEED_PU_PER_KRPM)));
   // Enable the SpinTAC Position Profile Generator
   STPOSMOVE_setEnable(stObj->posMoveHandle, true);
   // clear the position step command
   gMotorVars.PosStepInt_MRev = 0;
   gMotorVars.PosStepFrac_MRev = 0;

}

STPOSMOVE_run(stObj->posMoveHandle);
```


Copyright © 2013–2019, Texas Instruments Incorporated

13.3.5 Troubleshooting SpinTAC Position Move

13.3.5.1 Position Profile Limits

There are some combinations of limits that will not produce a valid profile. This will only happen when very small limits are provided to SpinTAC Position Move.

- If the velocity limit is set below 0.001 pu/s, the profile is not guaranteed.

  This can result in the motor not moving due to a minimal velocity limit or an error condition. This can also result in the motor moving in the opposite direction of intended motion; this is due to mathematical overflow. There are cases where setting a velocity limit less than 0.001 pu/s will generate a valid profile and any profile with a velocity limit greater than or equal to 0.001 pu/s will be valid.

13.3.5.2 ERR_ID

ERR_ID provides an error code for users. A list of errors defined in SpinTAC Position Move and the solutions for these errors are shown in Table 13-2.

Table 13-2. SpinTAC Position Move ERR_ID Code

ERR_ID	Problem	Solution
1	Invalid sample time value	Set cfg.T_sec within (0, 0.01)
2	Invalid system maximum velocity value	Set cfg.VelMax within [0.002, 1]
10	Invalid system maximum acceleration value	Set cfg.AccMax within [0.002, 120]
11	Invalid system maximum deceleration value	Set cfg.DecMax within [0.002, 120] and cfg.DecMax / cfg.AccMax within [1, 10]
12	Invalid system maximum jerk value	Set cfg.JrkMax within [0.001, 2000]
13	Invalid position rollover bound value	Set cfg.ROMax_mrev within [2, 100]
14	Invalid mechanical revolution [MRev] to [pu] ratio value	Set cfg.mrev_TO_pu within [0.008, 1]
32	Invalid axis ID	Set cfg.Axis within {ST_AXIS0, ST_AXIS1}
1001	Invalid velocity limit value	Set VelLim within (0, cfg.VelMax]
1002	Invalid acceleration limit value	Set AccLim within [0.001, cfg.AccMax]
1003	Invalid deceleration limit value	Set DecLim within [0.001, cfg.DecMax], and DecLim / AccLim within [1, 10]
1004	Invalid jerk limit value	Set JrkLim within [0.0005, cfg.JrkMax]
1005	Invalid curve type	Set cfg.CurveType within {ST_MOVE_CUR_TRAP, ST_MOVE_CUR_SCRV, ST_MOVE_CUR_STCRV}
1006	Invalid velocity start value	Set cfg.VelStart within [-cfg.VelMax, cfg.VelMax]
1007	Invalid velocity end value	Set VelEnd within [-cfg.VelMax, cfg.VelMax]
1008	Invalid position start value	Set cfg.PosStart_mrev within [-cfg.ROMax, cfg.ROMax]
1009	Invalid position step value	Set PosStepInt_mrev within [-2147483647, 2147483647] and PosStepFrac_mrev within (-1, 1)
1101	Calculation overflow, VelLim out of the range	This error occurs when the fixed-point calculation overflows. Typical cases are: VelLim or PosStep is too small. The action is to increase the values.
1102	Calculation overflow, AccLim out of the range	
1103	Calculation overflow, DecLim out of the range	
1104	Calculation overflow, JrkLim out of the range	
1105	Invalid profile mode value	Set cfg.ProfileType within {ST_POS_MOVE_VEL_TYPE, ST_POS_MOVE_POS_TYPE}
2001	Invalid mode switching from ST_POS_MOVE_VEL_TYPE to ST_POS_MOVE_POS_TYPE	Set the profile to reach zero speed, and then switch to position-controlled profile by setting cfg.ProfileType = ST_POS_MOVE_POS_TYPE
4001	Invalid SpinTAC license	Use the chip with valid license for SpinTAC
4003	Invalid ROM Version	Use a chip with a valid ROM version or use the SpinTAC library that is compatible with the current ROM version.

Copyright © 2013–2019, Texas Instruments Incorporated

13.4 InstaSPIN-MOTION Sequence Planning

SpinTAC Velocity Plan is a motion sequence planner. This allows you to build the motion sequence of your application without constructing a finite state machine. SpinTAC Velocity Plan contains many advanced features to enable complex finite state machines. It features conditional transitions allowing your motor to transition to one of many possible states. It also features variables that can be used to interface with external components (like sensors or actuators) or used as internal counters (to track the number of events within a particular state). SpinTAC Velocity Plan can be configured at run-time and can switching between multiple state machines.

SpinTAC Plan can work in either a position solution or a velocity solution. The features and functionality of SpinTAC Plan is the same. The only difference is during configuration for SpinTAC Position Plan there are some additional fields that need to be configured.

13.4.1 SpinTAC Velocity Plan Elements

SpinTAC Velocity Plan features elements that work together to generate a motion sequence. The different elements are: States, Transitions, Conditions, Variables, and Actions. Each of these elements is configured through separate API calls. The API details can be found in Section 3.5

13.4.1.1 States

States describe the steady operation of the profile. The user specifies the end speed (SpinTAC Velocity Plan) or position step (SpinTAC Position Plan) and the minimum time that SpinTAC Velocity Plan should remain in a state before transitioning to another state. In the example of a washing machine the states are defined as: Idle, Fill, Agitate CW, Agitate CCW, Drain, and Spin.

13.4.1.2 Transitions

Transitions define the allowable moves between states. They establish the connections between the states. Transitions allow the move between states to occur if a condition has been fulfilled. The user specifies the initial and target states, the profile limits, and the conditions to be evaluated prior to the transition.

13.4.1.3 Conditions

Conditions provide logical checks within transitions or action. A transition or action may have a maximum of two conditions. The condition(s) must be satisfied before the motor can transition from one state to the next or for the action to occur. To determine whether a condition is satisfied, a variable is compared against a specific value or value range. This returns a true or false value based on the criteria.

13.4.1.4 Variables

Variables allow SpinTAC Velocity Plan to interact with the rest of the project. There are three types of variables in SpinTAC Velocity Plan: input, output, and input-output. Input variables are used to receive values from outside SpinTAC Velocity Plan, and to evaluate conditions. Output variables are used to interact with the rest of the system. Output variables can be modified by SpinTAC Velocity Plan, but will not be used by SpinTAC Velocity Plan to check conditions. Note: The user must write the code that performs the event associated with the output variable (e.g., open a valve). Input-output variables are typically used as counters or timers and are used by actions or conditions.

13.4.1.5 Actions

Actions change the value of variables. Actions set a variable equal to a value, or add a value to a variable. Actions may take place within a specified state, or when SpinTAC VelocityPlan enters or exits that state. Actions may have associated conditions. This allows an action to occur only when the condition is satisfied. If an action is configured as an ENTER action, SpinTAC Plan will start evaluating the conditions of that action upon entering the state. The action will take place once, only after the conditions are satisfied. Similarly, if an action is configured as an EXIT action, SpinTAC Plan will start evaluating the conditions of that action when leaving the state. When the conditions are satisfied, the actions take place once.

Copyright © 2013–2019, Texas Instruments Incorporated

13.4.2 SpinTAC Velocity Plan Element Limits

SpinTAC Velocity Plan does not have a maximum number of elements that can be configured. The limit is based on the amount of memory that you would like to commit to the configuration of SpinTAC Velocity Plan. This is done in order to be both efficient as to how SpinTAC Velocity Plan is using the system memory and be flexible to allow for custom configurations of the element. Each element of SpinTAC Velocity Plan has a different memory foot print. These are collected in Table 13-3.

Table 13-3. Memory Requirements for SpinTAC Velocity Plan Elements

Plan Element	SpinTAC Velocity Plan (double words)	SpinTAC Position Plan (double words)
Actions	5	5
Conditions	3	3
Variables	2	2
Transitions	5	7
States	4	7

This additional flexibility requires you to declare a configuration array whose address needs to be passed into SpinTAC Velocity Plan. This configuration array needs to be sized according to how many elements are in your Plan. Table 13-3 provides the memory requirements for each element. It is a best practice to declare enumerations for the Plan elements that you wish to use. This makes it simple to calculate the amount of memory that is required for the configuration array.

13.4.2.1 Example of Sizing SpinTAC Velocity Plan Configuration Array

Our example state machine features the following elements:

- 4 Actions
- 3 Conditions
- 3 Variables
- 6 Transitions
- 3 States

This will require 83 double words of configuration space. This value is calculated from the above number of elements and the memory usage contained in Table 13-3.

```
4 Actions     * 5 Double Words = 20 Double Words
3 Conditions  * 3 Double Words = 9 Double Words
3 Variables   * 2 Double Words = 6 Double Words
6 Transitions * 5 Double Words = 30 Double Words
3 States      * 4 Double Words = 12 Double Words

Adding this together
20 + 9 + 6 + 30 + 12 = 83 Double Words

The declaration of the SpinTAC Velocity Plan configuration array should be as follows
uint32_t stVelPlanCfgArray[83];
```

An additional example can be found in the project Lab 6c, Motion Sequence Real World Example: Washing Machine. This also provides an excellent example of how to use enumerations in order to simplify the sizing of the SpinTAC Velocity Plan configuration array.

To size the configuration array for SpinTAC Position Plan you would need to follow the same procedure as outlined above, but use the memory usage for SpinTAC Position Plan.

Copyright © 2013–2019, Texas Instruments Incorporated

13.4.3 SpinTAC Velocity Plan Example: Washing Machine Agitation

Another example to introduce the basic elements of SpinTAC Velocity Plan is the agitation stage of a washing machine. The agitation cycle is a basic motion profile. In this example SpinTAC Velocity Plan does not interface to any external sensors or valves and has no conditional transitions. This motion sequence can easily be implemented in SpinTAC Velocity Plan. Figure 13-3 shows the state transition map for the washing machine agitation.

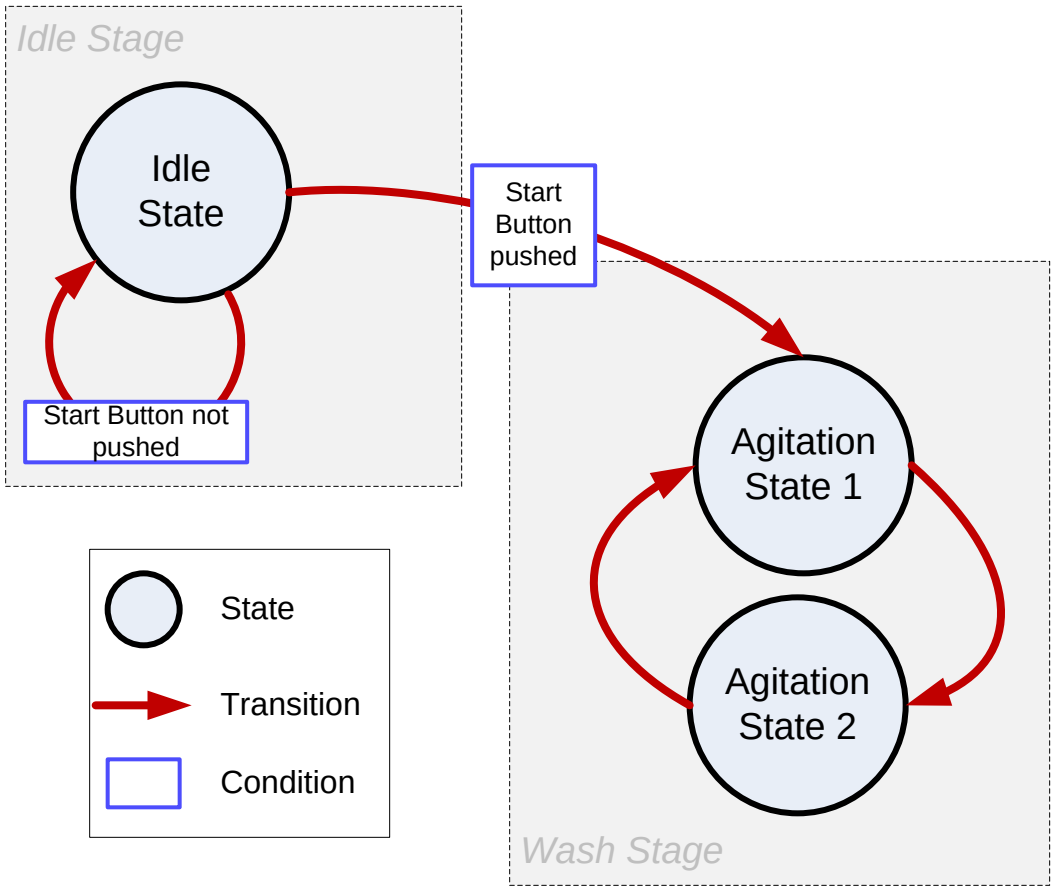


Figure 13-3. State Transition Map of Example Washing Machine Agitation

The washing machine agitation has two stages: Idle and Wash. The washing machine will stay in the idle state until the start button is pushed. Once the start button is pushed the machine will go into the wash stage where it will agitate between a positive and a negative speed until the washing machine agitation state machine is told to stop.

Figure 13-6 describes the motor velocity during the entire washing machine motion sequence, but it can also be used to describe the motor velocity during the washing machine agitation motion sequence. Refer to the wash stage section of Figure 13-6 and you should see the motor velocity run in the positive direction followed by the negative direction. This represents the washing machine agitation stage.

13.4.4 SpinTAC Velocity Plan Example: Garage Door

An example to introduce the basic elements of SpinTAC Plan is a garage door system. The garage door is a basic motion profile that includes conditional transitions, variables, and actions. This introduces all of the different components of SpinTAC Velocity Plan. This motion sequence can be easily implemented in SpinTAC Velocity Plan. Figure 13-4 shows the state transiton map of the example garage door.

Copyright © 2013–2019, Texas Instruments Incorporated

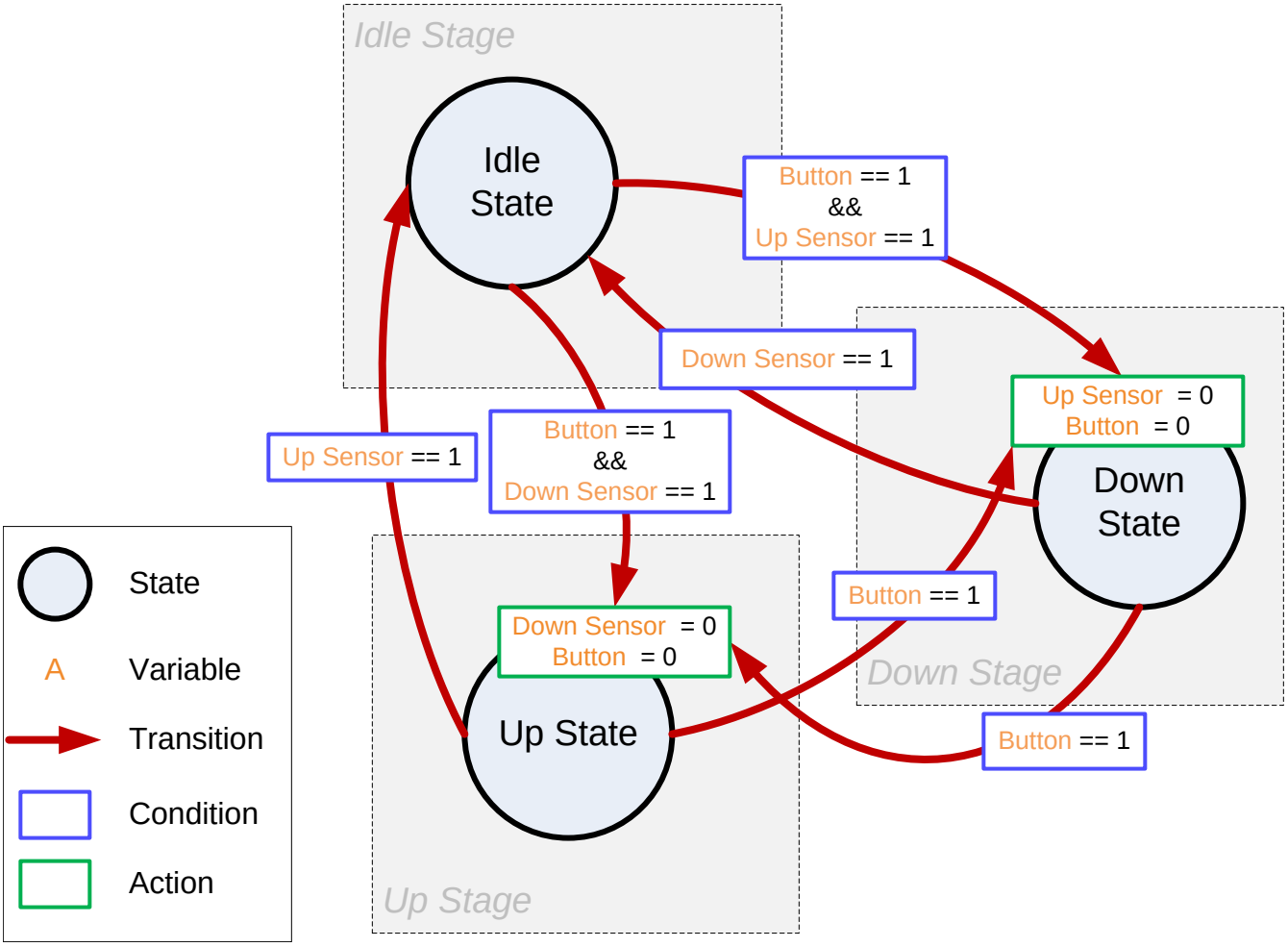


Figure 13-4. State Transition Map of Example Garage Door

The garage door has three stages: Idle, Up, and Down. The garage door will stay in the idle state until the Button is pressed. Once the Button is pressed, the garage door will transition either Up or Down depending on the current position. If the Button is pressed while the garage door is transitioning up or down, it will change direction.

13.4.5 SpinTAC Velocity Plan Example: Washing Machine

A great example of the use for SpinTAC Velocity Plan is in a washing machine. A washing machine has a complex motion sequence. In this example, SpinTAC Velocity Plan interfaces to sensors and valves, and has conditional state transitions. This entire motion sequence can be easily implemented in SpinTAC Velocity Plan. Figure 13-5 shows the state transition map for the washing machine.

Copyright © 2013–2019, Texas Instruments Incorporated

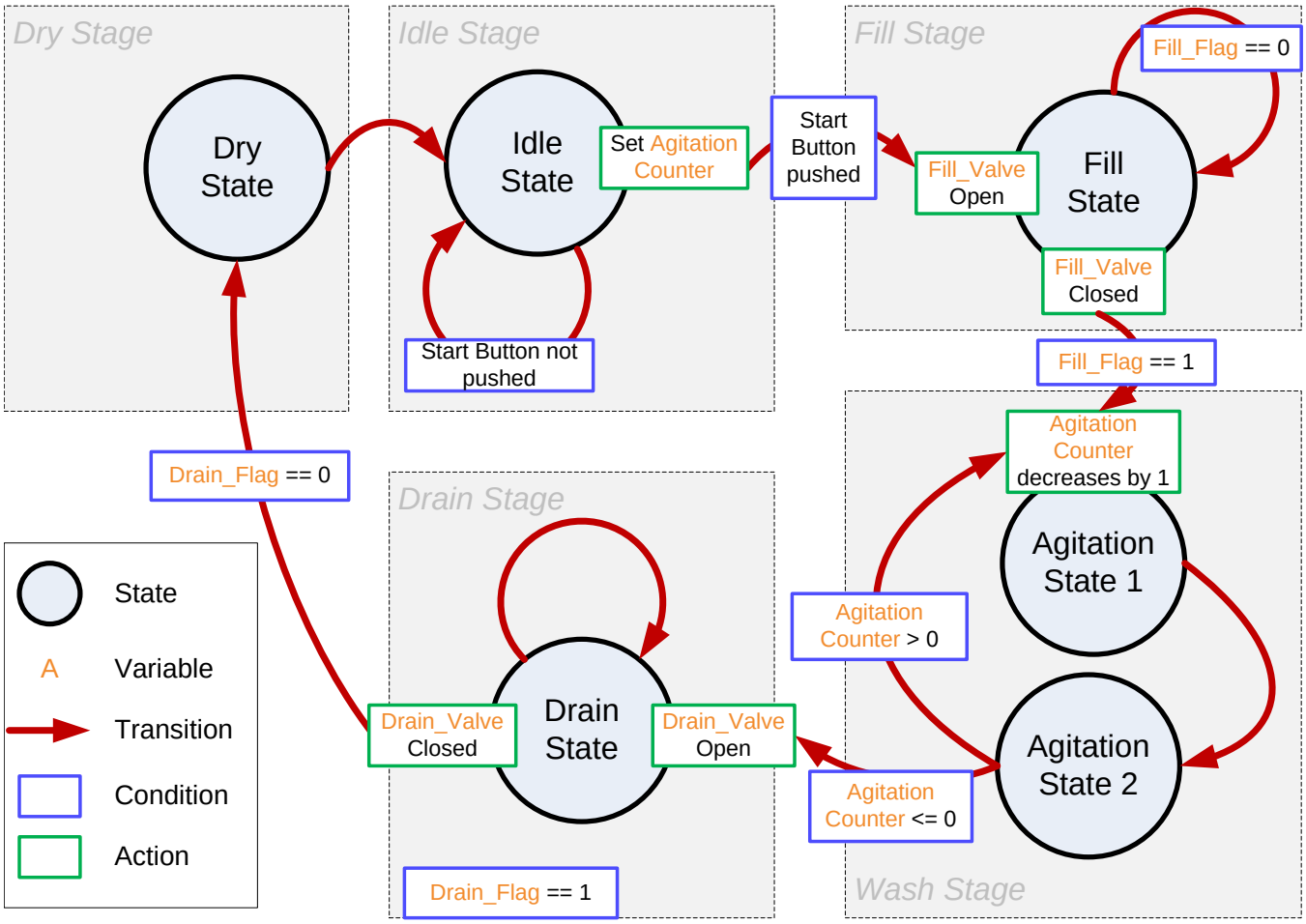


Figure 13-5. State Transition Map of Example Washing Machine

The washing machine has five stages: Idle, Fill, Wash, Drain, and Dry.

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, representing the number of agitation cycles to be performed.

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 application 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 turned closed, and it 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.

Figure 13-6 describes the motor velocity profile during the washing machine motion sequence. The motor will wait at 0 RPM until the fill stage is complete. At this point it will go through 20 agitation cycles oscillating between 250 RPM and -250 RPM. After the 20 agitations, the motor will return to 0 RPM until the water has finished draining from the washing machine. Upon exiting the drain stage, the motor will spin up to 2000 RPM in order to dry the clothes. At the conclusion of the dry stage, the motor will return to 0 RPM and the idle state.

Copyright © 2013–2019, Texas Instruments Incorporated

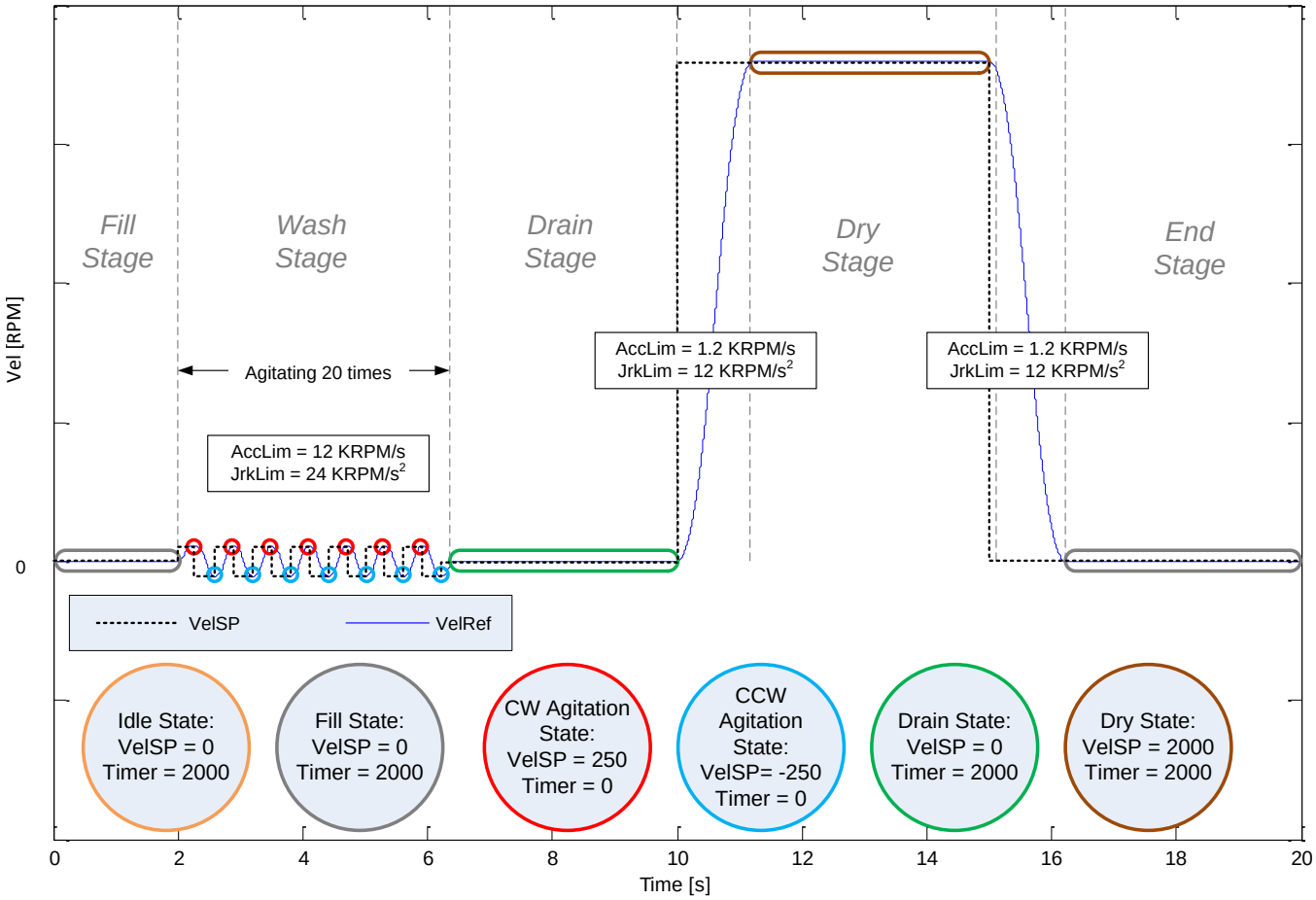


Figure 13-6. Velocity Profile During Example Washing Machine

13.4.6 SpinTAC Position Plan Example: Vending Machine

An example for the use of SpinTAC Position Plan is a vending machine. In this example the vending machine rotates in a circle and will only vend one item at a time. The state transition map for this example is in Figure 13-7.

Copyright © 2013–2019, Texas Instruments Incorporated

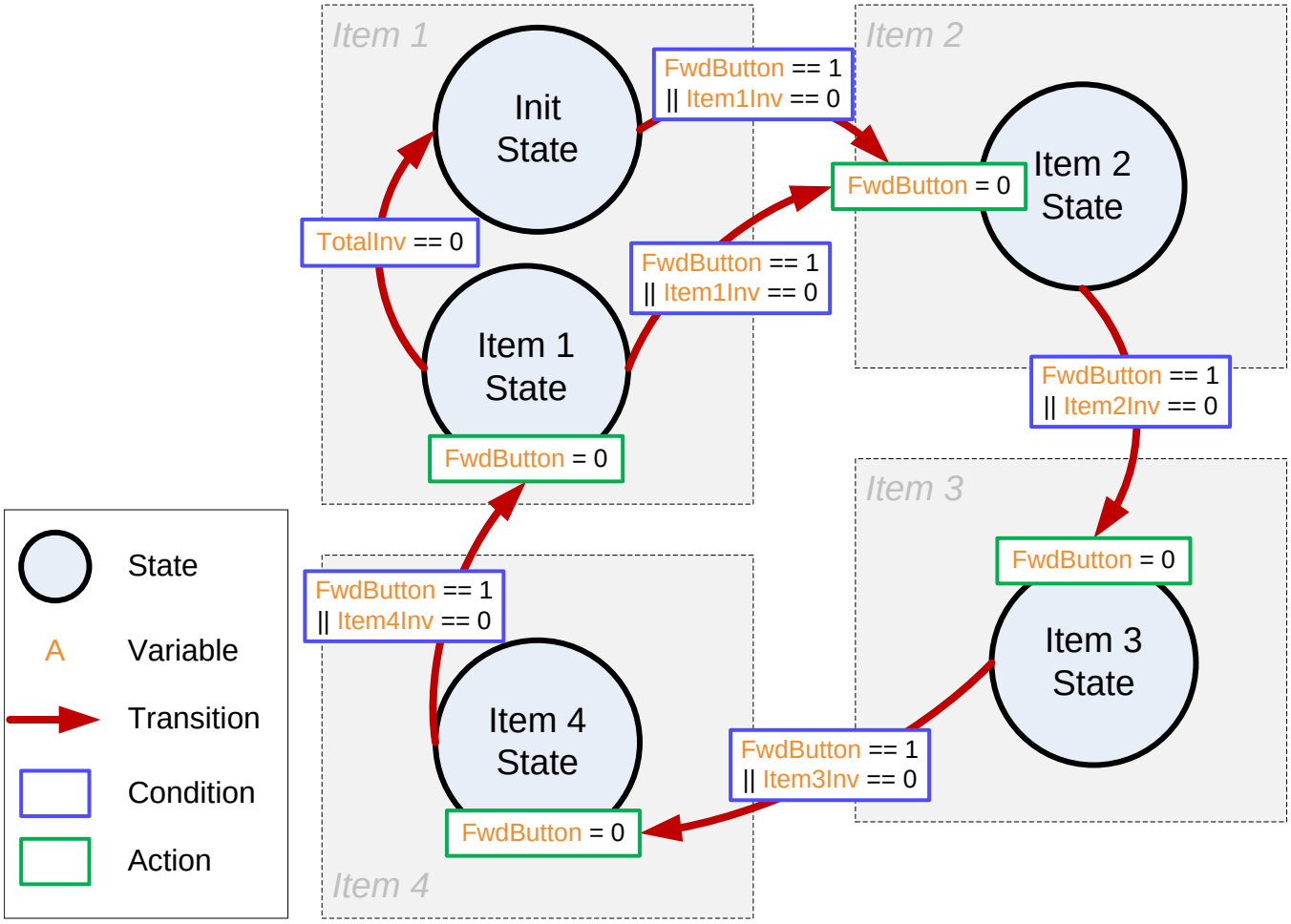


Figure 13-7. State Transition Map of Example Vending Machine

In this example the vending machine will display one item until the user presses the Forward Button (FwdButton). Once that button is detected the vending machine will advance one item and display the next item that is can vend.

When the user removes an item from the vending machine, the application will update that item's inventory. When the inventory is reduced to zero, the vending machine will skip that state. If all item's inventories are reduced to zero, the vending machine will return to the Init state and SpinTAC Position Plan will stop. This indicates that the vending machine needs to be refilled.

This example of SpinTAC Position Plan is implemented in Lab 13d, Motion Sequence Real World Example: Vending Machine.

13.5 Software Configuration for SpinTAC Velocity Plan

Configuring SpinTAC Velocity Plan requires seven steps. Lab 6c — Motion Sequence Real World Example: Washing Machine — is an example project that implements the steps required to use SpinTAC Velocity Plan. The header file spintac_velocity.h, included in MotorWare, allows you to quickly include the SpinTAC components in your project.

Copyright © 2013–2019, Texas Instruments Incorporated

13.5.1 Include the Header File

This should be done with the rest of the module header file includes. In Lab 6c example project this file is included in the spintac_velocity.h header file. For your project, this step can be completed by including spintac_velocity.h.

```
#include "sw/modules/spintac/src/32b/spintac_vel_plan.h"
```

13.5.2 Define the Size of the Configuration Array

This should be done with the rest of the MACRO defines. In the Lab 6c example project, this is done at the top of the main source file. Typically it is a best practice to use enumerations to define and label the states in your Plan. This allows you to easily size the configuration array to meet your application requirements. In the Lab 6c example project this step is done for you. Sizing the configuration array for your motion sequence is covered in Section 13.4.2.

```
#define ST_VELPLAN_CFG_ARRAY_DWORDS ((ST_VEL_PLAN_ACT_DWORDS   * 6) + \
                                     (ST_VEL_PLAN_COND_DWORDS  * 4) + \
                                     (ST_VEL_PLAN_VAR_DWORDS   * 5) + \
                                     (ST_VEL_PLAN_TRAN_DWORDS  * 7) + \
                                     (ST_VEL_PLAN_STATE_DWORDS * 6))
```

13.5.3 Declare the Global Structure

This should be done with the global variable declarations in the main source file. In the Lab 6c project, this structure is included in the ST_Obj structure that is declared as part of the spintac_velocity.h header file.

```
ST_Obj st_obj;      // The SpinTAC Object
ST_Handle stHandle; // The SpinTAC Handle
unsigned long gWaterLevel = 0;     // Stores the water level in the washer
_iq gVelPanVar[ST_PLAN_MAX_VAR_NUM];  // Stores the values of SpinTAC Plan variables
// Configuration array for SpinTAC Plan
uint32_t stVelPlanCfgArray[ST_VELPLAN_CFG_ARRAY_DWORDS];
```

This example is if you do not wish to use the ST_Obj structure that is declared in the spintac_velocity.h header file.

```
ST_VelPlan_t      stVelPlan;        //The SpinTAC Plan Object
ST_VELPLAN_Handle stVelPlanHandle; // The SpinTAC Plan Handle
uint32_t gWaterLevel = 0;     // Stores the water level in the washer
_iq gVelPanVar[ST_PLAN_MAX_VAR_NUM];   // Stores the values of SpinTAC Plan variables
// Configuration array for SpinTAC Plan
uint32_t stVelPlanCfgArray[ST_VELPLAN_CFG_ARRAY_DWORDS];
```

13.5.4 Initialize the Configuration Variables

This should be done in the main function of the project ahead of the forever loop. This will load all of the default values into SpinTAC Velocity Plan. This step can be completed by running the function ST_init that is declared in the spintac_velocity.h header file and the function ST_setupVelPlan that is declared in the main.c. If you do not wish to use these two functions, the code example below can be used to configure the SpinTAC Velocity Plan component. This example loads the washing machine profile that is discussed in Section 13.4.5. For more information about the SpinTAC Plan API, see Section 3.5.3.1.

```
// init the ST VelPlan object
stVelPlanHandle = STVELPLAN_init(&stVelPlan, sizeof(ST_VelPlan_t));
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
// Pass the configuration array pointer into SpinTAC Velocity Plan
// Parameters: handle, pointer to array, size of array, number of actions, number of
conditions, number of variables, number of transitions, number of states
STVELPLAN_setCfgArray(stVelPlanHandle, &stVelPlanCfgArray[0], sizeof(stVelPlanCfgArray), 6, 4,
5, 7, 6);

// Establish the Velocity, Acceleration, and Jerk Maximums
_iq velMax = STVELMOVE_getVelocityMaximum(stVelMoveHandle);
_iq accMax = STVELMOVE_getAccelerationMaximum(stVelMoveHandle);
_iq jrkMax = STVELMOVE_getJerkMaximum(stVelMoveHandle);

// Configure SpinTAC Velocity Plan: Sample Time, VelMax, AccMax, DecMax, JrkMax, LoopENB
STVELPLAN_setCfg(stVelPlanHandle, _IQ24(ST_SPEED_SAMPLE_TIME),
                                velMax, accMax, jrkMax, FALSE);
// Configure halt state: VelEnd, AccMax, JrkMax, Timer
STVELPLAN_setCfgHaltState(stVelPlanHandle, 0, accMax, jrkMax, 1000L);

//Example: STVELPLAN_addCfgState(handle, VelSetpoint[pups], StateTimer[ticks]);
//StateIdx0: Idle
STVELPLAN_addCfgState(stVelPlanHandle, 0,                                  2000L);
// StateIdx1: Fill
STVELPLAN_addCfgState(stVelPlanHandle, 0,                                  2000L);
// StateIdx2: AgiCW
STVELPLAN_addCfgState(stVelPlanHandle, _IQ24(0.25 * ST_SPEED_PU_PER_KRPM),  200L);
// StateIdx3: AgiCCW
STVELPLAN_addCfgState(stVelPlanHandle, _IQ24(-0.25 * ST_SPEED_PU_PER_KRPM), 200L);
// StateIdx4: Drain
STVELPLAN_addCfgState(stVelPlanHandle, 0,                                  2000L);
// StateIdx5: Dry
STVELPLAN_addCfgState(stVelPlanHandle, _IQ24(2 * ST_SPEED_PU_PER_KRPM),     2000L);

//Example: STVELPLAN_addCfgVar(handle, VarType,      InitialValue);
// VarIdx0: FillSensor {0: not filled; 1: filled}
STVELPLAN_addCfgVar(stVelPlanHandle,   ST_VAR_IN,    0);
// VarIdx1: DrainSensor {0: not drained; 1: drained}
STVELPLAN_addCfgVar(stVelPlanHandle,   ST_VAR_IN,    0);
// VarIdx2: CycleCounter
STVELPLAN_addCfgVar(stVelPlanHandle,   ST_VAR_INOUT, 0);
// VarIdx3: FillValve {0: valve closed; 1: valve open}
STVELPLAN_addCfgVar(stVelPlanHandle,   ST_VAR_OUT,   0);
// VarIdx4: DrainValve {0: valve closed; 1: valve open}
STVELPLAN_addCfgVar(stVelPlanHandle,   ST_VAR_OUT,   0);

//Example: STVELPLAN_addCfgCond(handle, VarIdx, Comparison,  Value1, Value2)
// CondIdx0: WaterFull Water is filled
STVELPLAN_addCfgCond(stVelPlanHandle,   0,      ST_COMP_EQ,  1,     0);
// CondIdx0: AgiNotDone SgitCycleCounter is greater than 0 (not done)
STVELPLAN_addCfgCond(stVelPlanHandle,   2,      ST_COMP_GT,  0,     0);
// CondIdx1: AgiDone SgitCycleCounter is equal or less than 0 (done)
STVELPLAN_addCfgCond(stVelPlanHandle,   2,      ST_COMP_ELW, 0,     0);
// CondIdx0: WaterEmpty Water is drained
STVELPLAN_addCfgCond(stVelPlanHandle,   1,      ST_COMP_EQ,  1,     0);
// Note: Set Value2 to 0 if Comparison is for only one value.

//Example: STVELPLAN_addCfgTran(handle, FromState, ToState, CondOption, CondIdx1,
CondiIdx2, AccLim[pups2], JrkLim[pups3]);
// From IdleState to FillState
STVELPLAN_addCfgTran(stVelPlanHandle, 0, 1, ST_COND_NC, 0, 0, _IQ24(0.1), _IQ20(1));
// From FillState to AgiState1
STVELPLAN_addCfgTran(stVelPlanHandle, 1, 2, ST_COND_FC, 0, 0, _IQ24(0.1), _IQ20(1));
// From AgiState1 to AgiState2
STVELPLAN_addCfgTran(stVelPlanHandle, 2, 3, ST_COND_NC, 0, 0, _IQ24(1), _IQ20(1));
// From AgiState2 to AgiState1
STVELPLAN_addCfgTran(stVelPlanHandle, 3, 2, ST_COND_FC, 1, 0, _IQ24(1), _IQ20(1));
// From AgiState2 to DrainState
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
STVELPLAN_addCfgTran(stVelPlanHandle, 3, 4, ST_COND_FC, 2, 0, _IQ24(0.1), _IQ20(1));
// From DrainState to DryState
STVELPLAN_addCfgTran(stVelPlanHandle, 4, 5, ST_COND_FC, 3, 0, _IQ24(0.2), _IQ20(1));
// From DryState to IdleState
STVELPLAN_addCfgTran(stVelPlanHandle, 5, 0, ST_COND_NC, 0, 0, _IQ24(0.1), _IQ20(1));
// Note: set CondIdx1 to 0 if CondOption is ST_COND_NC; set CondIdx2 to 0 if CondOption is
ST_COND_NC or ST_COND_FC

//Example: STVELPLAN_addCfgAct(handle, StateIdx, VarIdx, Operation, Value, ActionTriger);
// In IdleState, preset AgiCycleCounter to 20
STVELPLAN_addCfgAct(stVelPlanHandle, 0, ST_COND_NC, 0, 0, 2, ST_ACT_EQ, 20, ST_ACT_EXIT);
// Decrease AgiCycleCounter by 1 every time enters AgiState1
STVELPLAN_addCfgAct(stVelPlanHandle, 2, ST_COND_NC, 0, 0, 2, ST_ACT_ADD, -1, ST_ACT_ENTR);
// In FillState, set VarIdx3 to 1 to open FillValve
STVELPLAN_addCfgAct(stVelPlanHandle, 1, ST_COND_NC, 0, 0, 3, ST_ACT_EQ, 1, ST_ACT_ENTR);
// In FillState, set VarIdx3 to 0 to close FillValve when FillSensor = 1
STVELPLAN_addCfgAct(stVelPlanHandle, 1, ST_COND_NC, 0, 0, 3, ST_ACT_EQ, 0, ST_ACT_EXIT);
// In DrainState, set VarIdx4 to 1 to open DrainValve
STVELPLAN_addCfgAct(stVelPlanHandle, 4, ST_COND_NC, 0, 0, 4, ST_ACT_EQ, 1, ST_ACT_ENTR);
// In DrainState, set VarIdx4 to 0 to close DrainValve when DrainSensor = 1
STVELPLAN_addCfgAct(stVelPlanHandle, 4, ST_COND_NC, 0, 0, 4, ST_ACT_EQ, 0, ST_ACT_EXIT);

// If there was an error during the configuration, force Plan into the Halt State
if(STVELPLAN_getErrorID(stVelPlanHandle) != FALSE) {
  // Configure FSM: Ts, VelMax, AccMax, DecMax, JrkMax, LoopENB
  STVELPLAN_setCfg(stVelPlanHandle, _IQ24(ST_SPEED_SAMPLE_TIME),
                                    velMax, accMax, jrkMax, FALSE);
// Configure halt state: VelEnd, AccMax, JrkMax, Timer
STVELPLAN_setCfgHaltState(stVelPlanHandle, 0, accMax, jrkMax, 1000L);}
```

13.5.5 Call SpinTAC Velocity Plan

This can be done in the main loop. This code example includes the code required to interface with the fill and drain valves and sensors. It will also update the water level as part of the washing machine simulation.

```
if(gMotorVars.VelPlanRun == TRUE) {
  STVELPLAN_setEnable(stVelPlanHandle, TRUE);
}
// Run SpinTAC Velocity Plan
STVELPLAN_run(stVelPlanHandle);

// Update sensor values for SpinTAC Plan
// Get values for washer valve components
STVELPLAN _getVar(stVelPlanHandle, 3, &gVelPanVar[3]); // Get value of FillVale
STVELPLAN _getVar(stVelPlanHandle, 3, &gVelPanVar[4]);  // Get value of DrainValve
if(gVelPanVar[3] == TRUE) {
  // if FillValve is open, increase water level
  gWaterLevel += 1;
}
else if(gVelPanVar[4] == TRUE) {
  // if DrainValve is open, decrease water level
  gWaterLevel -= 1;
}
if(gWaterLevel >= WASHER_MAX_WATER_LEVEL) {
  // if water level is greater than maximum, set fill sensor to true
  gWaterLevel = WASHER_MAX_WATER_LEVEL;
  gVelPanVar[0] = TRUE;
}
else {
  // if water level is less than maximum, set FillSensor to false
  gVelPanVar[0] = FALSE;
}
if(gWaterLevel <= WASHER_MIN_WATER_LEVEL) {
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
  // if water level is at the minimum & set DrainSensor to true
  gWaterLevel = WASHER_MIN_WATER_LEVEL;
  gVelPanVar[1] = TRUE;
}
else {
  // if water level is greater than minimum, set DrainSensor to false
  gVelPanVar[1] = FALSE;
}
// Set values for washer sensor components
STVELPLAN _getVar(stVelPlanHandle, 0, gVelPanVar[0]); // Set value for FillSensor
STVELPLAN _getVar(stVelPlanHandle, 1, gVelPanVar[1]); // Set value for DrainSensor

if(STVELPLAN_getStatus(stVelPlanHandle) != ST_PLAN_IDLE) {
  // Send the profile configuration to SpinTAC Move
  gMotorVars.SpeedRef_krpm = _IQmpy(STVELPLAN_getVelocitySetpoint(stVelPlanHandle),
                             _IQ24(ST_SPEED_KRPM_PER_PU));
  gMotorVars.MaxAccel_krpmps = _IQmpy(STVELPLAN_getAccelerationLimit(stVelPlanHandle),
                               _IQ24(ST_SPEED_KRPM_PER_PU));
  gMotorVars.MaxJrk_krpmps2 = _IQ20mpy(STVELPLAN_getJerkLimit(stVelPlanHandle),
                              _IQ20(ST_SPEED_KRPM_PER_PU));
}
else {
  STVELPLAN_setEnable(stVelPlanHandle, FALSE);
}
```

13.5.6 Call SpinTAC Velocity Plan Tick

This should be done in the main ISR. This function needs to be called at the proper decimation rate for this component. The decimation rate is established by ST_ISR_TICKS_PER_SPINTAC_TICK; for more information, see Section 4.7.1.4.

```
// Run SpinTAC Velocity Plan Tick
STVELPLAN_runTick(stVelPlanHandle);
```

13.5.7 Update SpinTAC Velocity Plan with SpinTAC Velocity Move Status

This should be done in the main ISR. This function needs to be called when SpinTAC Velocity Move has completed a profile. This is to alert SpinTAC Velocity Plan that we have reached the goal speed that it provided to SpinTAC Velocity Move. This should be placed after the function call for SpinTAC Velocity Move.

```
// Update Plan when the profile is completed
if(STVELMOVE_getDone(stVelMoveHandle) != FALSE) {
STVELPLAN_setUnitProfDone(stVelPlanHandle, TRUE);
}
else {
  STVELPLAN_setUnitProfDone(stVelPlanHandle, FALSE);
}
```

13.6 Troubleshooting SpinTAC Velocity Plan

13.6.1 ERR_ID

ERR_ID provides an error code for users to identify the specific SpinTAC Velocity Plan function that caused the error. A list of ERR_IDs defined in SpinTAC Velocity Plan is shown in Table 13-4.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 13-4. SpinTAC Velocity Plan ERR_ID

ERR_ID	Plan Function
3000	STVELPLAN_addCfgCond
3001	STVELPLAN_delCfgCond
3002	STVELPLAN_setCfgCond
3003	STVELPLAN_getCfgCond
3004	STVELPLAN_addCfgTran
3005	STVELPLAN_delCfgTran
3006	STVELPLAN_setCfgTran
3007	STVELPLAN_getCfgTran
3008	STVELPLAN_addCfgAct
3009	STVELPLAN_delCfgAct
3010	STVELPLAN_setCfgAct
3011	STVELPLAN_getCfgAct
3012	STVELPLAN_addCfgVar
3013	STVELPLAN_delCfgVar
3014	STVELPLAN_setCfgVar
3015	STVELPLAN_getCfgVar
3016	STVELPLAN_addCfgState
3017	STVELPLAN_delCfgState
3018	STVELPLAN_setCfgState
3019	STVELPLAN_setVar
3020	STVELPLAN_getVar
3021	STVELPLAN_setCfg
3022	STVELPLAN_setCfgHaltState
3023	STVELPLAN_setCfgArray
3024	STVELPLAN_addCfgVarCond
3025	STVELPLAN_delCfgVarCond
3026	STVELPLAN_setCfgVarCond
3027	STVELPLAN_getCfgVarCond
4001	STVELPLAN_run (Invalid SpinTAC license. Use the chip with valid license for SpinTAC.)
4003	STVELPLAN_run (Invalid ROM version. Use a chip with a valid ROM version or use the SpinTAC library that is compatible with the current ROM version.)

13.6.2 Configuration Errors

The configuration errors are reported via the CfgError structure included in the main SpinTAC Velocity Plan structure. This structure contains elements that store additional information about the error. The elements are described below:

- CfgError.ERR_idx: Identifies the instance of configured element at which the error occurred.
- CfgError.ERR_code: Identifies the specific error condition that caused the error.

The ERR_code for a specific condition remains the same for all Plan functions. A list of ERR_codes and conditions defined in SpinTAC Velocity Plan is shown in Table 13-5.

Table 13-5. SpinTAC Velocity Plan ERR_code

ERR_code	Description	Solution
1	SpinTAC Plan is running	Place SpinTAC Plan into the idle status prior to running the configuration.
2	Maximum State number exceeded	The maximum number of States has been configured.
3	Maximum Condition number exceeded	The maximum number of Conditions has been configured.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 13-5. SpinTAC Velocity Plan ERR_code (continued)

ERR_code	Description	Solution
4	Maximum Transition number exceeded	The maximum number of Transitions has been configured.
5	Maximum Action number exceeded	The maximum number of Actions has been configured.
6	Maximum Variable number exceeded	The maximum number of Variables has been configured.
7	Invalid sample time value	Set sample time, cfg.T_sec, within (0, 0.01].
8	Invalid VelMax value	Choose VelMax within (0, 1].
9	Invalid AccMax value	Choose AccMax within [0.001, 120].
10	Invalid JrkMax value	Choose JrkMax within [0.0005, 2000].
11	Invalid LoopENB value	Choose LoopENB within { false, true }.
12	Invalid VelEnd value	Choose VelEnd within [(0, VelMax].
13	Invalid AccLim value	Choose AccLim within [0.001, AccMax].
14	Invalid JrkLim value	Choose JrkLim within [0.0005, JrkMax].
15	Invalid Timer_tick value	Choose a positive integer value.
16	Invalid State index	The index should be for a configured State index.
17	Invalid Condition index	The index should be for a configured Condition index.
18	Invalid Transition index	The index should be for a configured Transition index.
19	Invalid Action index	The index should be for a configured Action index.
20	Invalid Variable index	The index should be for a configured Variable index.
21	Invalid Variable type	Choose variable type from the values in ST_PlanVar_e.
22	Invalid value of Comparison	Choose comparison from the values in ST_PlanComp_e.
23	Invalid Operation	Choose operation from the values in ST_PlanActOptn_e.
24	Invalid AndOr value	Choose AndOr from the values in ST_PlanCond_e.
25	Improper Variable type	ST_VAR_OUT Variables cannot have a value set to them . ST_VAR_OUT Variables cannot be used in Conditions . ST_VAR_IN Variables cannot be used in Actions.
26	Improper values in Comparison	Value1 should be less than or equal to Value2.
27	Improper State index	In Transitions FromState cannot be equal to ToState, and these States must be equal to a configured State.
28	Improper Condition index in Transition	In Transitions: CondIdx1 cannot be equal to CondIdx2, and these Conditions must be equal to a configured Condition
29	Improper EnterExit value	Choose EnterExit from the values in ST_PlanActTrgr_e
30	Improper AndOr during Variable deletion	The AndOr value conflicts with the value of VarIdx. When deleting a Variable, it causes a configuration error in a Transition.
31	Cannot delete Variable as an Action depends on it	Remove Variable from Action configuration before deleting the Variable.
37	Plan Configuration array declared is too small for plan elements	Remove an Element from the configuration or declare a larger configuration array.
38	Cannot delete a State as a Transition depends on it	Remove State from Transition configuration before deleting the State.
39	Cannot delete a State as an Action depends on it	Remove State from Action configuration before deleting the State.
40	Improper values for variable comparison	Variable comparison conditions cannot have comparison enum greater than ST_COMP_ELW.
41	Cannot compare a variable to itself	Ensure that the variable indexes passed to the function are different and valid.
42	Cannot get a variable based Condition from the index of a value based type of Condition	Pass an index that is known to contain a variable based Condition.
43	Cannot delete a Condition as a Transition depends on it	Remove Condition from Transition configuration before deleting the Condition.

Copyright © 2013–2019, Texas Instruments Incorporated

13.7 Software Configuration for SpinTAC Position Plan

Configuring SpinTAC Position Plan requires seven steps. Lab 13d – Motion Sequence Real World Example: Vending Machine is an example project that implements the steps required to use SpinTAC Position Plan. The header file spintac_position.h, included in MotorWare, allows you to quickly include the SpinTAC components in your project.

13.7.1 Include the Header File

This should be done with the rest of the module header file includes. In the Lab 13d example project this file is included in the spintac_position.h header file. For your project, this step can be completed by including spintac_position.h.

```
#include "sw/modules/spintac/src/32b/spintac_pos_plan.h"
```

13.7.2 Define the Size of the Configuration Array

This should be done with the rest of the MACRO defines. In the Lab 13d example project this is done at the top of the main source file. Typically it is a best practice to use enumerations to define and label the states in your Plan. This allows you to easily size the configuration array to meet your application requirements. In the Lab 13d example project this step is done for you. Sizing the configuration array for your motion sequence is covered in Section 13.4.2.

```
#define ST_POSPLAN_CFG_ARRAY_DWORDS ((ST_POS_PLAN_ACT_DWORDS     * 4) + \
                                     (ST_POS_PLAN_COND_DWORDS    * 6) + \
                                     (ST_POS_PLAN_VAR_DWORDS     * 6) + \
                                     (ST_POS_PLAN_TRAN_DWORDS    * 6) + \
                                     (ST_POS_PLAN_STATE_DWORDS   * 5))
```

13.7.3 Declare the Global Structure

This should be done with the global variable declarations in the main source file. In the Lab 13d project this structure is included in the ST_Obj structure that is declared as part of the spintac_position.h header file.

```
ST_Obj       st_obj;    // The SpinTAC Object
ST_Handle stHandle;  // The SpinTAC Handle
_iq gVendFwdButton = 0; // Button to advance the displayed item
_iq gVendSelectButton = 0; // Button to vend the displayed item
uint16_t gVendInventory[4] = {VEND_INITIAL_INVENTORY, VEND_INITIAL_INVENTORY,
                              VEND_INITIAL_INVENTORY, VEND_INITIAL_INVENTORY};
VEND_State_e gVendAvailableItem = VEND_ITEM0;  // Current item available to vend
// Configuration array for SpinTAC Position Plan
uint32_t stPosPlanCfgArray[ST_POSPLAN_CFG_ARRAY_DWORDS];
```

This example is if you do not wish to use the ST_Obj structure that is declared in the spintac_position.h header file.

```
ST_PosPlan_t      stPosPlan;       //The SpinTAC Position Plan Object
ST_POSPLAN_Handle stPosPlanHandle; //The SpinTAC Position Plan Handle
_iq gVendFwdButton = 0; // Button to advance the displayed item
_iq gVendSelectButton = 0; // Button to vend the displayed item
uint16_t gVendInventory[4] = {VEND_INITIAL_INVENTORY, VEND_INITIAL_INVENTORY,
                              VEND_INITIAL_INVENTORY, VEND_INITIAL_INVENTORY};
VEND_State_e gVendAvailableItem = VEND_ITEM0;  // Current item available to vend
// Configuration array for SpinTAC Position Plan
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
uint32_t stPosPlanCfgArray[ST_POSPLAN_CFG_ARRAY_DWORDS];
```

13.7.4 Initialize the Configuration Variables

This should be done in the main function of the project ahead of the forever loop. This will load all of the default values into SpinTAC Position Plan. This step can be completed by running the function ST_init that is declared in the spintac_position.h header file and by running the function ST_setupPosPlan that is declared in the main.c. If you do not wish to use this function, the code example below can be used to configure the SpinTAC Position Plan component. This example loads the vending machine profile that is discussed in Section 13.4.6. For more information about the SpinTAC Plan API, see Figure 3-19.

```
// Pass the configuration array pointer into SpinTAC Velocity Plan
STPOSPLAN_setCfgArray(stPosPlanHandle, &stPosPlanCfgArray[0],
                                        sizeof(stPosPlanCfgArray), 4, 6, 6, 6, 5);

// Establish the Velocity, Acceleration, Deceleration, and Jerk Maximums
velMax = STPOSMOVE_getVelocityMaximum(stPosMoveHandle);
accMax = STPOSMOVE_getAccelerationMaximum(stPosMoveHandle);
decMax = STPOSMOVE_getDecelerationMaximum(stPosMoveHandle);
jrkMax = STPOSMOVE_getJerkMaximum(stPosMoveHandle);

// Establish the Velocity, Acceleration, Deceleration, and Jerk Limits
velLim = _IQ24(0.1 * ST_SPEED_PU_PER_KRPM);
accLim = _IQ24(0.5 * ST_SPEED_PU_PER_KRPM);
decLim = _IQ24(0.5 * ST_SPEED_PU_PER_KRPM);
jrkLim = _IQ24(1.0 * ST_SPEED_PU_PER_KRPM);

// Configure SpinTAC Velocity Plan: Sample Time, VelMax, AccMax, DecMax, JrkMax, LoopENB
STPOSPLAN_setCfg(stPosPlanHandle, _IQ24(ST_SAMPLE_TIME), velMax, accMax, decMax, jrkMax, false);
// Configure halt state: PosStepInt, PosStepFrac, VelMax, AccMax, JrkMax, Timer
STPOSPLAN_setCfgHaltState(stPosPlanHandle, 0, 0, velMax, accMax, jrkMax, 1000L);

//Example: STPOSPLAN_addCfgState(handle, PosStepInt[MRev], PosStepFrac[MRev],
StateTimer[ticks]);
STPOSPLAN_addCfgState(stPosPlanHandle, 0, 0,            200L); // StateIdx0: Init
STPOSPLAN_addCfgState(stPosPlanHandle, 0, _IQ24(0.25),  200L); // StateIdx1: Item0
STPOSPLAN_addCfgState(stPosPlanHandle, 0, _IQ24(0.25),  200L); // StateIdx2: Item1
STPOSPLAN_addCfgState(stPosPlanHandle, 0, _IQ24(0.25),  200L); // StateIdx2: Item2
STPOSPLAN_addCfgState(stPosPlanHandle, 0, _IQ24(0.25),  200L); // StateIdx2: Item3

//Example: STPOSPLAN_addCfgVar(handle, VarType,      InitialValue);
STPOSPLAN_addCfgVar(stPosPlanHandle,   ST_VAR_INOUT, 0);  // VarIdx0: FwdButton
STPOSPLAN_addCfgVar(stPosPlanHandle,   ST_VAR_IN,    10); // VarIdx1: Item0Inv
STPOSPLAN_addCfgVar(stPosPlanHandle,   ST_VAR_IN,    10); // VarIdx2: Item1Inv
STPOSPLAN_addCfgVar(stPosPlanHandle,   ST_VAR_IN,    10); // VarIdx3: Item2Inv
STPOSPLAN_addCfgVar(stPosPlanHandle,   ST_VAR_IN,    10); // VarIdx4: Item3Inv
STPOSPLAN_addCfgVar(stPosPlanHandle,   ST_VAR_IN,    40); // VarIdx5: TotalInv

//Example: STPOSPLAN_addCfgCond(handle, VarIdx, Comparison,   Value1, Value2)
// CondIdx0: Fwd Button Pressed
STPOSPLAN_addCfgCond(stPosPlanHandle,   0,      ST_COMP_EQ,   1,      0);
// CondIdx1: Item0 Empty
STPOSPLAN_addCfgCond(stPosPlanHandle,   1,      ST_COMP_ELW,  0,      0);
// CondIdx2: Item1 Empty
STPOSPLAN_addCfgCond(stPosPlanHandle,   2,      ST_COMP_ELW,  0,      0);
// CondIdx3: Item2 Empty
STPOSPLAN_addCfgCond(stPosPlanHandle,   3,      ST_COMP_ELW,  0,      0);
// CondIdx4: Item3 Empty
STPOSPLAN_addCfgCond(stPosPlanHandle,   4,      ST_COMP_ELW,  0,      0);
// CondIdx5: TotalInv Empty
STPOSPLAN_addCfgCond(stPosPlanHandle,   5,      ST_COMP_ELW,  0,      0);

//Example: STPOSPLAN_addCfgTran(handle, FromState, ToState, CondOption, CondIdx1, CondIdx2,
VelLim[pups], AccLim[pups2], DecLim[pups2], JrkLim[pups3]);
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
// NOTE: The deceleration limit must be set between the following bounds [acceleration limit,
10*acceleration limit]
// From Init to Item1
STPOSPLAN_addCfgTran(stPosPlanHandle, 0, 2, ST_COND_OR, 0, 2, velLim, accLim, decLim, jrkLim);
// From Item3 to Init
STPOSPLAN_addCfgTran(stPosPlanHandle, 1, 0, ST_COND_FC, 5, 0, velLim, accLim, decLim,
jrkLim);
// From Item0 to Item1
STPOSPLAN_addCfgTran(stPosPlanHandle, 1, 2, ST_COND_OR, 0, 1, velLim, accLim, decLim,
jrkLim);
// From Item1 to Item2
STPOSPLAN_addCfgTran(stPosPlanHandle, 2, 3, ST_COND_OR, 0, 2, velLim, accLim, decLim,
jrkLim);
// From Item2 to Item3
STPOSPLAN_addCfgTran(stPosPlanHandle, 3, 4, ST_COND_OR, 0, 3, velLim, accLim, decLim,
jrkLim);
// From Item3 to Item0
STPOSPLAN_addCfgTran(stPosPlanHandle, 4, 1, ST_COND_OR, 0, 4, velLim, accLim, decLim,
jrkLim);

//Example: STPOSPLAN_addCfgAct(handle, StateIdx, VarIdx, Operation, Value, ActionTriger);
// In Item1, clear Fwd Button
STPOSPLAN_addCfgAct(stPosPlanHandle, 1, ST_COND_NC, 0, 0, 0, ST_ACT_ENTR);
// In Item2, clear Fwd Button
STPOSPLAN_addCfgAct(stPosPlanHandle, 2, ST_COND_NC, 0, 0, 0, ST_ACT_ENTR);
// In Item3, clear Fwd Button
STPOSPLAN_addCfgAct(stPosPlanHandle, 3, ST_COND_NC, 0, 0, 0, ST_ACT_ENTR);
// In Item4, clear Fwd Button
STPOSPLAN_addCfgAct(stPosPlanHandle, 4, ST_COND_NC, 0, 0, 0, ST_ACT_ENTR);

if(STPOSPLAN_getErrorID(stPosPlanHandle) != false) {
  // Configure FSM: Ts, VelMax, AccMax, DecMax, JrkMax, LoopENB
  STPOSPLAN_setCfg(stPosPlanHandle, _IQ24(ST_SAMPLE_TIME), velMax, accMax, decMax, jrkMax,
false);
  // Configure halt state: PosStepInt[MRev], PosStepFrac[MRev], VelMax, AccMax, JrkMax, Timer
  STPOSPLAN_setCfgHaltState(stPosPlanHandle, 0, 0, velMax, accMax, jrkMax, 1000L);
}
```

13.7.5 Call SpinTAC Position Plan

This can be done in the main loop. This code example includes the code required to interface with the fill and drain valves and sensors. It will also update the item inventory as part of the vending machine simulation.

```
// SpinTAC Position Plan
if(gPosPlanRunFlag == ST_PLAN_STOP
                                && gMotorVars.SpinTAC.PosPlanRun == ST_PLAN_START) {
  if(STPOSMOVE_getDone(stPosMoveHandle) == true) {
    if(STPOSPLAN_getErrorID(stPosPlanHandle) != false) {
      STPOSPLAN_setEnable(stPosPlanHandle, false);
      STPOSPLAN_setReset(stPosPlanHandle, true);
      gMotorVars.SpinTAC.PosPlanRun = gPosPlanRunFlag;
    }
    else {
      STPOSPLAN_setEnable(stPosPlanHandle, true);
      STPOSPLAN_setReset(stPosPlanHandle, false);
      gPosPlanRunFlag = gMotorVars.SpinTAC.PosPlanRun;
    }
  }
}
if(gMotorVars.SpinTAC.PosPlanRun == ST_PLAN_STOP) {
  STPOSPLAN_setReset(stPosPlanHandle, true);
  gPosPlanRunFlag = gMotorVars.SpinTAC.PosPlanRun;
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
}
if(gPosPlanRunFlag == ST_PLAN_START
                                  && gMotorVars.SpinTAC.PosPlanRun == ST_PLAN_PAUSE) {
  STPOSPLAN_setEnable(stPosPlanHandle, false);
  gPosPlanRunFlag = gMotorVars.SpinTAC.PosPlanRun;
}
if(gPosPlanRunFlag == ST_PLAN_PAUSE
                                  && gMotorVars.SpinTAC.PosPlanRun == ST_PLAN_START) {
  STPOSPLAN_setEnable(stPosPlanHandle, true);
  gPosPlanRunFlag = gMotorVars.SpinTAC.PosPlanRun;
}

// if we have selected an item from the machine
if(gVendSelectButton == 1) {
  if(STPOSPLAN_getStatus(stPosPlanHandle) != ST_PLAN_IDLE) {
    // decrease our inventory
    gVendInventory[gVendAvailableItem - 1]--;
  }
  // toggle the select button off
  gVendSelectButton = 0;
}

// Update variables passed into Plan
STPOSPLAN_setVar(stPosPlanHandle, VEND_Fwd, gVendFwdButton);
STPOSPLAN_setVar(stPosPlanHandle, VEND_Item0Inv, gVendInventory[VEND_ITEM0 - 1]);
STPOSPLAN_setVar(stPosPlanHandle, VEND_Item1Inv, gVendInventory[VEND_ITEM1 - 1]);
STPOSPLAN_setVar(stPosPlanHandle, VEND_Item2Inv, gVendInventory[VEND_ITEM2 - 1]);
STPOSPLAN_setVar(stPosPlanHandle, VEND_Item3Inv, gVendInventory[VEND_ITEM3 - 1]);
STPOSPLAN_setVar(stPosPlanHandle, VEND_TotalInv,
      gVendInventory[0] + gVendInventory[1] + gVendInventory[2] + gVendInventory[3]);

// Run SpinTAC Position Plan
STPOSPLAN_run(stPosPlanHandle);

// display the selected item
if(STPOSPLAN_getCurrentState(stPosPlanHandle) > 0) {
  gVendAvailableItem = (VEND_State_e)STPOSPLAN_getCurrentState(stPosPlanHandle);
}
else {
  gVendAvailableItem = VEND_ITEM0;
}

// Update variables passed out of Plan
if(STPOSPLAN_getFsmState(stPosPlanHandle) == ST_FSM_STATE_STAY) {
  STPOSPLAN_getVar(stPosPlanHandle, VEND_Fwd, &gVendFwdButton);
}

if(STPOSPLAN_getStatus(stPosPlanHandle) != ST_PLAN_IDLE) {
  // Send the profile configuration to SpinTAC Position Profile Generator
  STPOSPLAN_getPositionStep_mrev(stPosPlanHandle,
        (_iq24 *)&gMotorVars.PosStepInt_MRev, (_iq24 *)&gMotorVars.PosStepFrac_MRev);
  gMotorVars.MaxVel_krpm = _IQmpy(STPOSPLAN_getVelocityLimit(stPosPlanHandle),
                                                    _IQ24(ST_SPEED_KRPM_PER_PU));
  gMotorVars.MaxAccel_krpmps =_IQmpy(STPOSPLAN_getAccelerationLimit(stPosPlanHandle),
                                                    _IQ24(ST_SPEED_KRPM_PER_PU));
  gMotorVars.MaxDecel_krpmps =_IQmpy(STPOSPLAN_getDecelerationLimit(stPosPlanHandle),
                                                    _IQ24(ST_SPEED_KRPM_PER_PU));
  gMotorVars.MaxJrk_krpmps2 = _IQ20mpy(STPOSPLAN_getJerkLimit(stPosPlanHandle),
                                                    _IQ20(ST_SPEED_KRPM_PER_PU));
}
else {
  if(gPosPlanRunFlag == ST_PLAN_START
                                  && gMotorVars.SpinTAC.PosPlanRun == ST_PLAN_START)
  {
    gMotorVars.SpinTAC.PosPlanRun = ST_PLAN_STOP;
    gPosPlanRunFlag = gMotorVars.SpinTAC.PosPlanRun;
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
  }
}
```

13.7.6 Call SpinTAC Position Plan Tick

This should be done in the main ISR. This function needs to be called at the proper decimation rate for this component. The decimation rate is established by ISR_TICKS_PER_SPINTAC_TICK; for more information, see Section 4.7.1.4.

```
// Run SpinTAC Position Plan Tick
STPOSPLAN_runTick(stPosPlanHandle);
```

13.7.7 Update SpinTAC Position Plan with SpinTAC Position Move Status

This should be done in the main ISR. This function needs to be called when SpinTAC Position Move has completed a profile. This is to alert SpinTAC Position Plan that we have reached the goal position that it provided to SpinTAC Position Plan. This should be placed after the function call for SpinTAC Position Move.

```
// Update SpinTAC Position Plan when the profile is completed
if(STPOSMOVE_getDone(stPosMoveHandle) != false) {
  STPOSPLAN_setUnitProfDone(stPosPlanHandle, true);
}
else {
  STPOSPLAN_setUnitProfDone(stPosPlanHandle, false);
}
```

13.8 Troubleshooting SpinTAC Position Plan

13.8.1 ERR_ID

ERR_ID provides an error code for users to identify the specific SpinTAC Position Plan function that caused the error. A list of ERR_IDs defined in SpinTAC Position Plan is shown in Table 13-6.

Table 13-6. SpinTAC Position Plan ERR_ID

ERR_code	Plan Function
3000	Configuration error in STPOSPLAN_addCfgCond
3001	Configuration error in STPOSPLAN_delCfgCond
3002	Configuration error in STPOSPLAN_setCfgCond
3003	Configuration error in STPOSPLAN_getCfgCond
3004	Configuration error in STPOSPLAN_addCfgTran
3005	Configuration error in STPOSPLAN_delCfgTran
3006	Configuration error in STPOSPLAN_setCfgTran
3007	Configuration error in STPOSPLAN_getCfgTran
3008	Configuration error in STPOSPLAN_addCfgAct
3009	Configuration error in STPOSPLAN_delCfgAct
3010	Configuration error in STPOSPLAN_setCfgAct
3011	Configuration error in STPOSPLAN_getCfgAct
3012	Configuration error in STPOSPLAN_addCfgVar
3013	Configuration error in STPOSPLAN_delCfgVar

Copyright © 2013–2019, Texas Instruments Incorporated

Table 13-6. SpinTAC Position Plan ERR_ID (continued)

ERR_code	Plan Function
3014	Configuration error in STPOSPLAN_setCfgVar
3015	Configuration error in STPOSPLAN_getCfgVar
3016	Configuration error in STPOSPLAN_addCfgState
3017	Configuration error in STPOSPLAN_delCfgState
3018	Configuration error in STPOSPLAN_setCfgState
3019	Configuration error in STPOSPLAN_setVar
3020	Configuration error in STPOSPLAN_getVar
3021	Configuration error in STPOSPLAN_setCfg
3022	Configuration error in STPOSPLAN_setCfgHaltState
3023	Configuration error in STPOSPLAN_setCfgArray
3024	Configuration error in STPOSPLAN_addCfgVarCond
3025	Configuration error in STPOSPLAN_delCfgVarCond
3026	Configuration error in STPOSPLAN_setCfgVarCond
3027	Configuration error in STPOSPLAN_getCfgVarCond
4001	STPOSPLAN_run (Invalid SpinTAC license. Use the chip with valid license for SpinTAC.)
4003	STPOSPLAN_run (Invalid ROM version. Use a chip with a valid ROM version or use the SpinTAC library that is compatible with the current ROM version.)

13.8.2 Configuration Errors

The configuration errors are reported via the CfgError structure included in the main SpinTAC Position Plan structure. This structure contains elements that store additional information about the error. The elements are described below:

- CfgError.ERR_idx: Identifies the instance of configured element at which the error occurred.
- CfgError.ERR_code: Identifies the specific error condition that caused the error.

The ERR_code for a specific condition remains the same for all Plan functions. A list of ERR_codes and conditions defined in SpinTAC Position Plan is shown in Table 13-7.

Table 13-7. SpinTAC Position Plan ERR_code

ERR_code	Description	Solution
1	SpinTAC Plan is running	Place SpinTAC Plan into the idle status prior to running the configuration.
2	Maximum State number exceeded	The maximum number of States has been configured.
3	Maximum Condition number exceeded	The maximum number of Conditions has been configured.
4	Maximum Transition number exceeded	The maximum number of Transitions has been configured.
5	Maximum Action number exceeded	The maximum number of Actions has been configured.
6	Maximum Variable number exceeded	The maximum number of Variables has been configured.
7	Invalid sample time value	Set sample time, cfg.T_sec, within (0, 0.01].
8	Invalid VelMax value	Choose VelMax within (0, 1].
9	Invalid AccMax value	Choose AccMax within [0.001, 120].
10	Invalid JrkMax value	Choose JrkMax within [0.0005, 2000].
11	Invalid LoopENB value	Choose LoopENB within { false, true }.
12	Invalid VelEnd value	Choose VelEnd within (0, VelMax].
13	Invalid AccLim value	Choose AccLim within [0.001, AccMax].
14	Invalid JrkLim value	Choose JrkLim within [0.0005, JrkMax].
15	Invalid Timer_tick value	Choose a positive integer value.
16	Invalid State index	The index should be for a configured State index.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 13-7. SpinTAC Position Plan ERR_code (continued)

ERR_code	Description	Solution
17	Invalid Condition index	The index should be for a configured Condition index.
18	Invalid Transition index	The index should be for a configured Transition index.
19	Invalid Action index	The index should be for a configured Action index.
20	Invalid Variable index	The index should be for a configured Variable index.
21	Invalid Variable type	Choose variable type from the values in ST_PlanVar_e.
22	Invalid value of Comparison	Choose comparison from the values in ST_PlanComp_e.
23	Invalid Operation	Choose operation from the values in ST_PlanActOptn_e.
24	Invalid AndOr value	Choose AndOr from the values in ST_PlanCond_e.
25	Improper Variable type	ST_VAR_OUT Variables cannot have a value set to them. ST_VAR_OUT Variables cannot be used in Conditions. ST_VAR_IN Variables cannot be used in Actions.
26	Improper values in Comparison	Value1 should be less than or equal to Value2.
27	Improper State index	In Transitions FromState cannot be equal to ToState, and these States must be equal to a configured State.
28	Improper Condition index in Transition	In Transitions: CondIdx1 cannot be equal to CondIdx2, and these Conditions must be equal to a configured Condition.
29	Improper EnterExit value	Choose EnterExit from the values in ST_PlanActTrgr_e.
30	Improper AndOr during Variable deletion	The AndOr value conflicts with the value of VarIdx. When deleting a Variable, it causes a configuration error in a Transition.
31	Cannot delete Variable as an Action depends on it	Remove Variable from Action configuration before deleting the Variable.
32	Invalid VelLim value	Choose VelLim within (0, VelMax).
33	Invalid DecLim value	Choose DecLim within [0.001, DecMax] and keep the ratio DecLim/AccLim within [0.1, 10].
34	Invalid DecMax value	Choose DecLim within [0.001, 120] and keep the ratio DecLim/AccLim within [0.1, 10].
35	Invalid PosStepInt_mrev or PosStepFrac_mrev for HaltState	Choose PosStepInt_mrev within [-2, 2] and PosStepFrac_mrev within (-1, 1).
36	Invalid PosStepInt_mrev or PosStepFrac_mrev for State	Choose PosStepInt_mrev within [-2147483647, 2147483647] and PosStepFrac_mrev within (-1, 1).
37	Plan Configuration array declared is too small for plan elements	Remove an Element from the configuration or declare a larger configuration array.
38	Cannot delete a State as a Transition depends on it	Remove State from Transition configuration before deleting the State.
39	Cannot delete a State as an Action depends on it	Remove State from Action configuration before deleting the State.
40	Improper values for variable comparison	Variable comparison conditions cannot have comparison enum greater than ST_COMP_ELW.
41	Cannot compare a variable to itself	Ensure that the variable indexes passed to the function are different and valid.
42	Cannot get a variable based Condition from the index of a value based type of Condition	Pass an index that is known to contain a variable based Condition.
43	Cannot delete a Condition as a Transition depends on it	Remove Condition from Transition configuration before deleting the Condition.
44	The first State must have a PosStep of 0 [MRev]	Configure the first State to have PosStepInt and PosStepFrac both equal to 0.

Copyright © 2013–2019, Texas Instruments Incorporated

13.9 Conclusion

InstaSPIN-MOTION provides an easy way to design trajectory changes and motion sequences. This allows you to quickly implement your application. It allows you to get your motion sequence designed and tested very quickly. SpinTAC Velocity Move generates constraint-based, time-optimal, repeatable trajectory profiles. These profiles are triggered by SpinTAC Velocity Plan which implements the application motion sequence.

Copyright © 2013–2019, Texas Instruments Incorporated

Managing Full Load at Startup, Low-Speed and Speed Reversal

The FAST algorithm included in InstaSPIN-FOC has certain aspects that need to be considered while operating at low speeds with mechanical load attached to the motor shaft. This document describes several aspects of typical motor control problems while operating sensorless control during low speed operation. Comprehensive lab results for the performance of FAST are documented in the *TMS320F2806xF InstaSPIN-FOC Technical Reference Manual* (SPRUHI9), the *TMS320F2805xF InstaSPIN-FOC Technical Reference Manual* (SPRUHW0), and the *TMS320F2802xF InstaSPIN-FOC Technical Reference Manual* (SPRUHP4).

The motor control scenarios to be covered in this section are:

1. Low-speed operation with full load
2. Speed reversal with full load
3. Motor startup with full load
4. Rapid acceleration from standstill with full load
5. Overloading and motor overheating.

The system used in these experiments includes the following components:

- Texas Instruments C2000 Processor: TMS320F28069F with InstaSPIN-FOC Version 1.6
- Texas Instruments Inverter Model: TMDSHVMTRPFCKIT Version 1.1
- IPM Motor with the following characteristics:
  - Rated Voltage = 300 V
  - Rated Current = 4 A
  - Motor Maximum Current = 6 A
  - Stator Resistance (Rs) = 2.6 Ω
  - Stator Quadrature Inductance (Ls_q) = 13.5 mH
  - Stator Direct Inductance (Ls_d) = 11.5 mH
  - Rotor Flux (ψ) = 0.5 V/Hz = 0.08 Wb
  - Rated Torque = 1.9 N.m
- Magtrol Dynamometer Model: HD-715-8N
- Magtrol Dynamometer Controller Model: DSP6001

The motor under test is coupled to a Dynamometer as shown in Figure 14-1.

Copyright © 2013–2019, Texas Instruments Incorporated

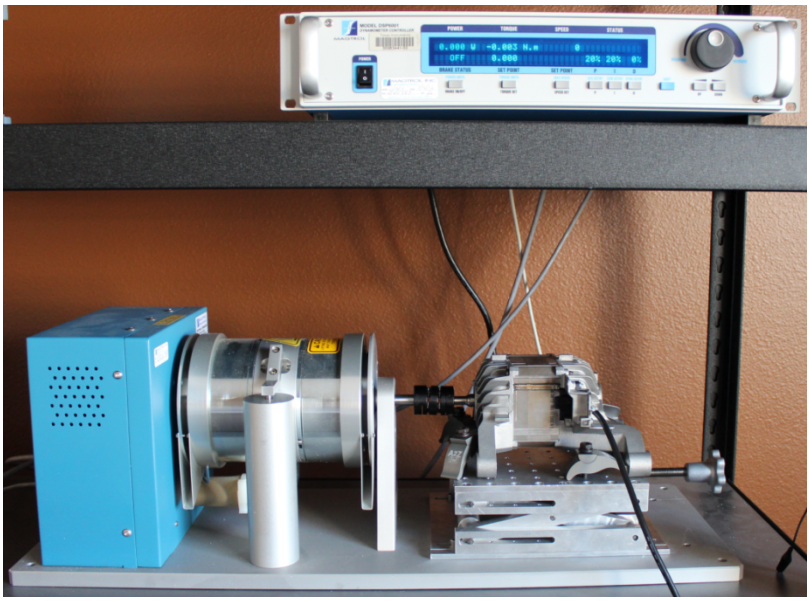


Figure 14-1. Photograph of Test Fixture

Topic **Page**

14.1 Low-Speed Operation with Full Load 484
14.2 Speed Reversal with Full Load 495
14.3 Motor Startup with Full Load 502
14.4 Rapid Acceleration from Standstill With Full Load 511
14.5 Overloading and Motor Overheating 521
14.6 InstaSPIN-MOTION and Low-Speed Considerations 526

Copyright © 2013–2019, Texas Instruments Incorporated

14.1 Low-Speed Operation with Full Load

In order to operate at low speeds, first let us discuss the low speed operation with no mechanical load. The lowest speed we can control with no load can be seen visually in the motor rotation as well as in the current waveform. For example, running the motor under test down to 30 RPM, or 2 Hz, is possible with a speed variation of 30 ±3 RPM. However, in order to get to a speed fairly constant we must have the speed controller responsive enough to compensate against speed variations.

14.1.1 Low Speed with Full Load Considerations

In summary the following considerations are needed when operating at low speeds:

- Enable offsets recalibration; described in Section 14.1.1.1.
- Enable stator Rs recalibration; described in Section 14.1.1.2.
- Disable forced angle; described in Section 14.1.1.3.
- Tune speed controller to avoid motor stall; described in Section 14.1.1.4.
- Tune voltage feedback circuit; described in Section 14.1.1.5.

14.1.1.1 Enable Offsets Recalibration

For low speed and full load performance to be as expected, the offsets for the currents and voltages need to be well calibrated. In order to do this, the offset recalibration in the controller object should be enabled prior to running the motor in closed loop. The following code example enables the offsets recalibration.

```
// enable automatic calculation of bias values
CTRL_setFlag_enableOffset(ctrlHandle, TRUE);
```

Keep in mind that the enable-function of the offsets recalibration must be called prior to enabling the controller, which is done by calling the CTRL_setFlag_enableCtrl(ctrlHandle, TRUE) function.

Offsets recalibration is critical for low speed performance.

Offsets recalibration is critical for low speed performance, especially the voltage offsets, since in the low speed range, the voltage feedback from the motor tend to be very small values, so having well calibrated offsets is a must.

14.1.1.2 Enable Stator Rs Recalibration

Another important factor to consider when operating in low speeds is the motor model represented in software. The stator resistance plays a significant role in the accuracy of the estimated parameters. Hence Rs recalibration should be enabled prior to enabling the controller. The following code example enabled Rs recalibration.

```
// enable Rs recalibration
EST_setFlag_enableRsRecalc(obj->estHandle,TRUE);
```

14.1.1.3 Disable Forced Angle

When running in low speeds, forced-angle must be turned off in order to allow the estimator to converge to the estimated values with no intervention of an external angle being forced. In order to disable forced-angle the following code example can be used:

Forced-angle must be turned off to allow estimator to converge.

```
// disable the forced angle
EST_setFlag_enableForceAngle(obj->estHandle,FALSE);
```


Copyright © 2013–2019, Texas Instruments Incorporated

14.1.1.4 Tune Speed Controller to Avoid Motor Stall

In order to continue driving the motor at low speeds, without stalling the motor, user must tune the speed controller so that the reaction of the speed controller is fast enough to prevent the motor from completely stopping during torque transients. The following functions can be used to update the speed controller inside of InstaSPIN:

```
_iq New_Kp_spd;
_iq New_Ki_spd;

// set the kp and ki speed controller gains
CTRL_setKp(handle,CTRL_Type_PID_spd, New_Kp_spd);
CTRL_setKi(handle,CTRL_Type_PID_spd, New_Ki_spd);
```

14.1.1.5 Tune Voltage Feedback Circuit

Low speed operation assumes that the back EMF voltage coming back from the motor will be very small. Since the FAST algorithm requires the phase voltages as inputs to the algorithm, it is encouraged to have the maximum number of ADC bits per volt. For example, if the maximum input voltage to the system is 400 V, then it is highly recommended to have a voltage feedback that only goes up to that voltage, so when low voltage is present in the motor's back EMF (i.e. during low speed operation) the maximum number of bits are present in the ADC converter output so that the estimation has more information to support the motor model.

In order to illustrate the importance of having the maximum number of bits per volt, consider the TMDSHVMTRPFCKIT Version 1.1 for the voltage feedback as shown in Figure 14-2.

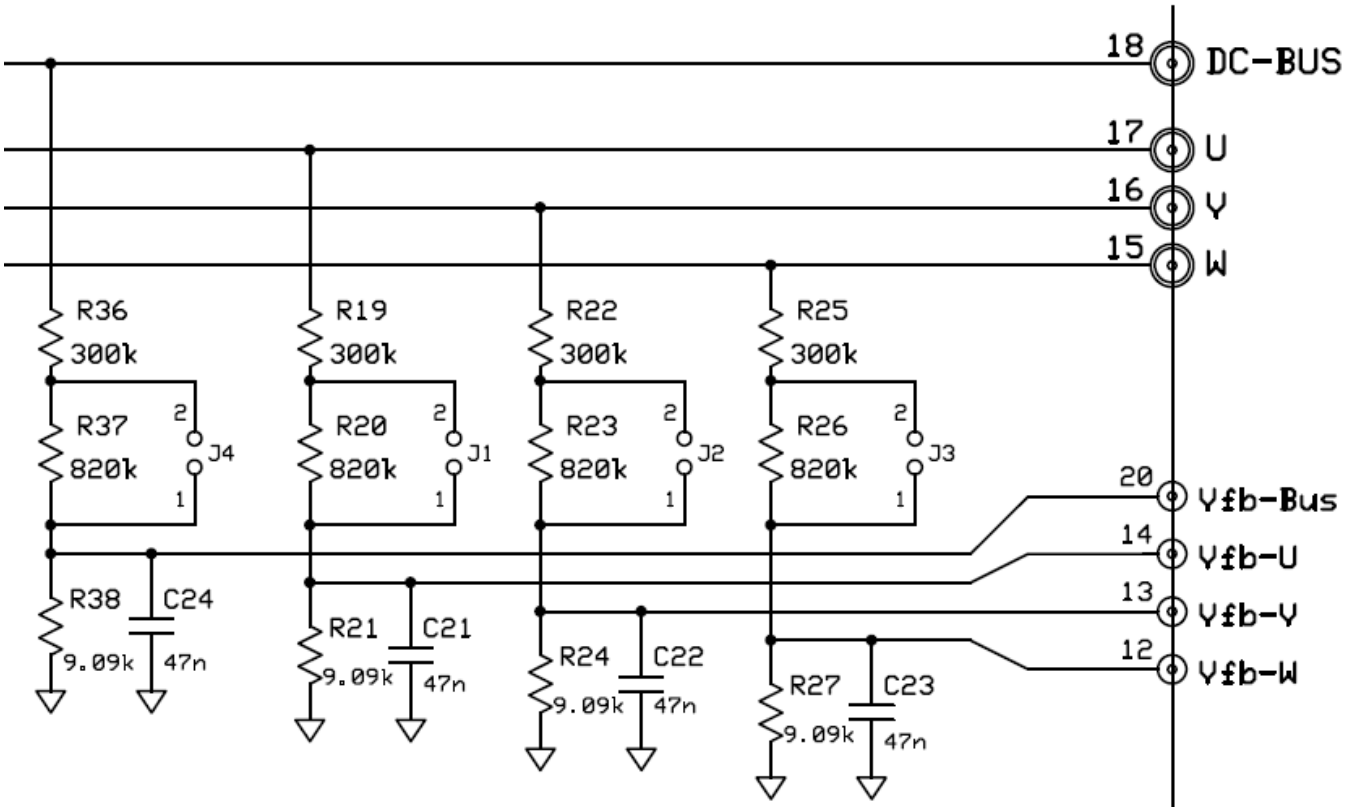


Figure 14-2. Voltage Feedback Circuit of High-Voltage Kit (TMDSHVMTRPFCKIT)

Copyright © 2013–2019, Texas Instruments Incorporated

This circuit with jumpers J1, J2, J3 and J4 open (see section M5 of the PCB), would give us a maximum Vfb-Bus of 3.3 V when DC-BUS is at 409.9 V. This means that this setup provides the best ADC resolution for motors with input of 409.9 V. However if the motor's operating voltage is only 100 V, then the ADC resolution to represent the motor's internal voltage will be very small, in fact, around ¼ of what it could be. If the motor's operating voltage is known to be 100 V, then the above circuit should be changed so that a maximum Vfb-Bus of 3.3 V is present in the ADC pin when DC-BUS is 100 V.

Scale voltage feedback circuit to maximize ADC 3.3V input range.

Changing the voltage feedback value affects two parameters configured in user.h. For example in the above schematic, with J1, J2, J3 and J4 open, the following parameters in user.h are defined:

```
#define USER_ADC_FULL_SCALE_VOLTAGE_V    (409.9)
#define USER_VOLTAGE_FILTER_POLE_Hz    (375.5)
```

This can be derived as follows (J1, J2, J3 and J4 open):

$ADC_IN_{max} = 3.3 \quad V$

$R_1 = 300 \quad k\Omega + 820 \quad k\Omega = 1120 \quad k\Omega$

$R_2 = 9.09 \quad k\Omega$

$$USER_ADC_FULL_SCALE_VOLTAGE_V = \frac{ADC_IN_{max} \times R_1 + R_2}{R_2} = 409.9 \text{ V}$$

$$R_{Parallel} = \frac{R_2 \times R_1}{R_2 + R_1} = 9.017 \text{ k}\Omega$$

$$C = 47 \text{ nF}$$

$$USER_VOLTAGE_FILTER_POLE_Hz = \frac{1}{2 \times \pi \times R_{Parallel} \times C} = 375.5 \text{ Hz}$$

With J1, J2, J3 and J4 shorted, the following parameters in user.h are redefined:

```
#define USER_ADC_FULL_SCALE_VOLTAGE_V    (112.2)
#define USER_VOLTAGE_FILTER_POLE_Hz    (383.8)
```

This can be derived as follows:

$ADC_IN_{max} = 3.3 \quad V$

$R_1 = 300 \quad k\Omega$

$R_2 = 9.09 \quad k\Omega$

$$USER_ADC_FULL_SCALE_VOLTAGE_V = \frac{ADC_IN_{max} \times R_1 + R_2}{R_2} = 112.2 \text{ V} \tag{79}$$

$$R_{Parallel} = \frac{R_2 \times R_1}{R_2 + R_1} = 8.823 \text{ k}\Omega$$

$$C = 47 \text{ nF}$$

$$USER_VOLTAGE_FILTER_POLE_Hz = \frac{1}{2 \times \pi \times R_{Parallel} \times C} = 383.8 \text{ Hz}$$

14.1.2 Low Speed With Full Load Transient Examples

After considering the above requirements, let us look at a few examples of low speed response with the motor under test using a dynamometer.

14.1.2.1 4-Hz, No-Load to Full-Load Transient

Figure 14-3 shows the current waveform under these conditions:

Copyright © 2013–2019, Texas Instruments Incorporated

- Dynamometer = 1.9 N·m (full load)
- Speed Controller = 4Hz (60 RPM ± 1 RPM)

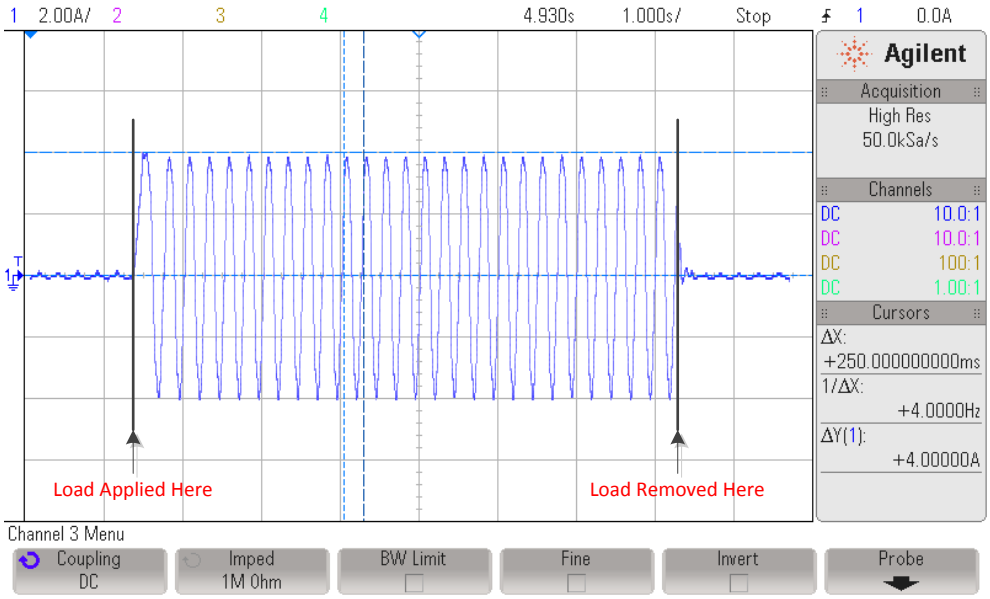


Figure 14-3. 4-Hz, No-Load to Full-Load Transient Plot

A torque transient of the motor's rated torque of 1.9 N·m is applied to the motor shaft, resulting in a current of 4 A. The electrical frequency as seen in the oscilloscope plot is 4 Hz. For a 4-pole pair motor, this frequency results in a speed of 60 ± 1 RPM once it has stabilized. Notice that the time where the load is applied might be different compared with the time of the capture variables. However the conditions of applied torque shown in the scope plot compared to captured variables is identical. The difference in time is due to the fact that the capture current was captured in a different test although having the same parameters.

FAST stands for Flux, Angle, Speed and Torque. Figure 14-4, Figure 14-5, Figure 14-6, and Figure 14-7 show the behavior of the FAST algorithm and how the torque step command affects the FAST output variables.

FAST variables are consistent even with a 100% step- load.

Figure 14-4 is the estimated flux of the motor. It is actually the flux linkage provided by FAST, and it is shown to be fairly constant. The variation of this flux is a result of different aspects such as motor parameter accuracy as well as how well the magnetic circuit of the motor is designed for a particular load.

Copyright © 2013–2019, Texas Instruments Incorporated

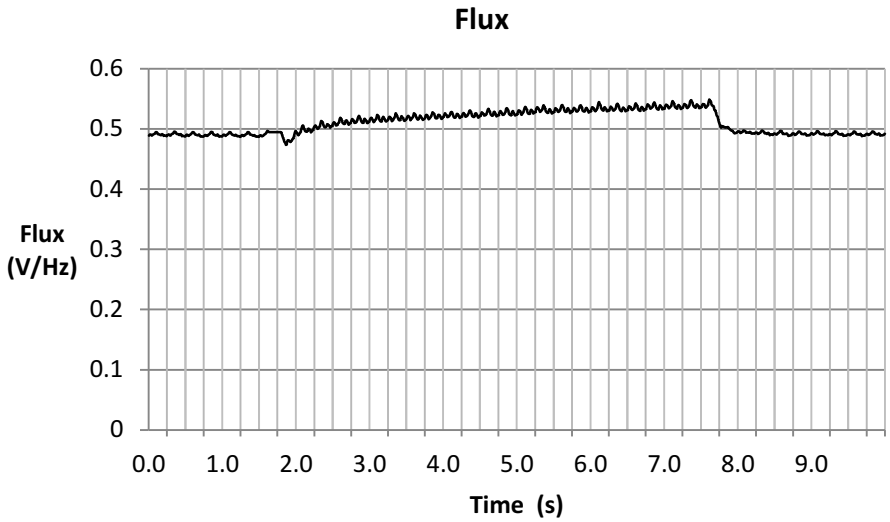


Figure 14-4. Flux Plot

Figure 14-5, Figure 14-6, and Figure 14-7 show the flux angle provided by FAST. As can be seen, the angle is tracked through the increase of motor load, and also the decrease of motor load.

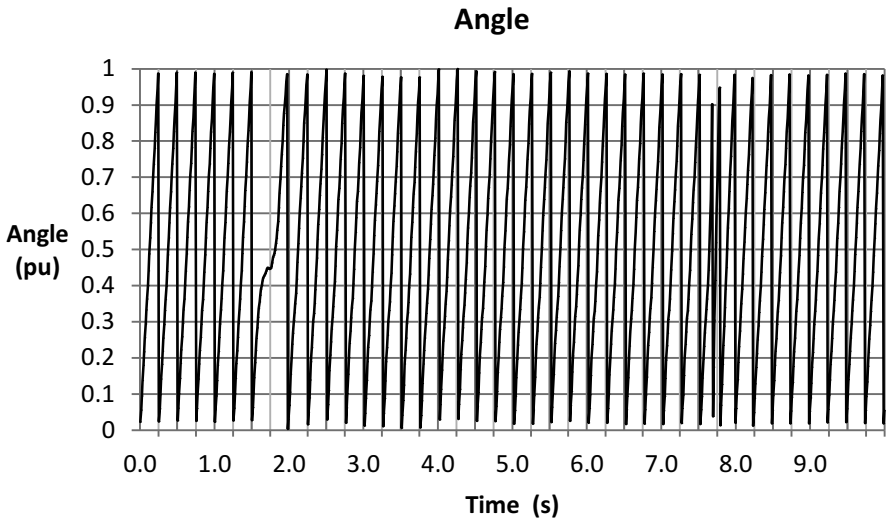


Figure 14-5. Angle Plot

If the angle is zoomed in where the motor is loaded, it can be seen how the rate of change of the angle changes to a very low rate of change, and once the speed controller corrects for this, the rate of change is picked back up to the commanded speed.

Copyright © 2013–2019, Texas Instruments Incorporated

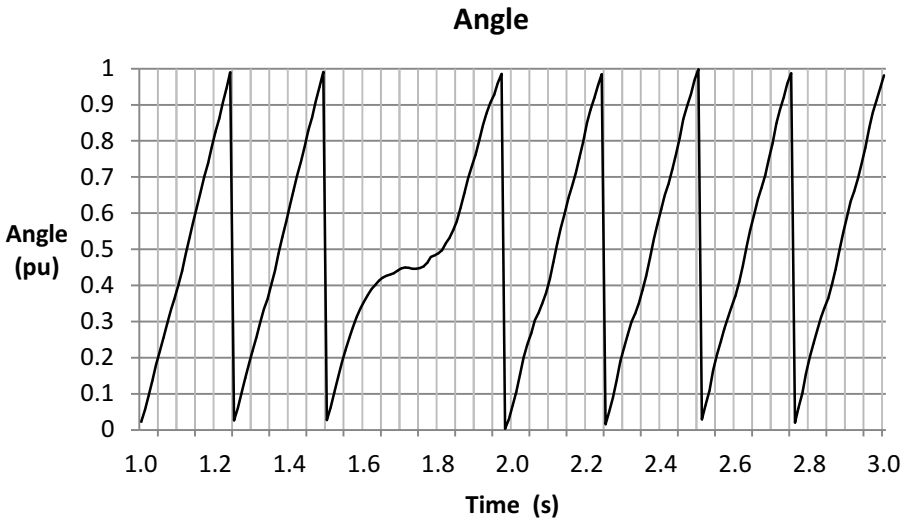


Figure 14-6. Zoom-in on Angle Plot - Motor Loaded

The same behavior can be seen when the load is removed from the motor shaft. The speed is increased due to the torque command provided by the speed controller, and after some time, the speed controller regulates the speed down to the 4 Hz (60 RPM) command.

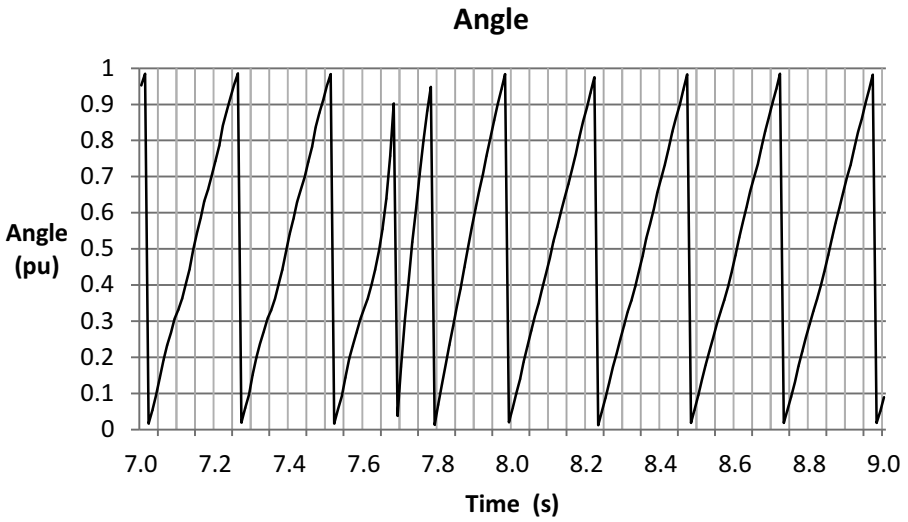


Figure 14-7. Zoom-in on Angle Plot - Load Removed

It is worth mentioning that the purpose of showing the speed variation is not to show the performance of the speed controller. In fact, the speed controller has nothing to do with the FAST estimator. Figure 14-8 shows how the estimator tracks the speed of the motor even when the torque demand stalls the motor for a small period of time.

Copyright © 2013–2019, Texas Instruments Incorporated

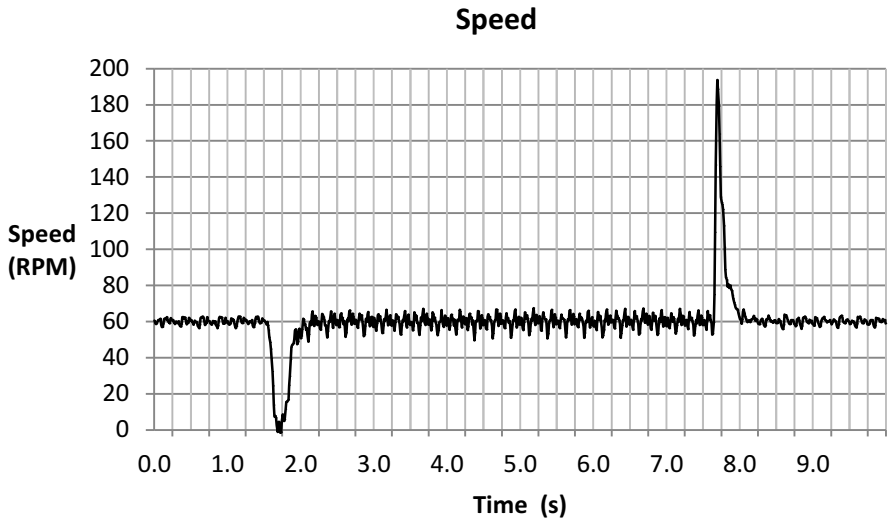


Figure 14-8. Speed Plot

Figure 14-9 shows the torque signal produced by FAST. This torque signal is useful to know the instantaneous torque on the motor shaft, and calculate motor loading without a torque sensor. This high bandwidth signal shows tracking of the torque even when steps are commanded.

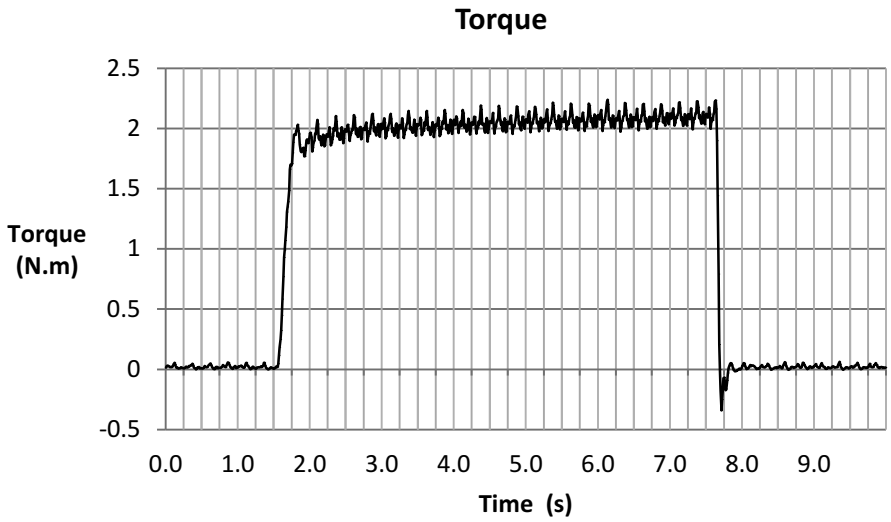


Figure 14-9. Torque Plot

Also, we plot the Iq current waveform to show the field oriented control performance that FAST allows when torque steps are commanded. As can be seen in the current plot for Iq (Figure 14-10), the response to the current demand can follow a step as in the example, where a step load is applied to the shaft. The angle tracking capability of the estimator allows this step response in the Iq controller. You might also notice that the torque curve is not as flat as the current curve. This is due to the variation of the flux linkage seen in the previous flux plot, possibly due to a mismatch on the motor model compared to what the reality of the model is.

Copyright © 2013–2019, Texas Instruments Incorporated

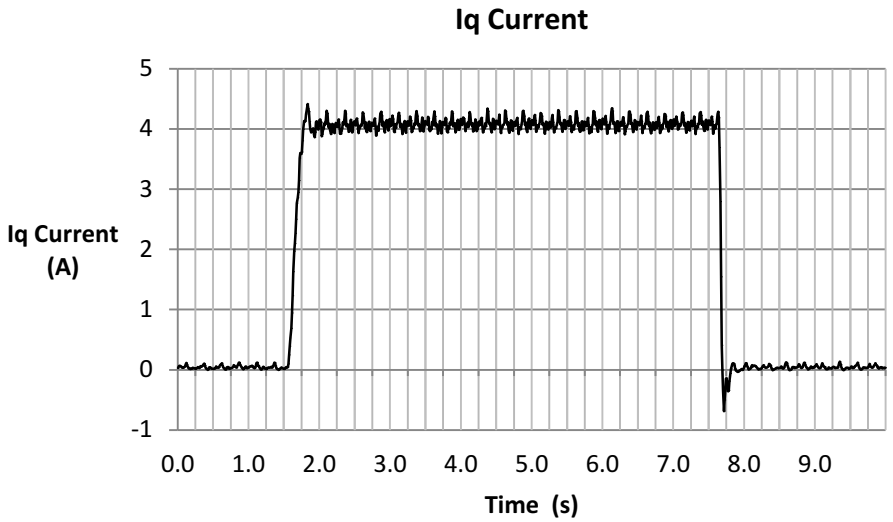


Figure 14-10. Iq Current Plot

In the previous example it can be seen that when the load changes so drastically, the speed of the motor can fall all the way to zero. This response can be improved with the speed controller loop itself, but the point of the test is to show how the variables provided by FAST are consistent and valid even with a 100% step on the load command.

14.1.2.2 2-Hz, No-Load to Full-Load Transient

Figure 14-11 shows the current waveform under these conditions:

- Dynamometer = 1.9 N·m (full load)
- Speed Controller = 2 Hz (30 RPM ± 3 RPM)

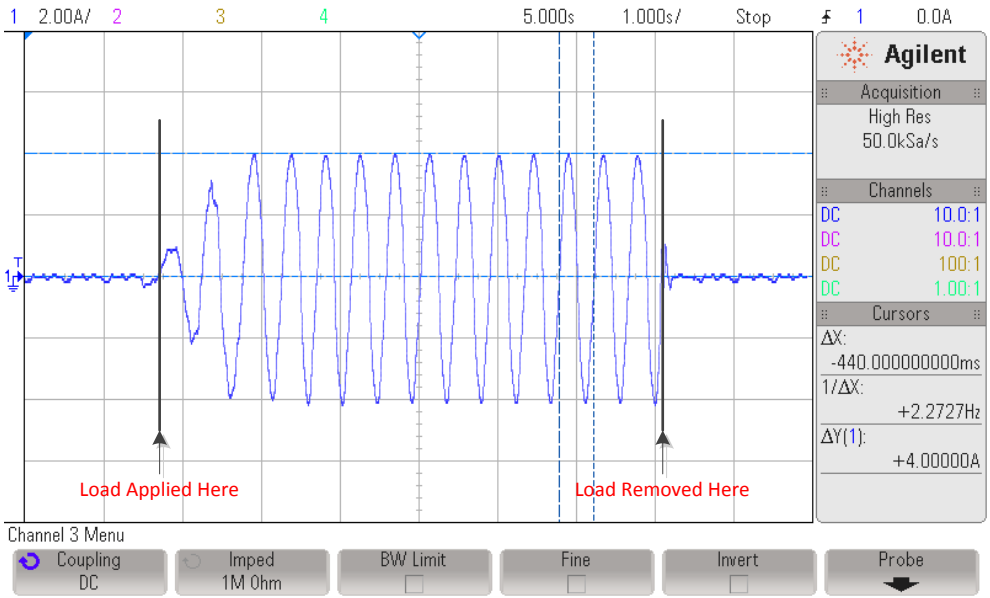


Figure 14-11. 2-Hz, No-Load to Full-Load Transient Plot

A torque transient of the motor's rated torque of 1.9 N·m is applied to the motor shaft, resulting in a current of 4 A. The electrical frequency as seen in the oscilloscope plot is 2.2 Hz, which is about 3 RPM higher than commanded by the speed reference. For a 4-pole pair motor, this frequency results in a speed of 30 ± 3 RPM once it has stabilized.

Copyright © 2013–2019, Texas Instruments Incorporated

The challenge we run into when using hysteresis dynamometers is that the torque production and the detent torque present in the dynamometer shaft produce an instantaneous torque higher than the commanded torque, causing the motor to be stalled from time to time. This is the main reason why we increased the torque command of the dynamometer at a lower rate compared to the previous example, to avoid the dynamometer to produce more torque than commanded when the motor is stalled temporarily.

Figure 14-12, Figure 14-13, Figure 14-14, and Figure 14-15 show the behavior of the FAST algorithm. FAST stands for Flux, Angle, Speed and Torque, and this is how the torque step command affects those variables. The first variable is the flux linkage of the motor.

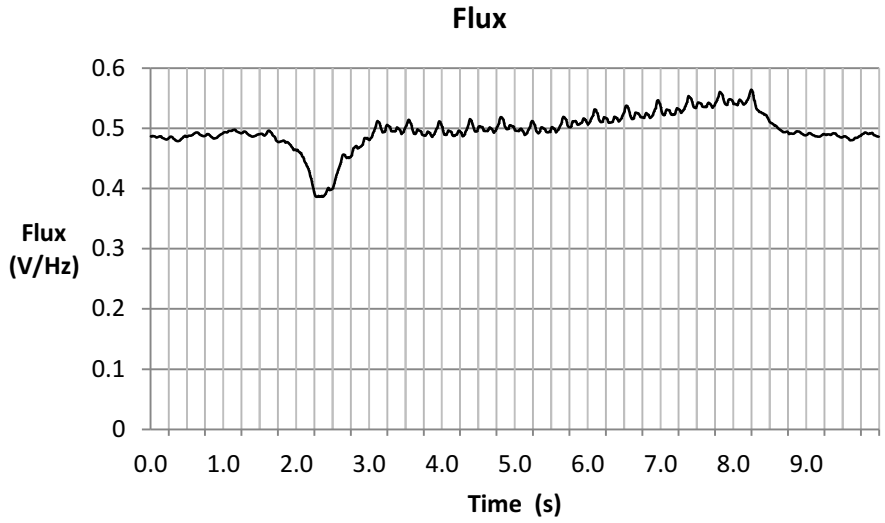


Figure 14-12. Flux Plot

Figure 14-13, Figure 14-14, and Figure 14-15 show the flux angle provided by FAST. As was seen with the previous test, the angle is tracked through the increase of motor load, and also the decrease of motor load.

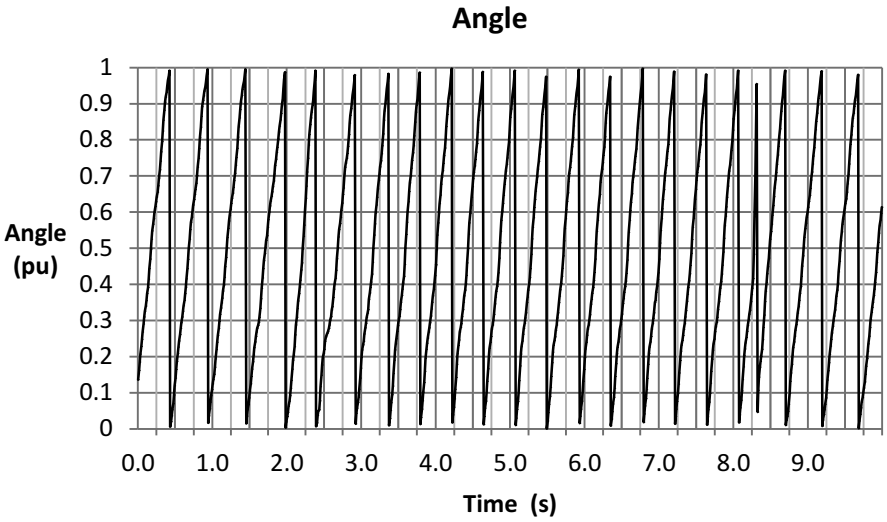


Figure 14-13. Angle Plot

Zooming in the angle plot, we can see transients when the motor is being loaded, and when the load is removed. As we get lower in speed, the quality of the signals, combined with the torque pulsations of the hysteresis dynamometer, makes the angle not look like a perfect saw tooth. Even then, the angle information provides good enough information to run a full FOC control at 2 Hz and a full load transient.

Copyright © 2013–2019, Texas Instruments Incorporated

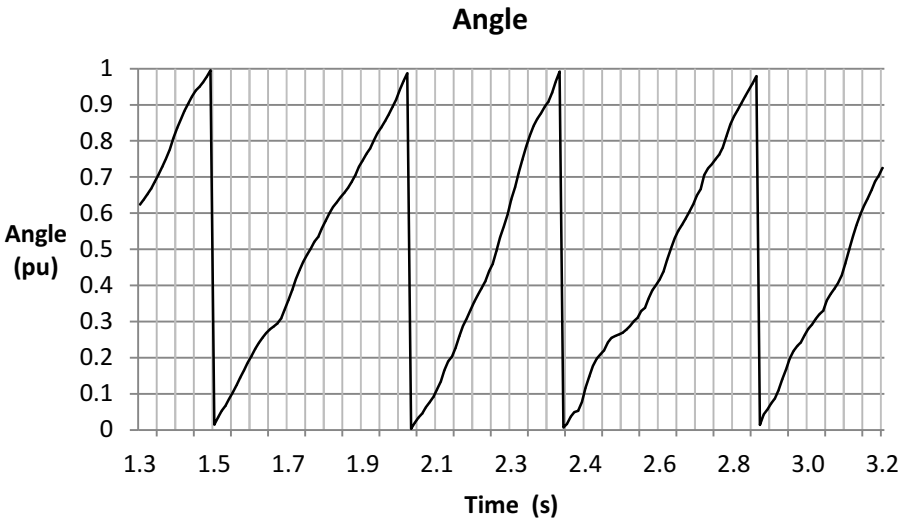


Figure 14-14. Zoom-in on Angle Plot - Increased Motor Load

Zooming in when the load is removed from the shaft, we can see an instantaneous angle tracking.

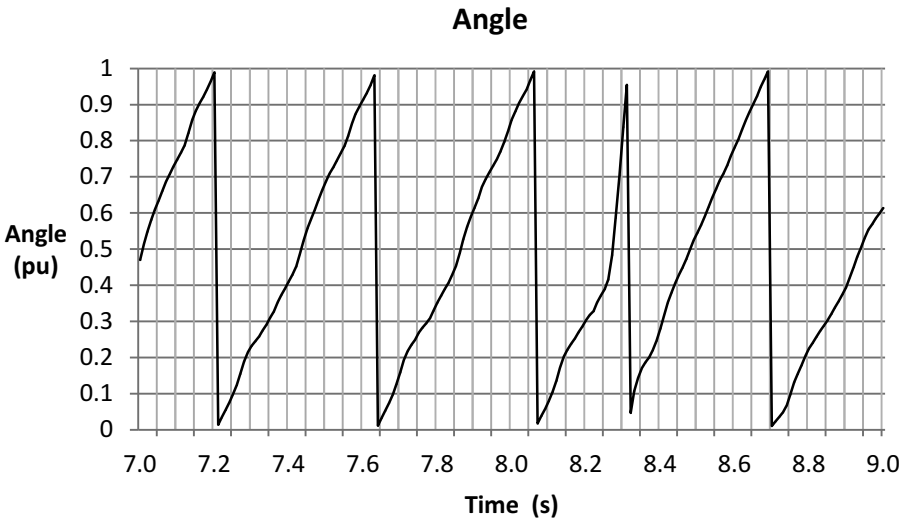


Figure 14-15. Zoom-in on Angle Plot - Decreased Motor Load

The speed plot is shown in Figure 14-16. The target speed is 30 RPM, and we can see higher ripple on the estimated speed compared to 60 RPM. This is due to the pulsating torque present in the hysteresis dynamometer and also, the estimated speed output is instantaneous as opposed to every electrical cycle. So any distortion on the angle ramp will be reflected in a speed oscillation.

FAST variables consistently enable FOC system to apply full torque even with a 100% step-load at low speeds.

Also, when the load is completely removed, which is done by turning off the dynamometer controller, the speed estimation follows the real speed even when there is rapid acceleration, as shown in Figure 14-11.

Copyright © 2013–2019, Texas Instruments Incorporated

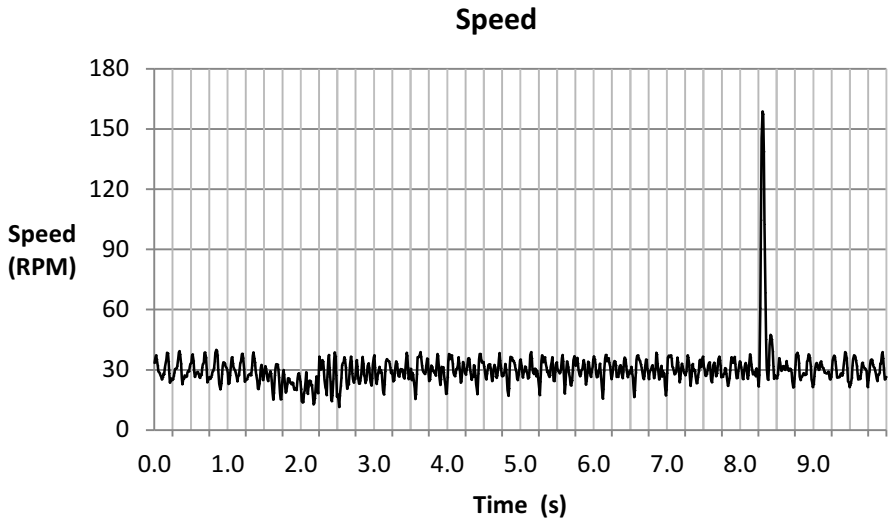


Figure 14-16. Speed Plot

The torque signal is shown in Figure 14-17. Oscillations are due to the low frequency of the estimator, as well as the torque pulsations present in the hysteresis dynamometer at low speeds.

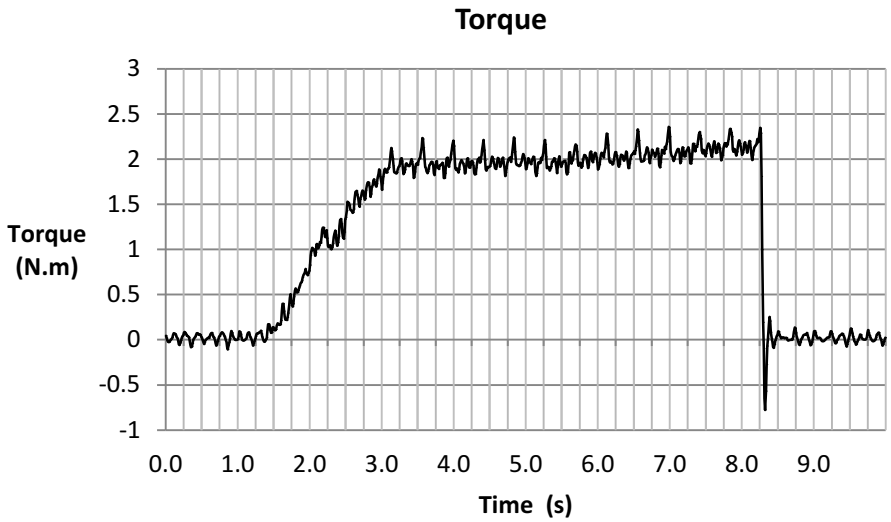


Figure 14-17. Torque Plot

The current controller follows the curve of the commanded torque as can be seen in Figure 14-18, taking the current to the rated 4 A in Iq.

Copyright © 2013–2019, Texas Instruments Incorporated

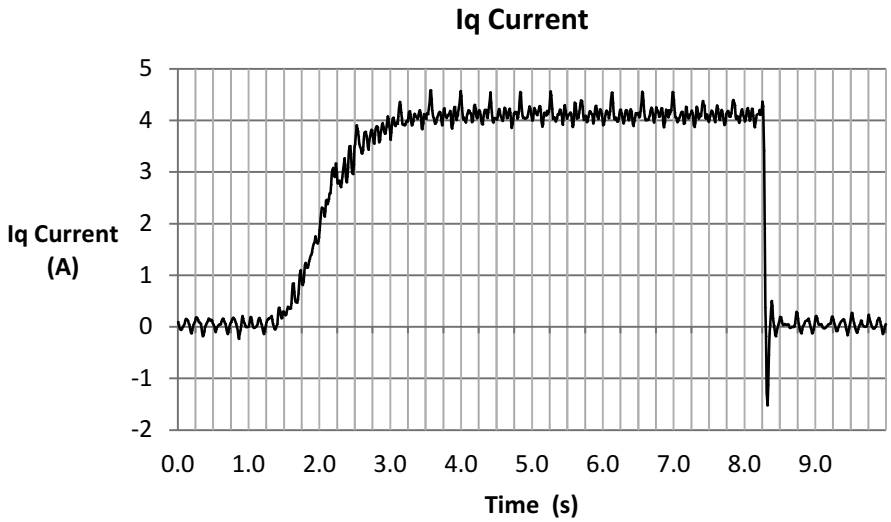


Figure 14-18. Iq Current Plot

14.2 Speed Reversal with Full Load

In order to do a speed reversal, either from high or low positive speed to high or low negative speed, at any acceleration rate, it is important to consider the same points we considered for the torque transient response example.

14.2.1 Low Speed with Full Load Speed Reversal Considerations

These considerations are required for this mode of operation:

- Enable offsets recalibration; described in Section 14.1.1.1.
- Enable stator Rs recalibration; described in Section 14.1.1.2.
- Disable forced angle; described in Section 14.1.1.3.
- Tune speed controller to avoid motor stall; described in Section 14.1.1.4.
- Tune voltage feedback circuit; described in Section 14.1.1.5.

14.2.2 Low Speed with Full Load Speed Reversal Examples

Once the above items are covered, let us look at a few examples of speed reversal response with the motor under test using the dynamometer.

14.2.2.1 From -4 to +4 Hz with Full Load

Figure 14-19 shows the current waveform under these conditions. Notice how the current changes phase indicating a change in direction.

Copyright © 2013–2019, Texas Instruments Incorporated

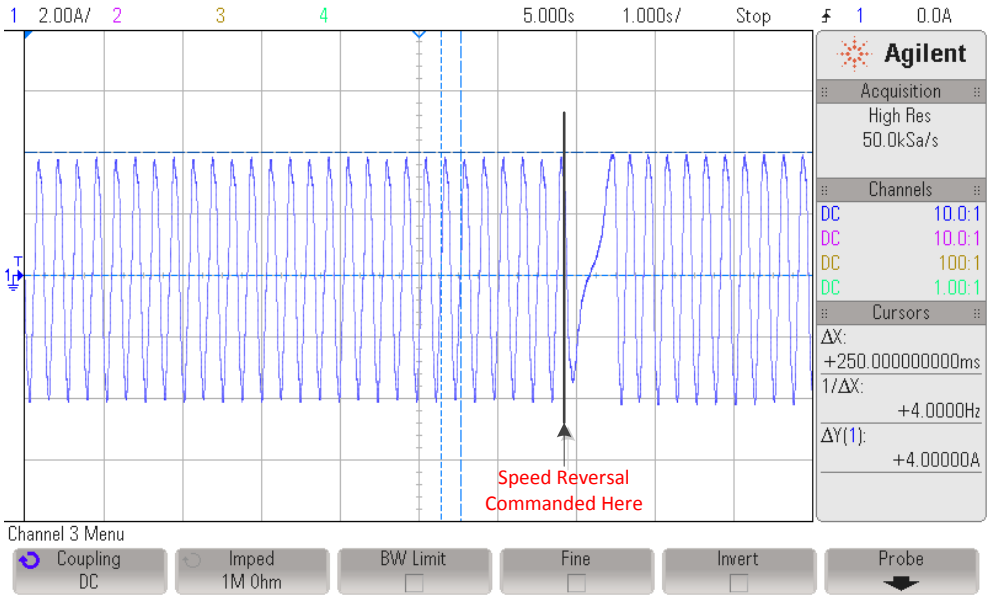


Figure 14-19. -4 to +4 Hz with Full Load Plot

The flux estimation can be seen to have a transient when it changes direction, although it stabilizes after a few seconds. A constantly growing flux can be noticed from the plot shown in Figure 14-20. This can indicate a slight mismatch between the motor model represented in software compared to the real system. The error in flux might be due to inaccuracies of the modeled motor compared to the actual motor, possibly due to overheating or current and voltage sensing tolerances. In cases where the flux is constantly growing, this might indicate that the stator resistance is converging into a new value due to the motor load causing the motor to warm up. It is recommended to try the Rs Online feature of InstaSPIN in such a case. For an example on how to run Rs Online feature, see Chapter 15.

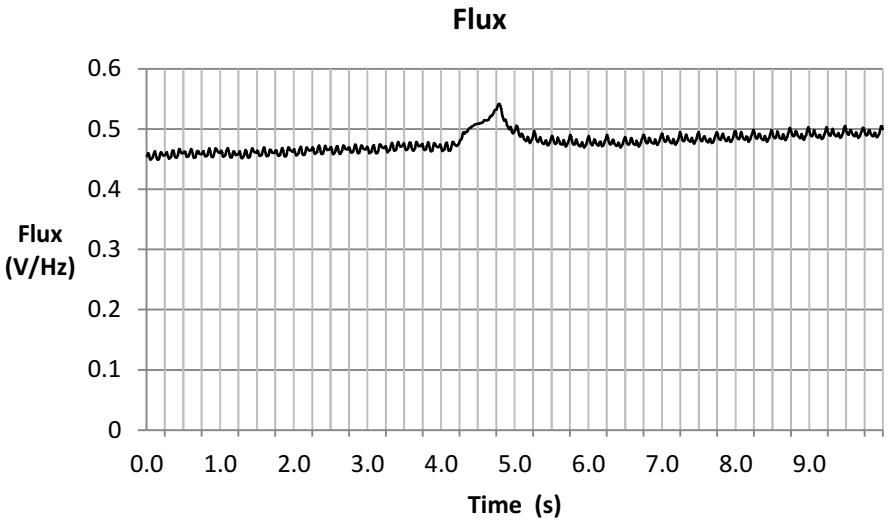


Figure 14-20. Flux Plot

In Figure 14-21, we can see the flux angle changing phases when it is going through zero speed.

Copyright © 2013–2019, Texas Instruments Incorporated

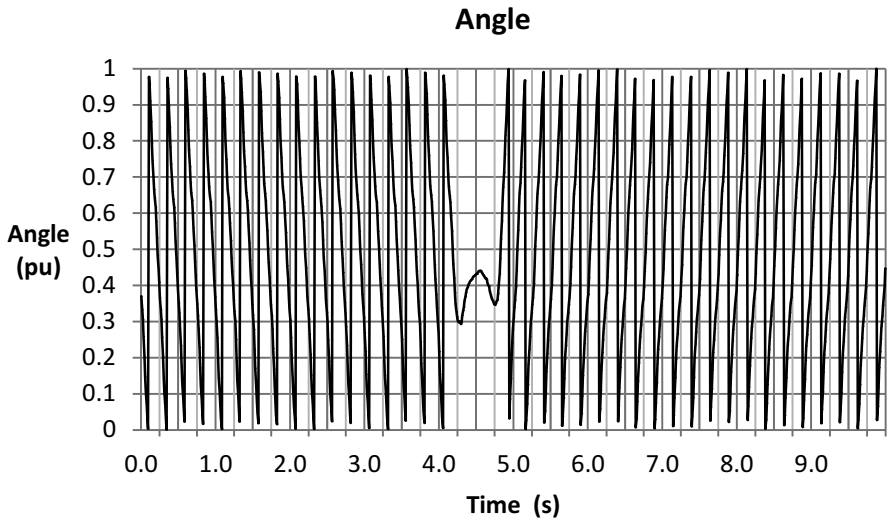


Figure 14-21. Angle Plot

If we zoom-in when the motor is changing direction (Figure 14-22), we can see more clearly how this transition is done. We can actually see that it changes direction twice. This is because at near zero, the algorithm tries to find the direction in which the angle is rotating.

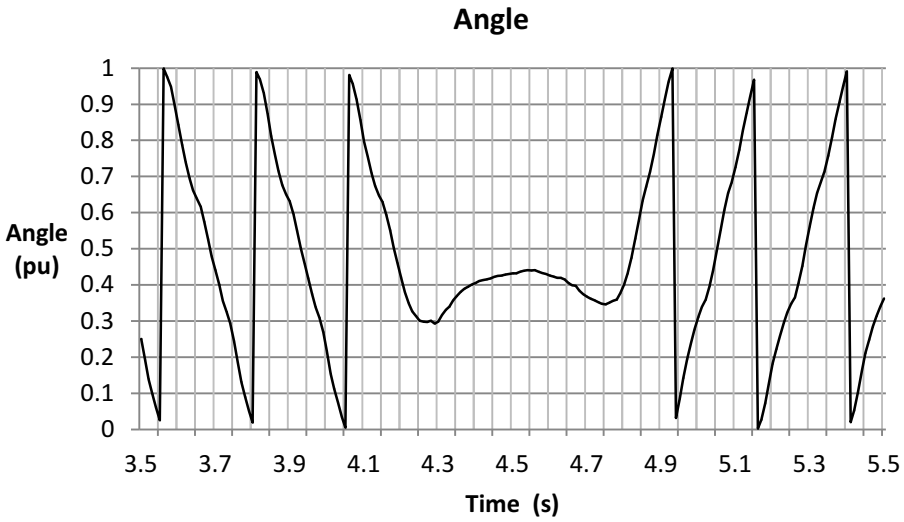


Figure 14-22. Zoom-in on Angle Plot

The estimated speed of the motor also shows how the speed when it crosses zero can have some error in sign (Figure 14-23). This is when it is within ±10 RPM, which translates to ±0.66 Hz.

Copyright © 2013–2019, Texas Instruments Incorporated

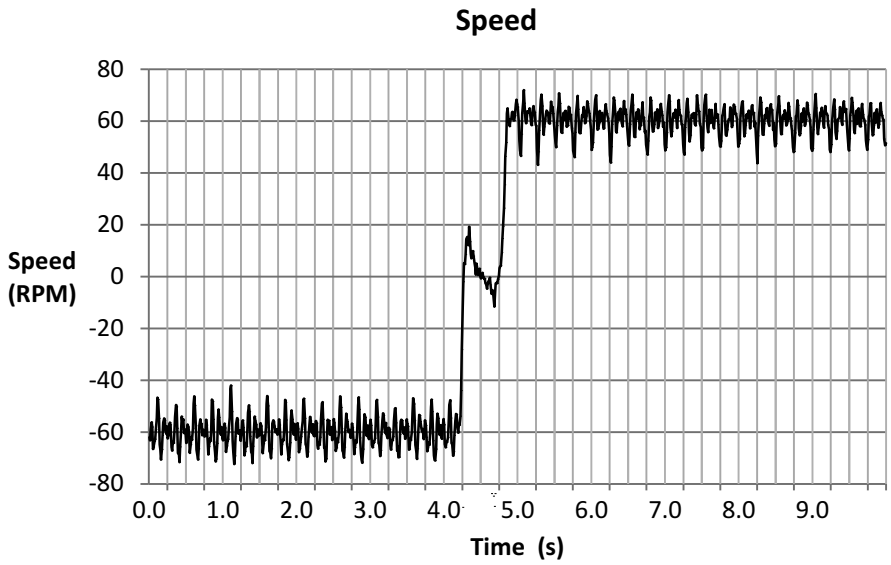


Figure 14-23. Speed Plot

The torque signal coming from FAST can be seen continuously growing from -1.9 Nm to +1.9 Nm (Figure 14-24), with a small overshoot on the positive side. That overshoot might have been the accumulation of current in the hysteresis dynamometer while going through zero speed, which is an expected behavior of these types of dynamometers.

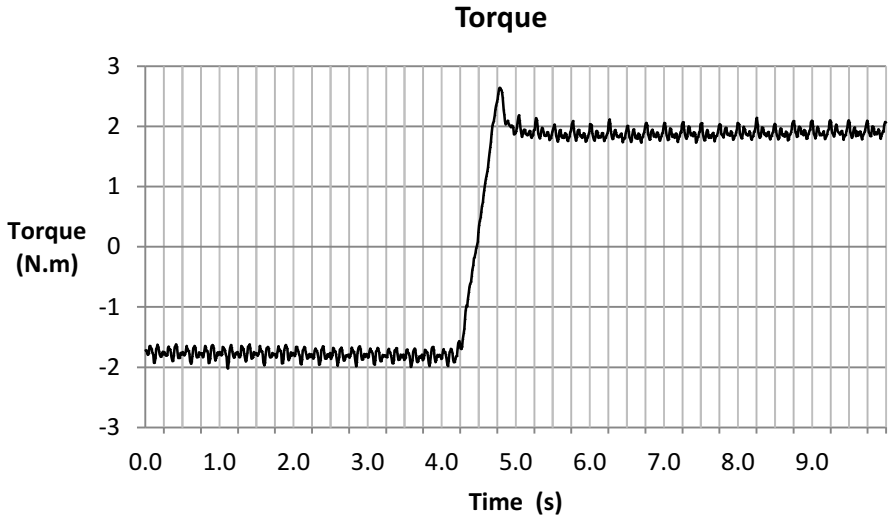


Figure 14-24. Torque Plot

The quadrature current, Iq, is displayed in Figure 14-25. It can be seen how this follows a very similar waveform compared to the estimated torque waveform. Not too much noticeable in this plot, but we can see that the current is flatter than the torque. This is due to a flux estimation converging to a new value after some time driving full load.

Copyright © 2013–2019, Texas Instruments Incorporated

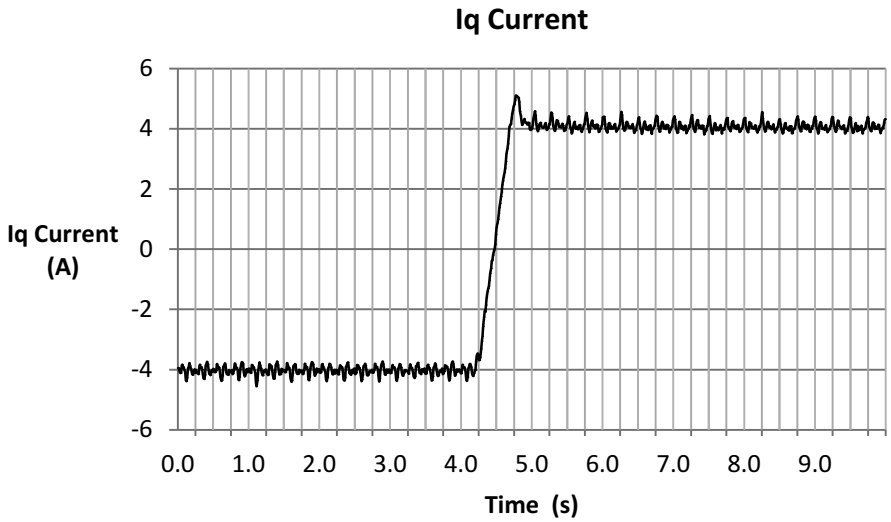


Figure 14-25. Iq Current Plot

14.2.2.2 From -2 to +2 Hz with Full Load

Figure 14-26 shows the current waveform under these conditions. Notice how the current changes phase indicating a change in direction.

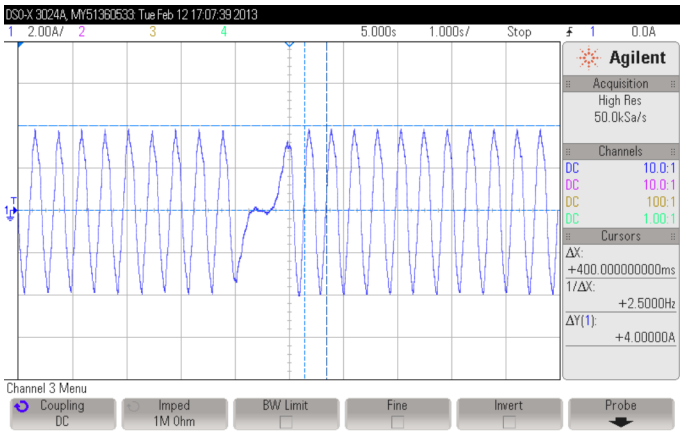


Figure 14-26. -2 to +2 Hz with Full Load Plot

In this example the actual speed is about 2.5 Hz (37 RPM), so we have a total error of about 7 RPM in this example. This error in the speed calculation is due to the error in flux. The error in flux might be due to inaccuracies of the modeled motor compared to the actual motor, possibly due to overheating or current and voltage sensing tolerances.

For best speed-reversal performance: enable Offsets and Rs Recal, disable forced angle, tune speed controller to avoid stalls, tune voltage feedback circuit.

The flux is higher in this test, as shown in Figure 14-27. This again might be a difference in motor model inaccuracies due to motor overheating after many tests under load.

Copyright © 2013–2019, Texas Instruments Incorporated

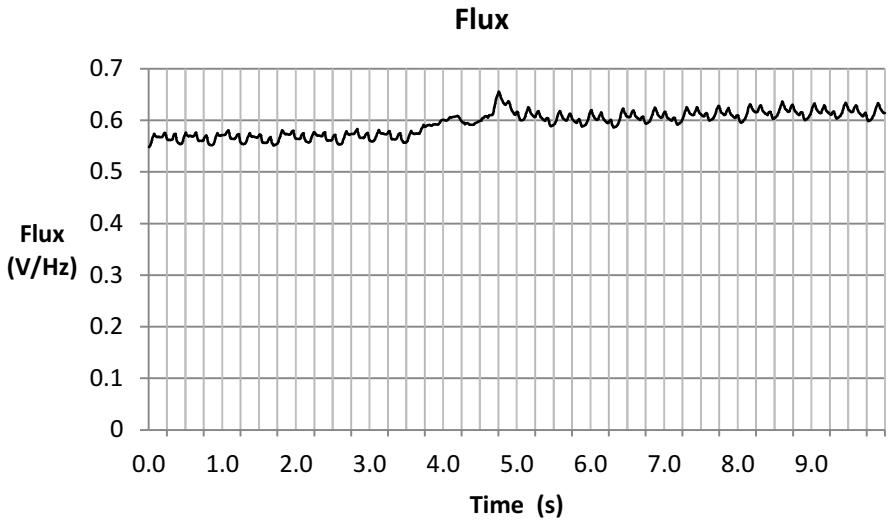


Figure 14-27. Flux Plot

The angle can be seen as it changes direction in Figure 14-28.

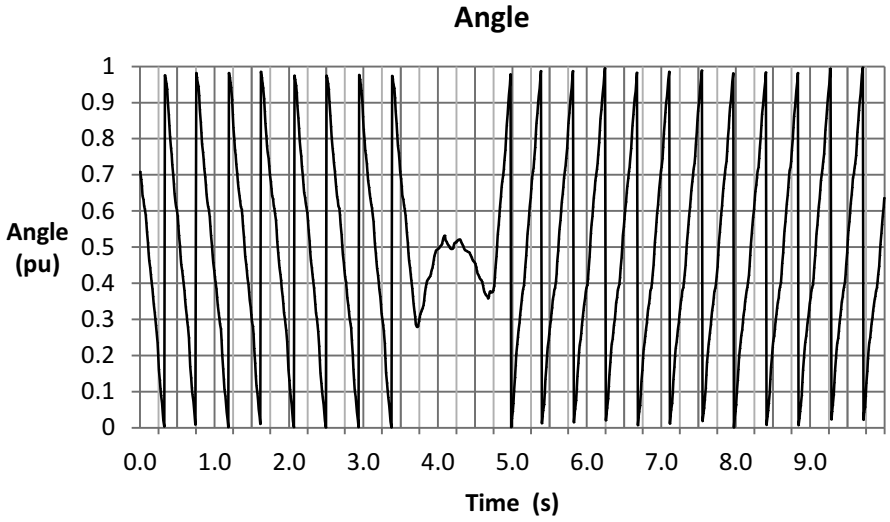


Figure 14-28. Angle Plot

If we zoom-in on the angle (Figure 14-29), when it changes direction, we can see how FAST provides a stable angle even when doing speed reversals with full load.

Copyright © 2013–2019, Texas Instruments Incorporated

Figure 14-29. Zoom-in on Angle Plot

The speed has a double change in sign here as well (Figure 14-30), and again this is because at near zero speed, especially with full load, the speed estimator chases the sign of the speed and in a transient, we can see how it changes signs from ±10 RPM, or ±0.66 Hz.

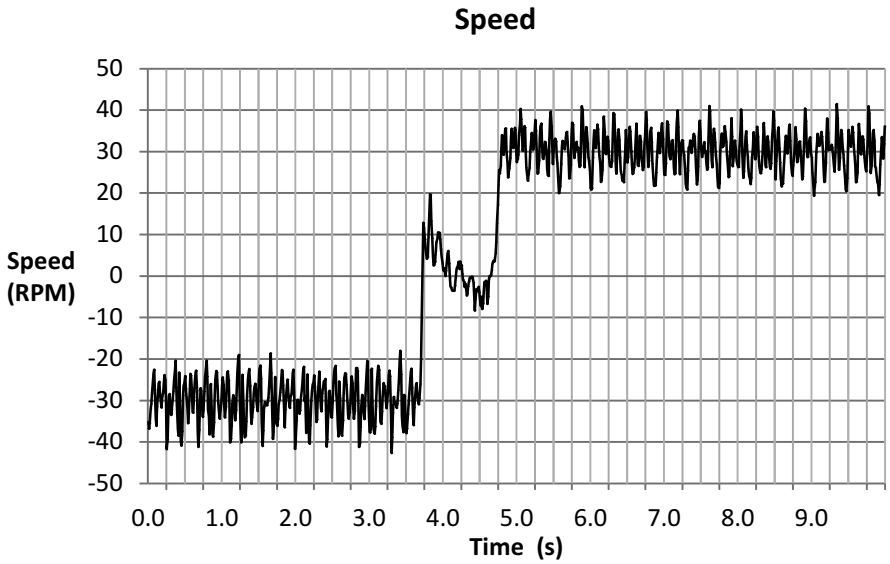


Figure 14-30. Speed Plot

The torque estimator provides a clean zero crossing, and final values (Figure 14-31). However, as the estimation of the torque depends on the estimation of the flux, there is a small offset as the flux changed as per Figure 14-26 in this example.

Copyright © 2013–2019, Texas Instruments Incorporated

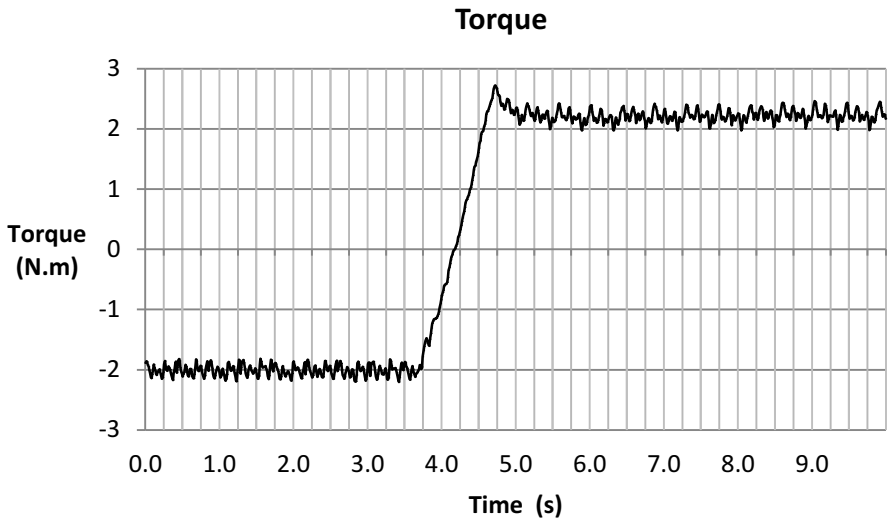


Figure 14-31. Torque Plot

The current, Iq, is also shown in Figure 14-32.

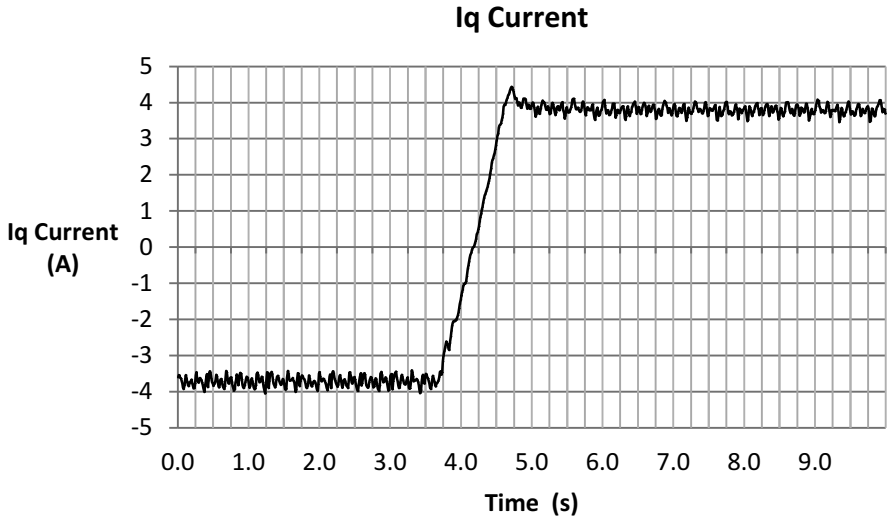


Figure 14-32. Iq Current Plot

14.3 Motor Startup with Full Load

In this section we will discuss a set of considerations to have in order to allow a full load startup with InstaSPIN and the FAST algorithm. After looking at the consideration we will look at a few practical examples.

14.3.1 Motor Startup with Full Load Considerations

The considerations discussed in the previous sections also apply to this mode of operation:

- Enable offsets recalibration; described in Section 14.1.1.1.
- Enable stator Rs recalibration; described in Section 14.1.1.2.
- Enable forced angle; described in Section 14.3.1.1.
- Tune speed controller to avoid motor stall; described in Section 14.1.1.4.
- Tune voltage feedback circuit; described in Section 14.1.1.5.

Copyright © 2013–2019, Texas Instruments Incorporated

14.3.1.1 Enable Forced Angle

In order to start up the motor with full load from stand still, the estimator needs an initial rotating angle to allow some back EMF to be present in the motor, as shown in Figure 14-33. Typically, less than 1 electrical cycle is required for FAST to lock on the real angle. In order to enable a rotating angle, user must enable the forced angle feature of InstaSPIN. Once the motor has gone through a startup, it is recommended to disable forced angle so that the motor can go through a speed reversal. However, if the motor is stalled for a few seconds during any of the low speeds or speed reversal tests, it is recommended to re-enable the forced angle mode to get out of a stalled motor condition.

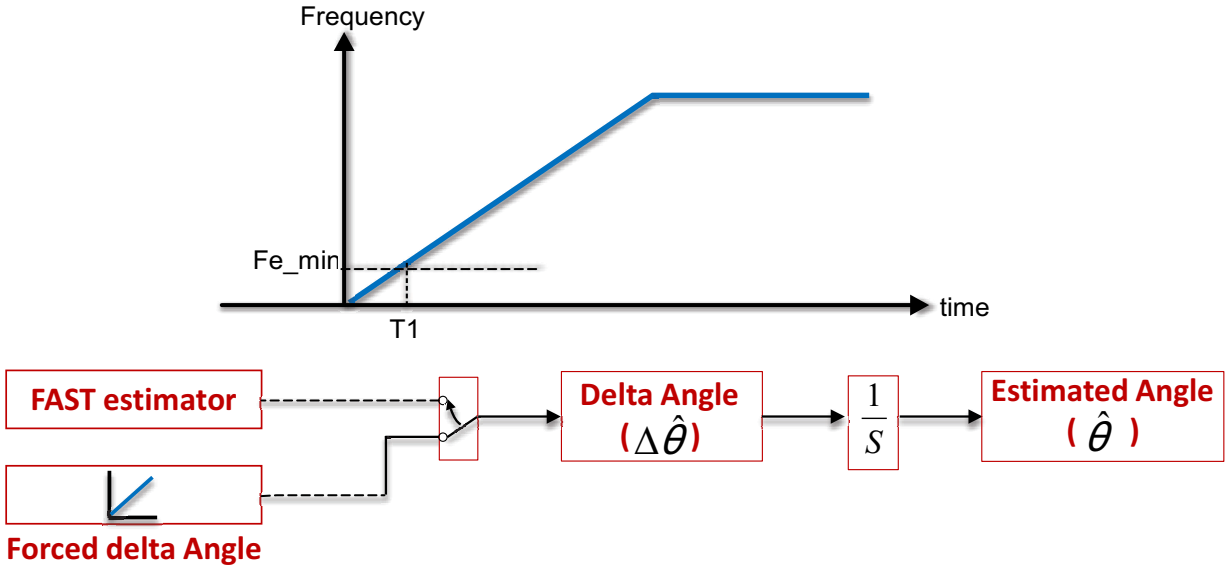


Figure 14-33. Enable Forced Angle

- The forced angle is applied to force the estimator's d-axis angle at low rotor speeds. The forced angle is active from zero to the Fe_min frequency with the default setting of 1 Hz
- Closed loop vector control starts after time T1 by using the angle information from the FAST estimator output.
- The FAST algorithm converges on the rotor angle within one-cycle of electrical frequency. The FAST algorithm is stable at all speeds, even at zero speed.
- For smooth transitions when changing speed direction, turn the forced angle off

The following code example enables forced angle prior to enabling the controller.

```
// enable the forced angle
EST_setFlag_enableForceAngle(obj->estHandle,TRUE);
```

FAST configuration different for best start-up performance: enable forced angle.

14.3.2 Motor Startup with Full Load Examples

Once the above items are covered, let us look at a few examples of motor startup with full load using the dynamometer.

Copyright © 2013–2019, Texas Instruments Incorporated

14.3.2.1 From Standstill to 4 Hz with Full Load

Figure 14-34 shows the current waveform under these conditions. Notice how the current grows from 0 A all the way to 6 A, which is set as the maximum output of the speed controller, or maximum Iq current controller reference (see max current in Chapter 5). It can also be seen that within one cycle of forced angle, the motor current goes back to 4 A, which is the rated current to produce full torque. Keep in mind that in this case the maximum current of the motor is 6 A, while the rated current to produce rated torque is 4 A.

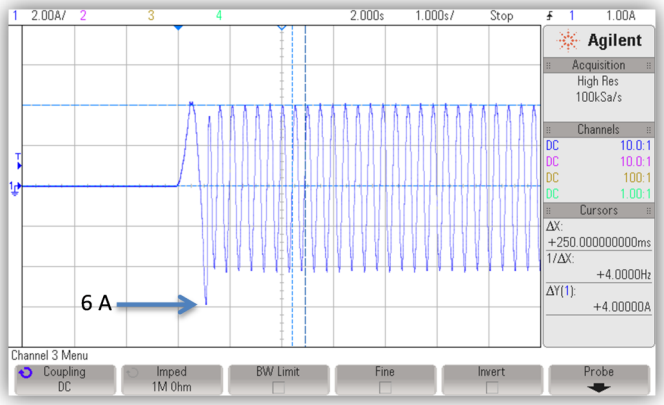


Figure 14-34. Standstill to 4 Hz with Full Load Plot

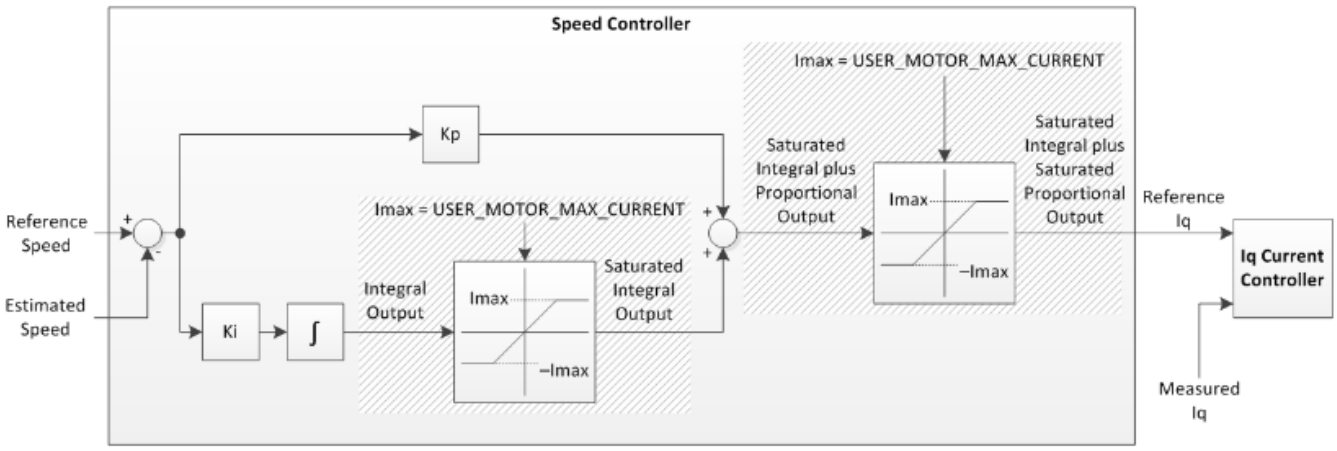


Figure 14-35. Speed Controller Cycle

Figure 14-36 is the flux plot, where we can see how it has a transient, and then it stabilizes.

Copyright © 2013–2019, Texas Instruments Incorporated

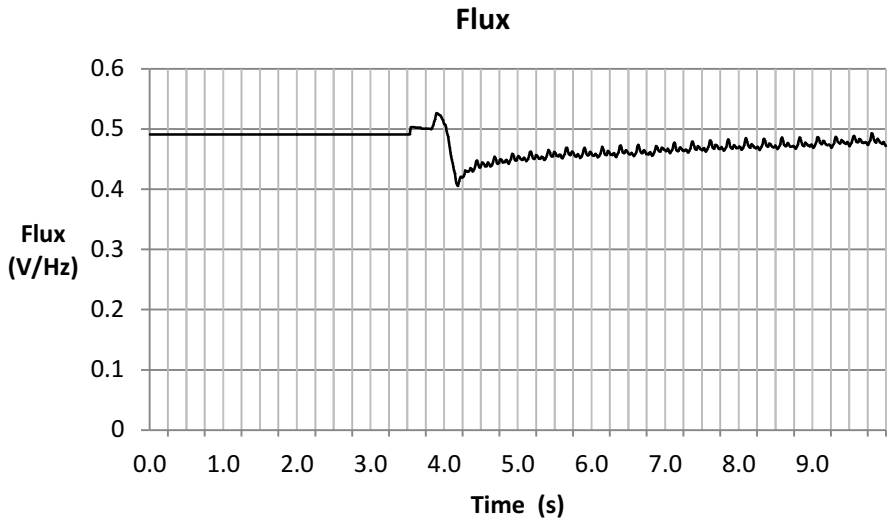


Figure 14-36. Flux Plot

As far as the angle goes, it can be seen that at the beginning, a forced angle is done for less than one cycle (Figure 14-37).

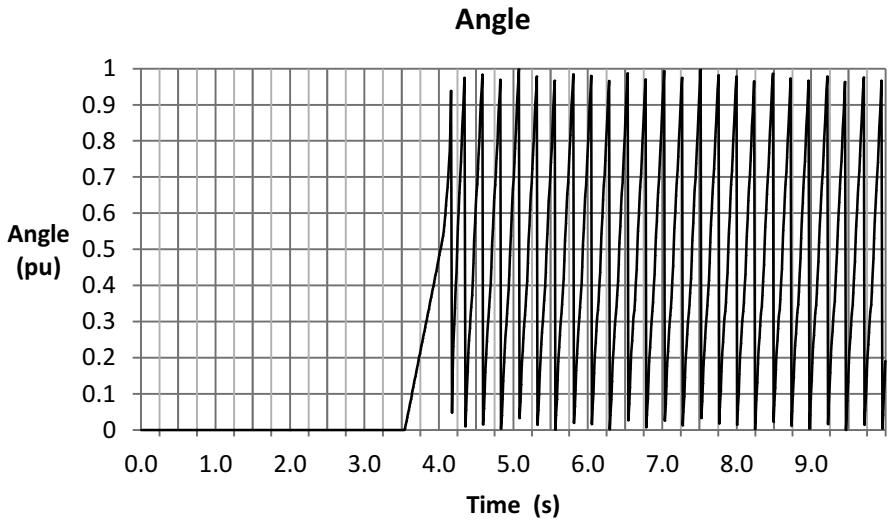


Figure 14-37. Angle Plot

In fact if we zoom-in we are able to tell how many cycles, or actually what percentage of one cycle, the angle was forced. In this example only half of one electrical cycle was used for forced angle.

Typically, less than 1 electrical cycle required to lock on angle.

This can be calculated from Figure 14-38, where a slope of 1 electrical cycle per second is generated, since the angle increased from 0 to 0.5 in 0.5 seconds.

Copyright © 2013–2019, Texas Instruments Incorporated

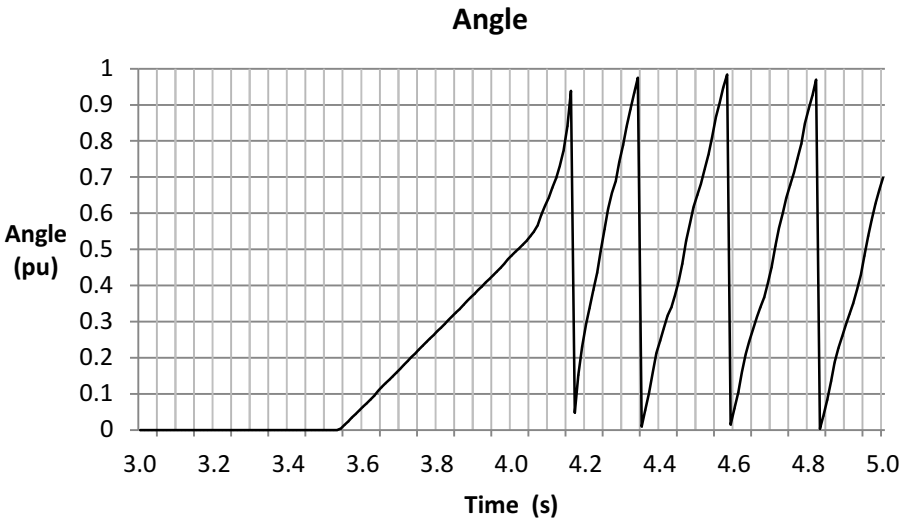


Figure 14-38. Zoom-in on Angle Plot

If other frequencies of forced angle are required, user can change the frequency by modifying the following value in user.h file:

```
//! \brief Defines the forced angle frequency, Hz
#define USER_FORCE_ANGLE_FREQ_Hz (1.0)
```

This frequency might need to be changed due to startup time requirements, by providing a faster forced angle. However, having a faster forced angle requires a faster speed in open loop, which might not be slow enough to rotate the load in open loop.

The estimated speed tells a lot in this example (Figure 14-39). First, it can be seen that there was no initial alignment of the motor compared to the forced angle, that's why the speed goes negative for a period of time. Also, it can be seen that by the time the motor is aligned, there is more current than needed to speed up the motor to the commanded speed. That is why the speed overshoots so much. Typically, a maximum of one electrical cycle is needed for the estimator to catch up with the rotor's flux angle. So a typical maximum of one electrical cycle would be driven in the reverse direction.

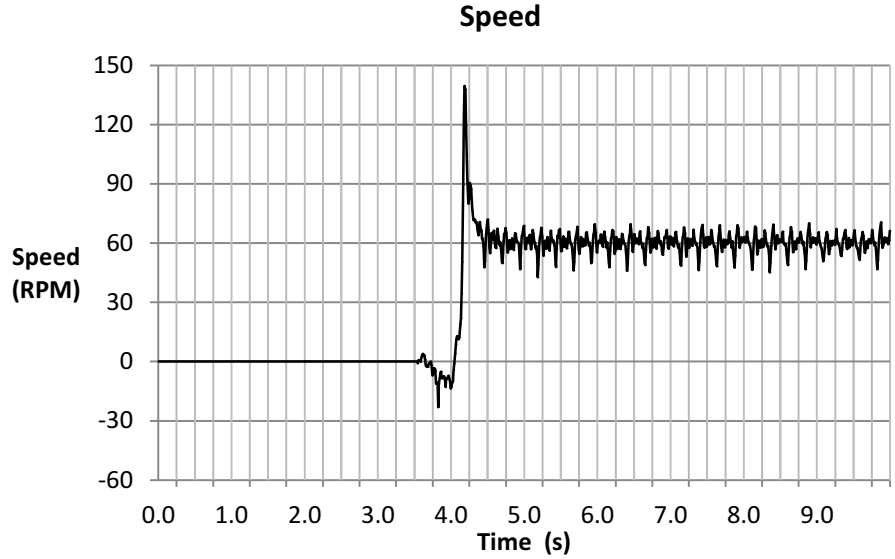


Figure 14-39. Speed Plot

Copyright © 2013–2019, Texas Instruments Incorporated

The estimated torque overshoots due to the transient present in the estimated flux, as shown in Figure 14-40. Although as soon as the estimated angle aligns with the motor angle, this transient is reduced and the estimated torque stabilizes and matches what the dynamometer controller displays.

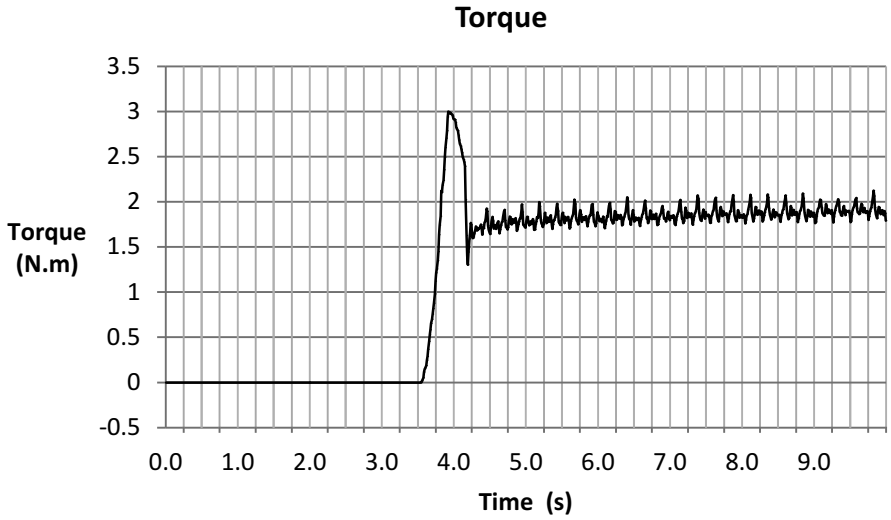


Figure 14-40. Torque Plot

The last plot we will show in this example is the current in Iq (Figure 14-41). Recall we have a limitation of the Iq reference of 6 A, which is the maximum safe current for this motor. It can be seen how this maximum is reached when starting up, then when there is angle alignment the current goes back to the rated current of 4 A.

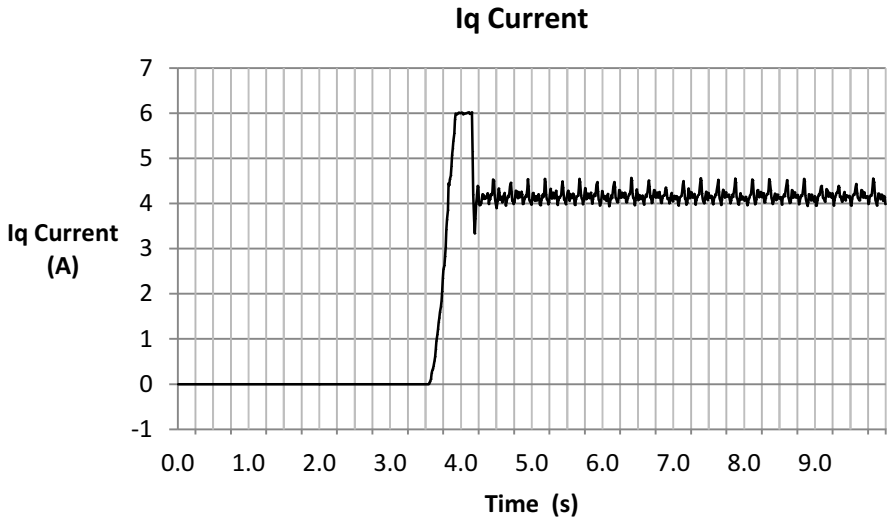


Figure 14-41. Iq Current Plot

14.3.2.2 From Standstill to 2 Hz with Full Load

Again in Figure 14-42 the current reached the maximum safe current of this motor of 6 A, but this time for a longer period of time. However, the time in which forced angle is applied is still under one electrical cycle. It can also be seen that we are approaching the limits of the estimator, as the measured frequency shows 2.38 Hz, where the commanded speed is 2 Hz, effectively a difference of about 6 RPM.

Copyright © 2013–2019, Texas Instruments Incorporated

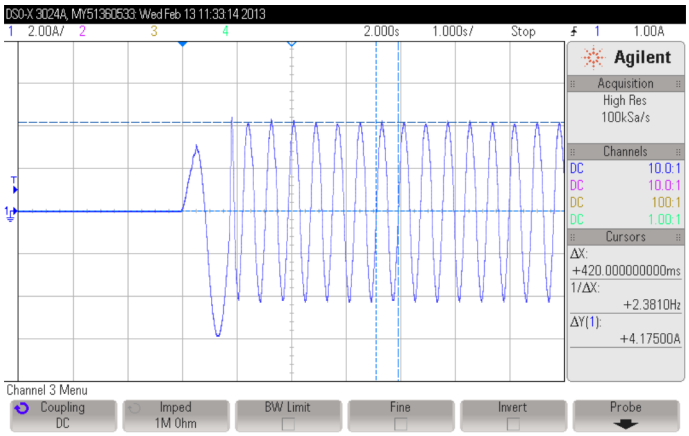


Figure 14-42. Standstill to 2 Hz with Full Load Plot

The estimated flux shows some error and a transient (Figure 14-43), although is still comparable to what we got in other tests running at 2 Hz. The difference in flux compared to a rated flux of 0.5 v/Hz is the reason of the difference of actual electrical frequency compared to estimated electrical frequency.

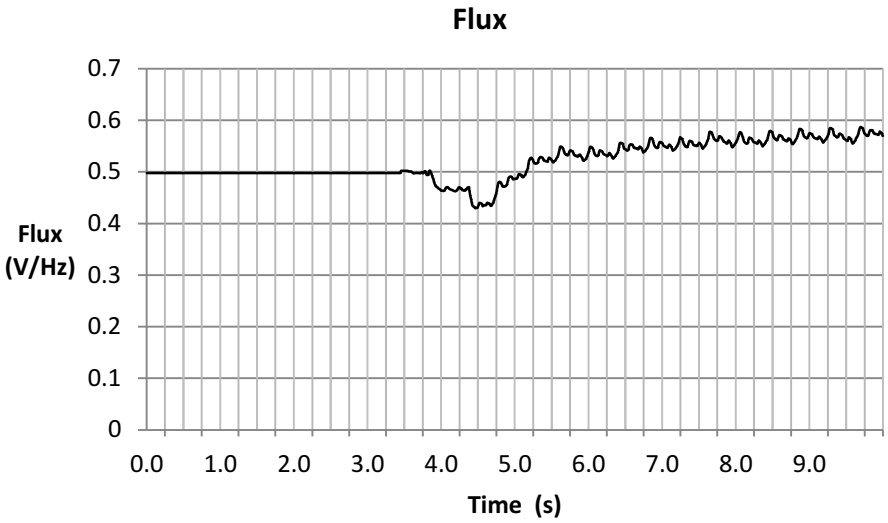


Figure 14-43. Flux Plot

The angle estimation shows a forced angle of just under one electrical cycle (Figure 14-44).

Copyright © 2013–2019, Texas Instruments Incorporated

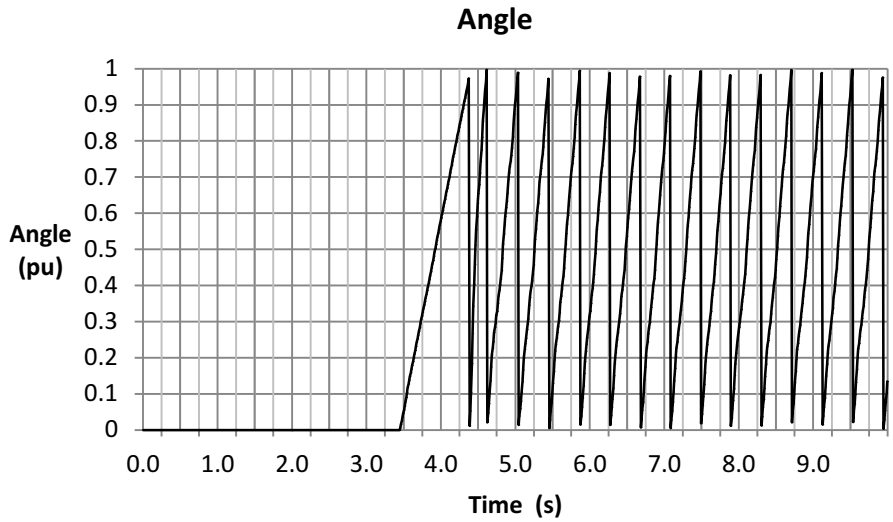


Figure 14-44. Angle Plot

Zooming in the angle, it can be seen how the first cycle is less than one second before it changes frequency (Figure 14-45). Once the motor angle and estimated angle are aligned, the torque production is much higher, causing a motor acceleration beyond the target speed of 30 RPM. This is why the cycle right after the forced angle cycle has a much higher frequency than the following cycle.

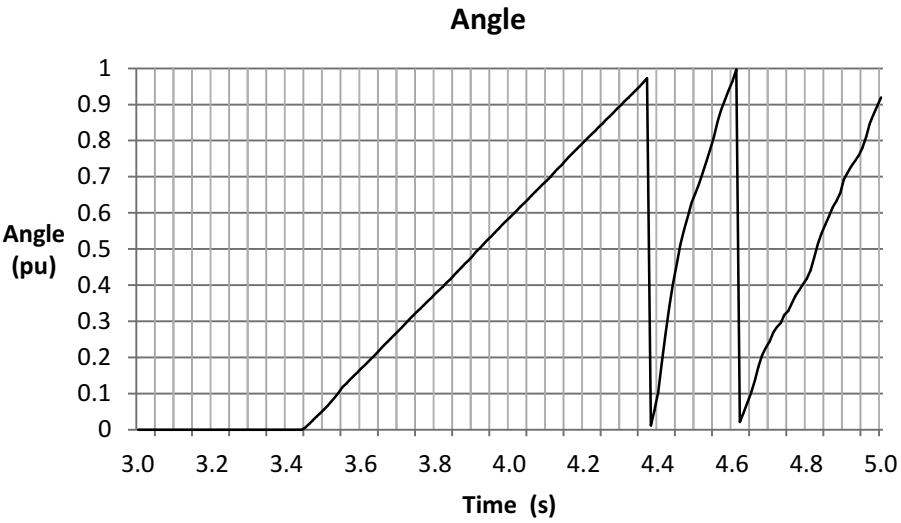


Figure 14-45. Zoom-in on Angle Plot

The speed overshoot can be seen in Figure 14-46. It can also be seen that the motor spins backwards for a small period of time before accelerating to the commanded direction.

Copyright © 2013–2019, Texas Instruments Incorporated

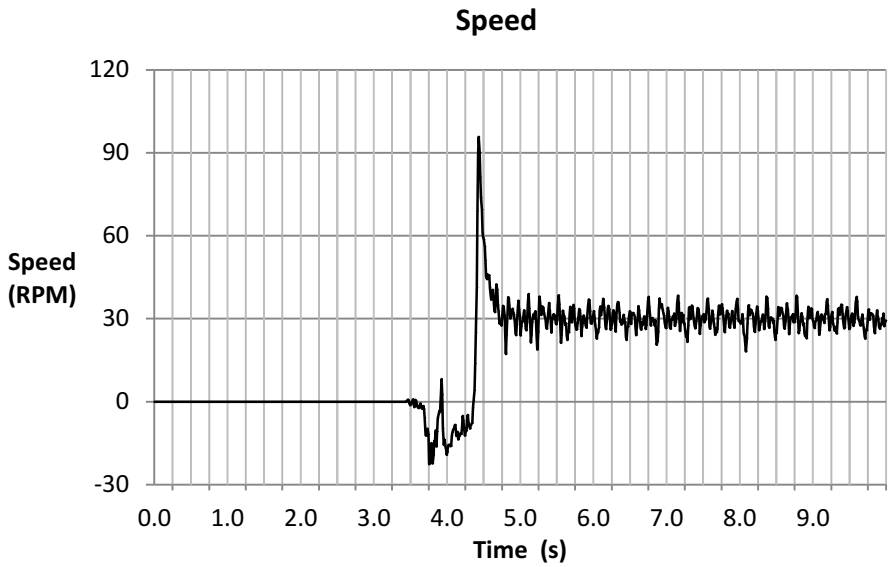


Figure 14-46. Speed Plot

The estimated torque in Figure 14-47 shows the same behavior as the previous example: a transient at the beginning due to the error in the estimated flux, and a steady state error possibly due to measurements inaccuracies and motor heating up.

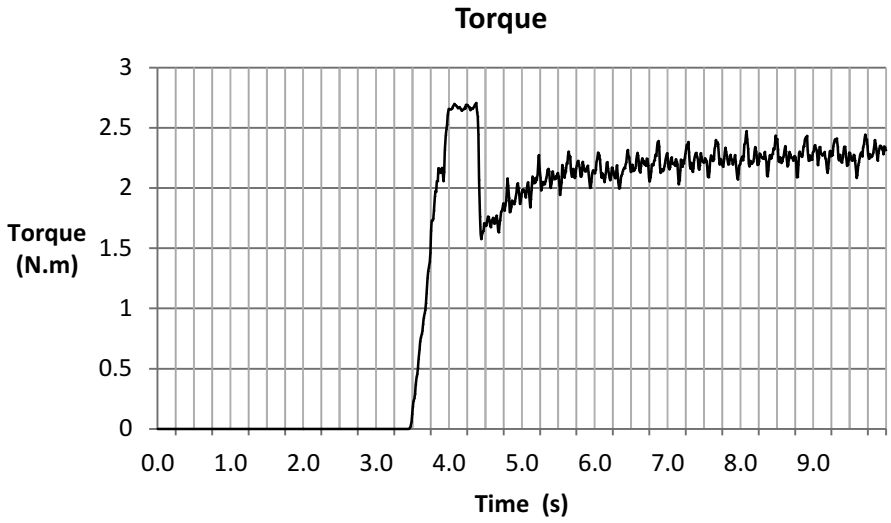


Figure 14-47. Torque Plot

The last plot in this example (Figure 14-48) shows the current Iq, which actually shows a torque production of over 4 A, generating a higher torque than the previous example.

Copyright © 2013–2019, Texas Instruments Incorporated

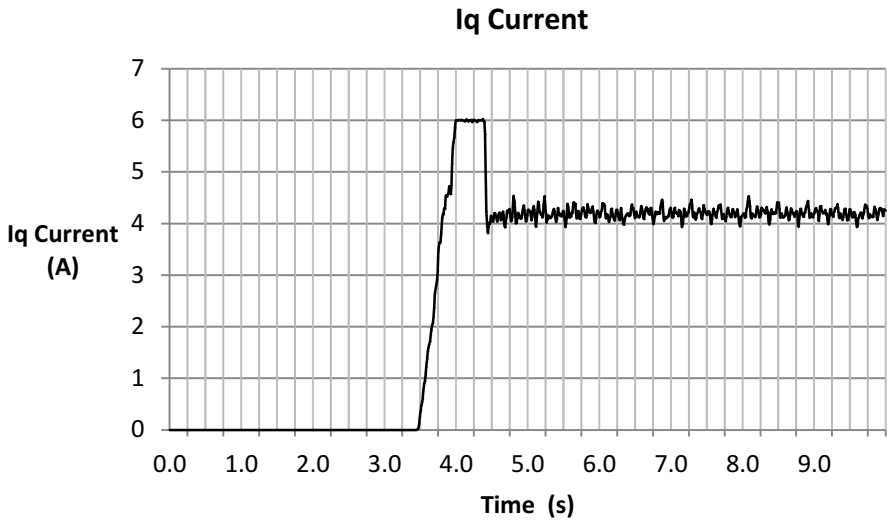


Figure 14-48. Iq Current Plot

14.4 Rapid Acceleration from Standstill With Full Load

In this section we will discuss a set of considerations to allow a full-load startup with the FAST algorithm achieving the quickest ramp from standstill to commanded speed. Reduced start-up time is also discussed in Figure 14-4. The challenges of a full-load fast-ramp at start-up are discussed in this section.

Regardless of the speed controller used, and speed controller gains, there are a few considerations, or configurations, that need to be taken into account in order to get the motor in closed loop as fast as possible from the moment the user enables the system. After looking at the considerations we will look at a few practical examples.

14.4.1 Fastest Motor Startup with Full Load without Motor Alignment Considerations

The considerations discussed in the previous sections also apply to this mode of operation:

- Load valid offsets and disable offset recalibration; described in Section 14.4.1.1.
- Load valid Rs and disable Rs recalibration; described in Section 14.4.1.2
- Enable forced angle; Section 14.3.1.1.
- Tune speed controller to avoid motor stall; described in Section 14.1.1.4.
- Tune voltage feedback circuit; described in Section 14.1.1.5.

14.4.1.1 Load Valid Offsets and Disable Offset Recalibration

This configuration allows the system to avoid time spent doing the offsets recalibration after the user has commanded a motor startup. However, since low speed and motor startup requires the offsets to be correct, the user must load these pre-calibrated offsets prior to the CTRL_setFlag_enableCtrl(ctrlHandle, TRUE) function. The following code example loads known pre-calculated offsets into the HAL object.

```
// disable automatic calculation of bias values
CTRL_setFlag_enableOffset(ctrlHandle,FALSE);

// set the current bias
HAL_setBias(halHandle,HAL_SensorType_Current,0,_IQ(I_A_offset));
HAL_setBias(halHandle,HAL_SensorType_Current,1,_IQ(I_B_offset));
HAL_setBias(halHandle,HAL_SensorType_Current,2,_IQ(I_C_offset));

// set the voltage bias
HAL_setBias(halHandle,HAL_SensorType_Voltage,0,_IQ(V_A_offset));
HAL_setBias(halHandle,HAL_SensorType_Voltage,1,_IQ(V_B_offset));
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
HAL_setBias(halHandle,HAL_SensorType_Voltage,2,_IQ(V_C_offset));
```

Notice that I_A_offset, I_B_offset, I_C_offset, V_A_offset, V_B_offset and V_C_offset are the pre-calculated offsets on previous runs of the system. The following example can be used to get these offsets from the HAL object when they are updated after offsets recalibrations are enabled.

```
// enable automatic calculation of bias values
CTRL_setFlag_enableOffset(ctrlHandle,TRUE);

// Return the bias value for currents
I_A_offset = HAL_getBias(halHandle,HAL_SensorType_Current,0);
I_B_offset = HAL_getBias(halHandle,HAL_SensorType_Current,1);
I_C_offset = HAL_getBias(halHandle,HAL_SensorType_Current,2);

// Return the bias value for voltages
V_A_offset = HAL_getBias(halHandle,HAL_SensorType_Voltage,0);
V_B_offset = HAL_getBias(halHandle,HAL_SensorType_Voltage,1);
V_C_offset = HAL_getBias(halHandle,HAL_SensorType_Voltage,2);
```

14.4.1.2 Load Valid Rs and Disable Rs Recalibration

In order to avoid spending time recalibrating the resistance, it is also important to make sure that the resistance value provided in user.h is accurate, and that the resistance recalibration feature is disabled. The resistance provided in user.h is shown here:

```
#define USER_MOTOR_Rs (2.6)
```

And the following code example disables R_s recalibration:

```
EST_setFlag_enableRsRecalc(obj->estHandle,FALSE);
```

14.4.1.3 Fastest Motor Startup with Full Load without Motor Alignment Example

Figure 14-49 represents one of the phase currents when doing a fast acceleration with no alignment. As can be seen, the current goes up to the maximum limit for less than one cycle, then it speeds up to the commanded speed reference of 200 RPM.

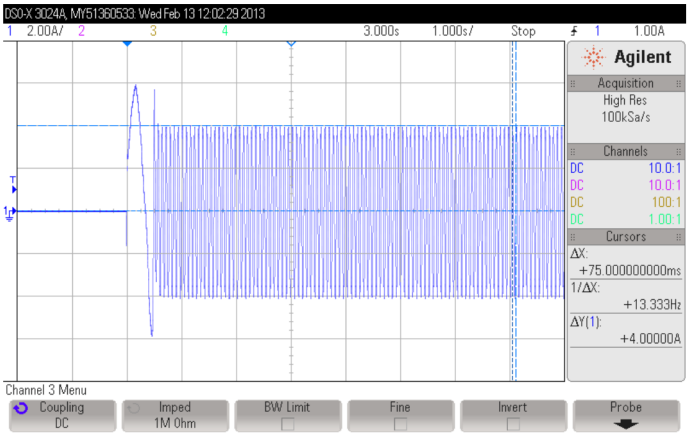


Figure 14-49. Fast Acceleration Without Alignment Plot

Copyright © 2013–2019, Texas Instruments Incorporated

The flux also has a transient which happens while the estimated angle is not aligned with the actual motor angle (Figure 14-50), and then after the transient, it stabilizes to a fairly constant value.

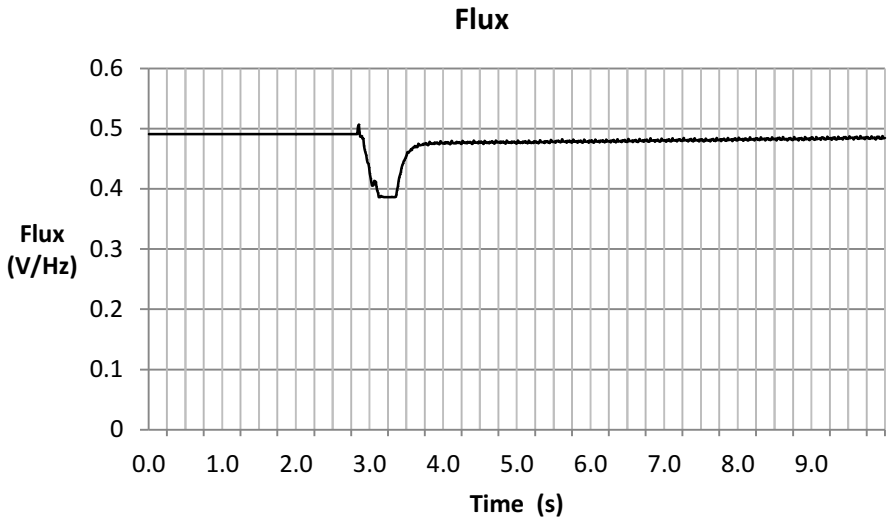


Figure 14-50. Flux Plot

The angle can be seen to be forced for the first cycle (Figure 14-51), and then the frequency is rapidly changed since it is already in closed loop using the estimated angle instead of the forced angle.

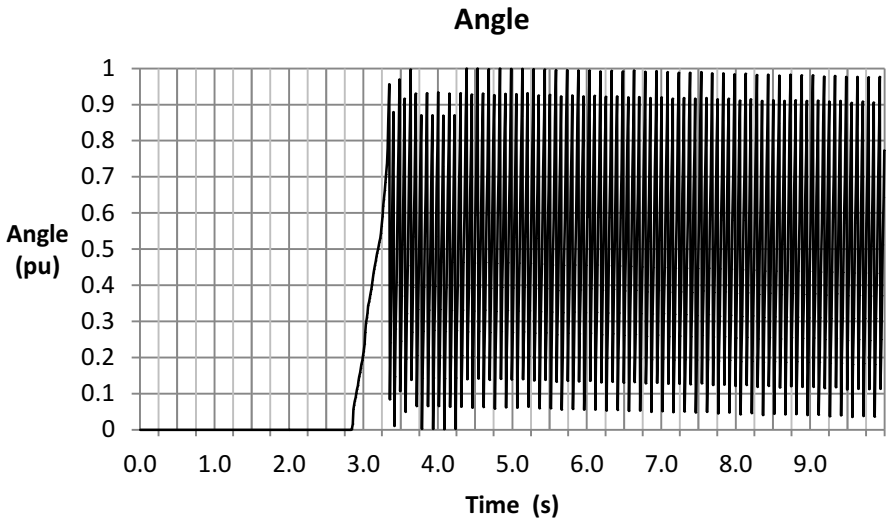


Figure 14-51. Angle Plot

Zooming in to the angle (Figure 14-52) it can be seen that the forced angle lasts less than once electrical cycle to ramp up to the commanded speed of 200 RPM.

Copyright © 2013–2019, Texas Instruments Incorporated

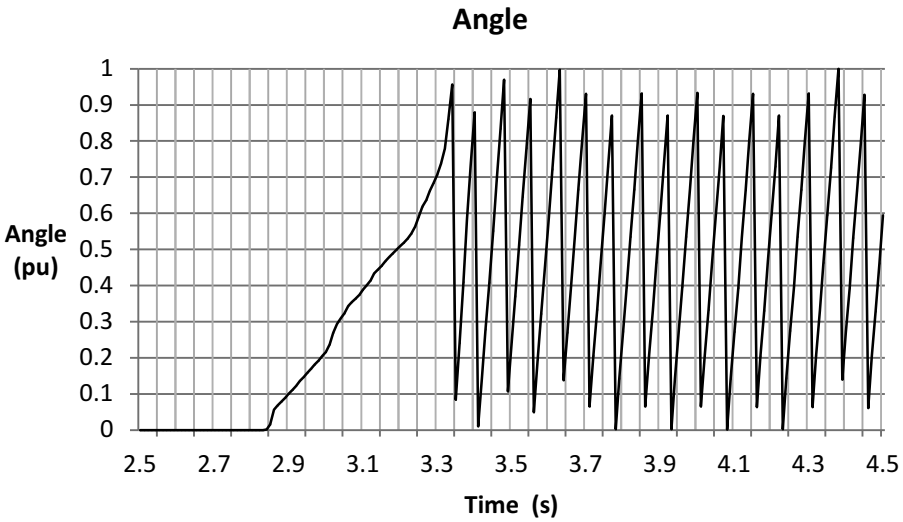


Figure 14-52. Zoom-in on Angle Plot

Due to the initial misalignment, it can be seen that the speed goes negative for a short period of time (Figure 14-53), and once the estimated angle is aligned with the motor angle, the speed accelerates very rapidly up to the commanded speed of 200 RPM. In fact, there is an overshoot due to the excess of current accumulated on the integral portion of the speed controller while the angle was forced.

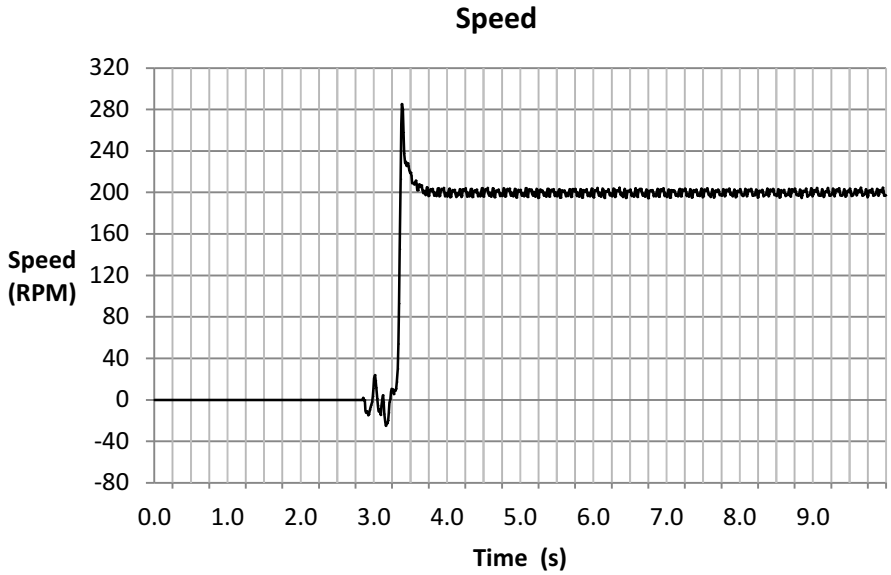


Figure 14-53. Speed Plot

We can see the effect of the flux angle error in the torque signal (Figure 14-54), as well as the excess in current due to the forced angle. After this transient in the flux, the torque signal is accurate and can be seen constant once the speed has stabilized.

Copyright © 2013–2019, Texas Instruments Incorporated

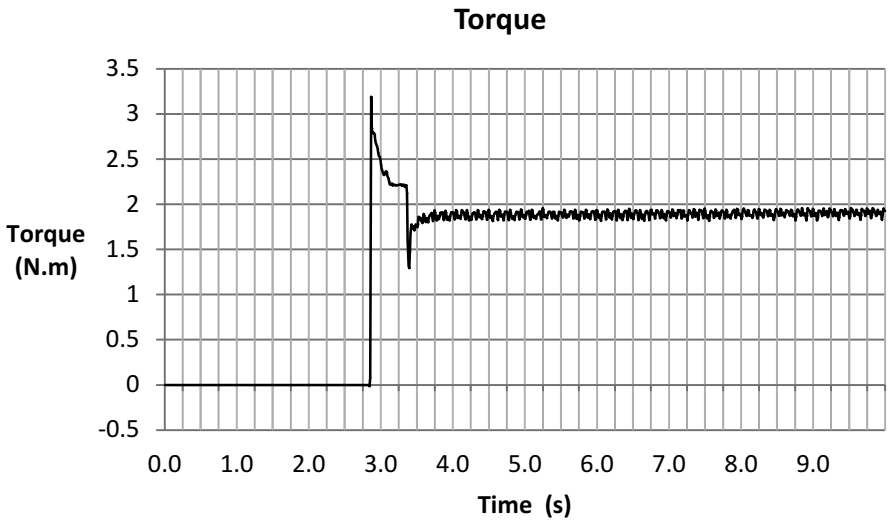


Figure 14-54. Torque Plot

The current Iq has an overshoot of about 0.5 A due to the step command on the speed, as shown in Figure 14-55. That overshoot can be seen as a small impulse at the beginning, and then it goes down to the limit of 6 A that we configured in user.h as the motor maximum current. The current goes down to the rated current value of 4 A to produced full torque after the motor has sped up to the commanded speed and the estimated flux has stabilized.

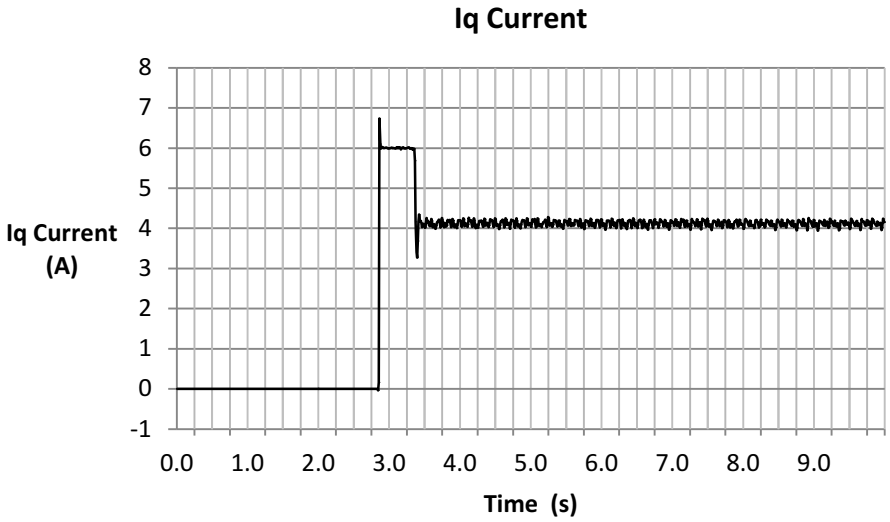


Figure 14-55. Iq Current Plot

As a conclusion of this example we can say that the fastest acceleration we can achieve will depend on the initial difference between the initial estimated angle and the real value of the motor. This is because the forced angle feature will be enabled while those angles are aligned, taking a maximum of one electrical cycle. The electrical cycle of the forced angle provided in user.h has a default frequency of 1 Hz, so worst case it takes 1 second to speed up from standstill. However, depending on the type of load present in the motor shaft, this frequency of the forced angle can be changed to a higher frequency, providing a fastest acceleration from standstill.

Copyright © 2013–2019, Texas Instruments Incorporated

14.4.2 Fastest Motor Startup with Full Load with Motor Alignment Considerations

To overcome the initial motor misalignment problem, we will talk about using the Rs recalibration as a tool to allow a motor alignment prior to running the motor in closed loop. In this example, the Rs recalibration current is used to rotate the motor shaft to an initial alignment position. The considerations discussed in the following sections also apply to this mode of operation:

- Load valid offsets and disable offset recalibration; described in Section 14.4.1.1.
- Enable Rs stator recalibration; described in Section 14.1.1.2.
- Maximize current slope; described in Section 14.4.2.1.
- Enable forced angle; Section 14.3.1.1.
- Tune speed controller to avoid motor stall; described in Section 14.1.1.4.
- Tune voltage feedback circuit; described in Section 14.1.1.5.

14.4.2.1 Maximize Current Slope

This consideration is required in this mode of operation since a negative DC current is applied to the motor in order to recalibrate Rs. This DC current must be removed as fast as possible before running the motor in closed loop. In order to do this, the following code example should be used to change the maximum current slope value to a maximum.

```
// set max current slope value to a maximum
EST_setMaxCurrentSlope_pu(obj->estHandle,_IQ(127.99));
```

The input parameter of the maximum slope function expects a value in IQ24, which has a maximum value of 127.99. This would cause a step increase in the current when it starts injecting current to recalibrate the resistance, as well as a step to remove such current, which is what we want to accomplish.

14.4.2.2 Fastest Motor Startup with Full Load with Motor Alignment Example

In this example the application must allow an initial alignment time before running in closed loop. That alignment time is configured in user.c as the Rs recalibration time, loaded in the following two array members:

```
pUserParams->RsWaitTime[EST_Rs_State_RampUp] = (uint_least32_t)(1.0*USER_EST_FREQ_Hz);
pUserParams->RsWaitTime[EST_Rs_State_Fine] = (uint_least32_t)(5.0*USER_EST_FREQ_Hz);
```

In this example, a total of 6 seconds will be used to align the motor. Alignment in this example should be done with enough time and enough current to allow a visual motor reposition and alignment before running in closed loop. In future revisions of InstaSPIN, initial position detection (IPD) will be added to avoid motor alignment altogether. Keep in mind that the internal resistance Rs is only updated during the EST_Rs_State_Fine state. Figure 14-56 shows one of the current waveforms when this is run in. It can be seen how clean the current accelerates in a step from a DC value of -1 A to 0 A (which is the removal of the current used for the motor alignment) and then from 0 A to 4 A sinusoidal (8 A peak to peak). This is because the motor was initially aligned, and no reverse operation was caused by any misalignment. Again, the alignment should be verified visually by the user.

Copyright © 2013–2019, Texas Instruments Incorporated

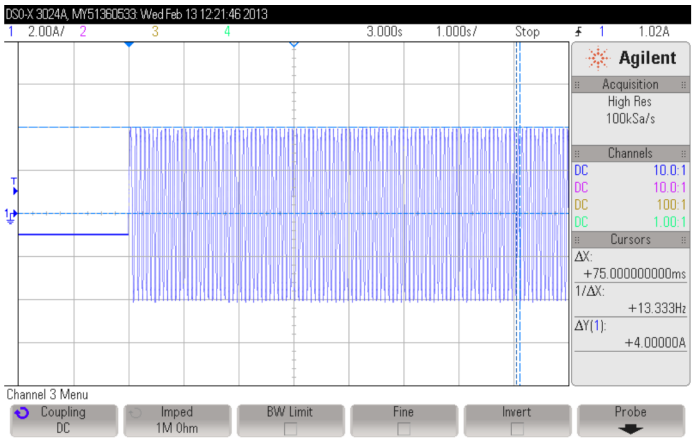


Figure 14-56. Fastest Motor Startup with Full Load with Motor Alignment Plot

If we zoom-in the first portion of the current (Figure 14-57), we can see how the -1 A of current is removed instantaneously with a step, due to our very high current slope configuration, and then it is followed by the speed response. At this point, the speed controller can be tuned as aggressive as desired to achieve the desired acceleration response.

Copyright © 2013–2019, Texas Instruments Incorporated

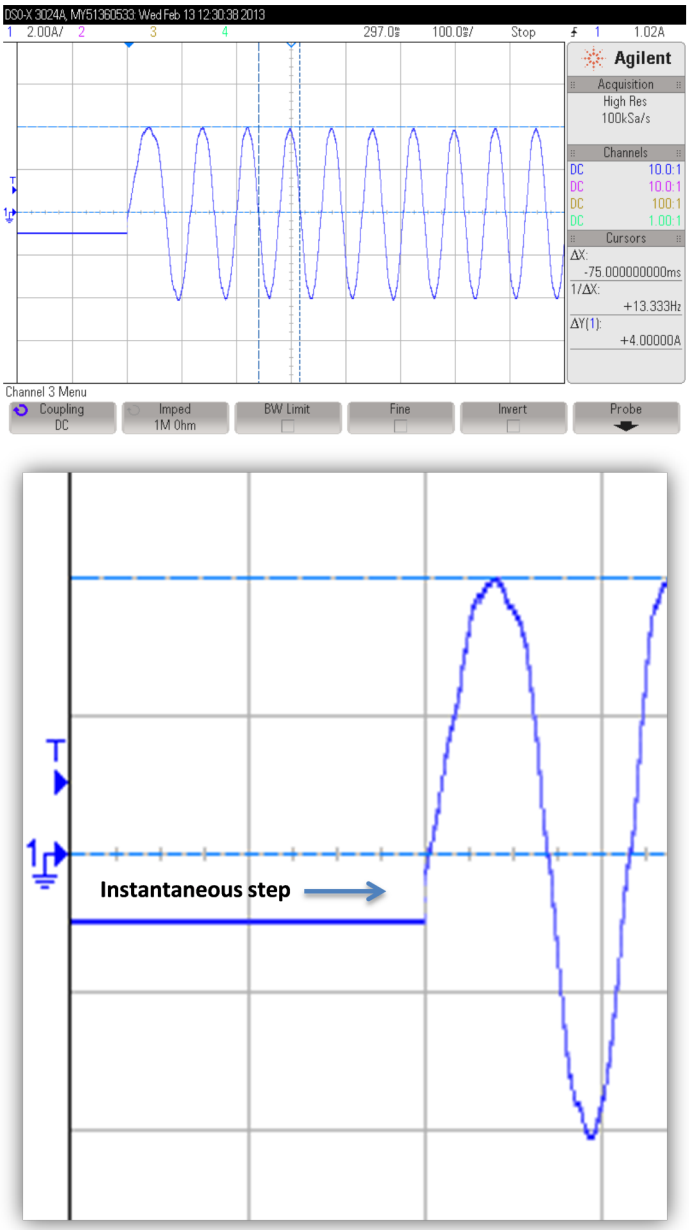


Figure 14-57. Zoom-in on the Current Plot

We can see how the flux instead of a transient like it had before, now it only stabilizes to a constant value when it's running in closed loop (Figure 14-58).

Copyright © 2013–2019, Texas Instruments Incorporated

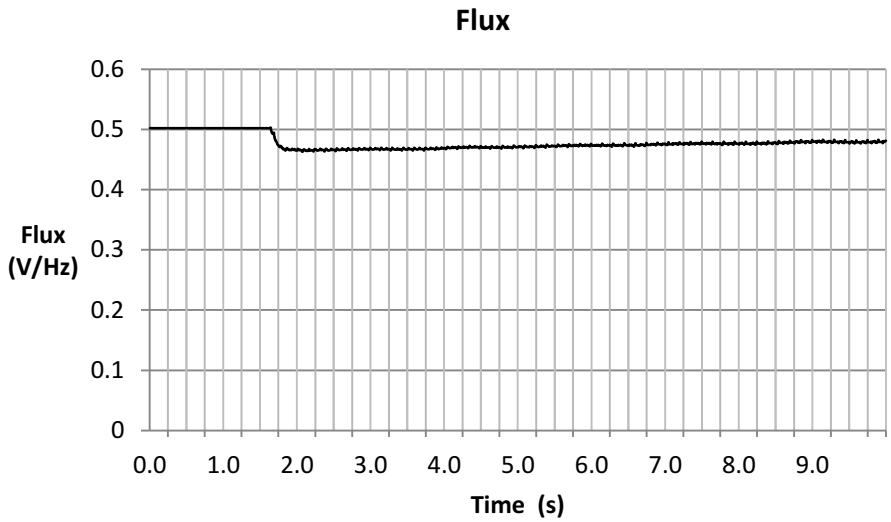


Figure 14-58. Flux Plot

The angle waveform goes from zero to a high frequency (Figure 14-59), suggesting that the forced angle was not even active during ramp up due to the initial alignment.

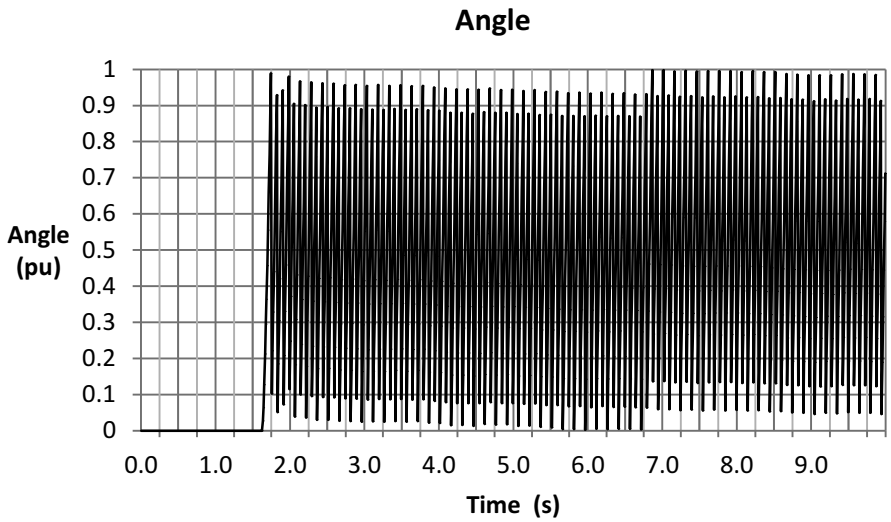


Figure 14-59. Angle Plot

Zooming in to the angle (Figure 14-60), again it can be seen that it ramped up with no interaction of the forced angle feature, taking the motor into a closed loop right from standstill.

Copyright © 2013–2019, Texas Instruments Incorporated

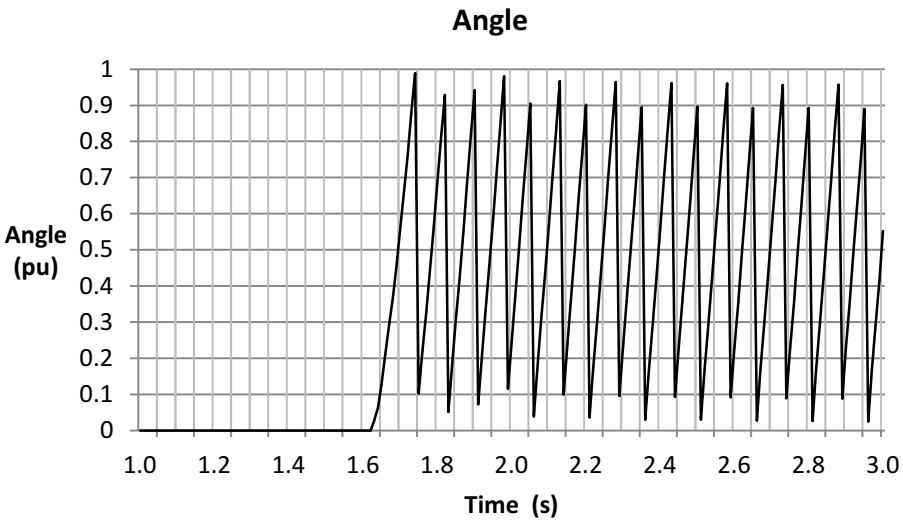


Figure 14-60. Zoom-in on Angle Plot

Looking at the estimated speed (Figure 14-61), it can be seen that there is no negative rotation. Instead, the speed response is the speed controller speed response, with no forced angle initial misalignments.

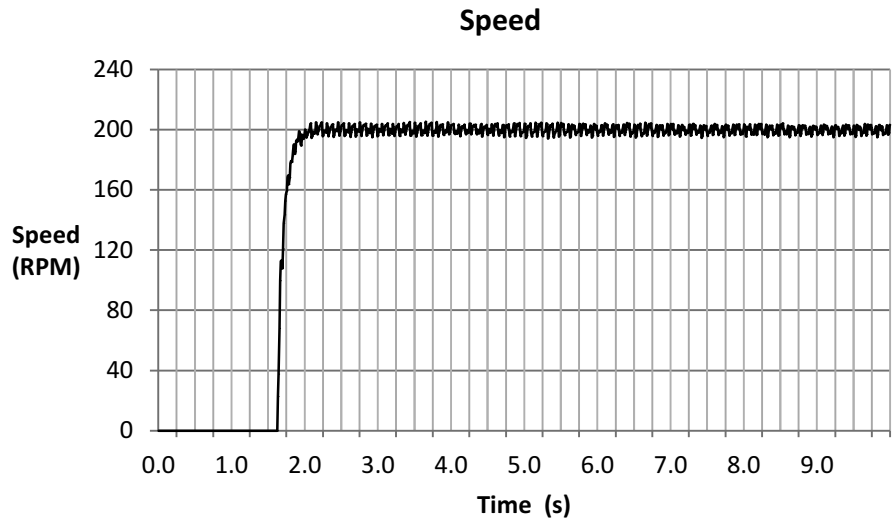


Figure 14-61. Speed Plot

The estimated torque also grows from zero to a target of almost 2 Nm (Figure 14-62), stabilizing at around 1.9 which is the target we set it to in the dynamometer.

Copyright © 2013–2019, Texas Instruments Incorporated

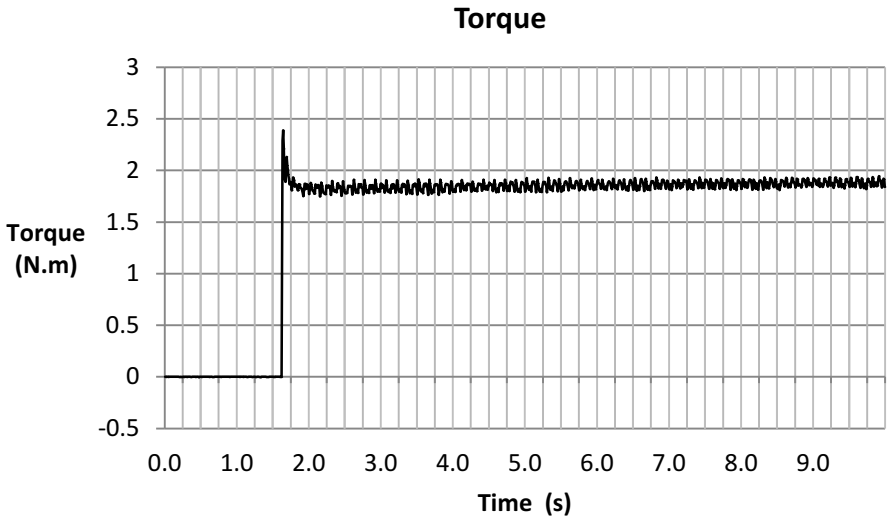


Figure 14-62. Torque Plot

We can see how the current overshoots at the very beginning due to the high acceleration command provided by the speed controller (Figure 14-63), and right after the overshoot we can see it stable at around 4 A producing full torque.

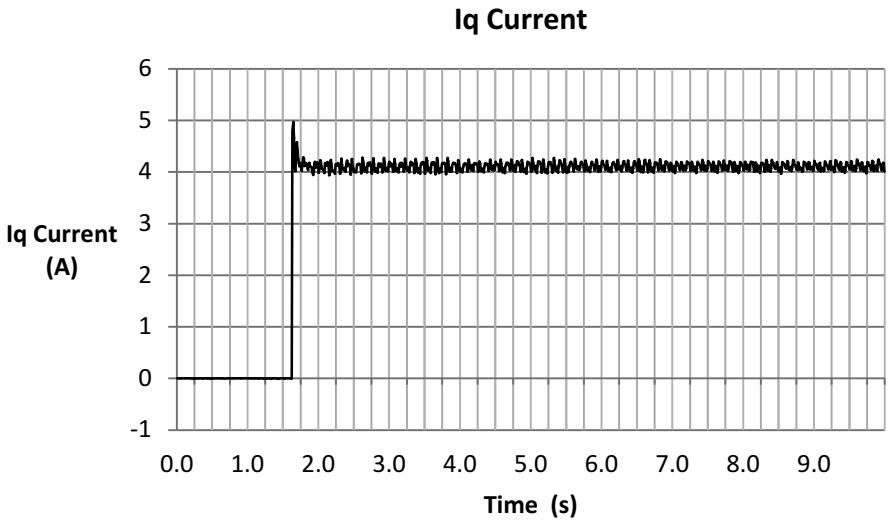


Figure 14-63. Iq Current Plot

14.5 Overloading and Motor Overheating

In this section we will discuss a set of considerations to have in order to allow an overloading condition, and still be able to run for long periods of time. In order to achieve this, we will make use of the Rs Online feature of InstaSPIN which allows us to identify and recalibrate the stator resistance (Rs) while the motor is running.

14.5.1 Overloading and Motor Overheating Considerations

The considerations discussed in the following sections also apply to this mode of operation:

- Enable offsets recalibration; described in Section 14.1.1.1.
- Enable stator Rs recalibration; described in Section 14.1.1.2.
- Enable Rs online feature; described in Chapter 15.

Copyright © 2013–2019, Texas Instruments Incorporated

- Enable forced angle; Section 14.3.1.1.
- Tune speed controller to avoid motor stall; described in Section 14.1.1.4.
- Tune voltage feedback circuit; described in Section 14.1.1.5.

14.5.2 Overloading and Motor Overheating Example

In this example the load is increased 30% above the rated torque of the motor (Figure 14-64). The dynamometer is set to a torque command of 2.5 N·m. Keep in mind that the rated torque capability of this motor is about 1.9 N·m, so in fact we are putting about 130% load to the motor shaft. This will cause the motor to overheat, hence the need for the Rs Online feature to keep an accurate resistance while the motor is running.

In Figure 14-64, where a slow rotating angle, which is due to the Rs Online feature can be seen superimposed on the 5 A amplitude current. To learn more about the Rs Online feature, see Chapter 15.

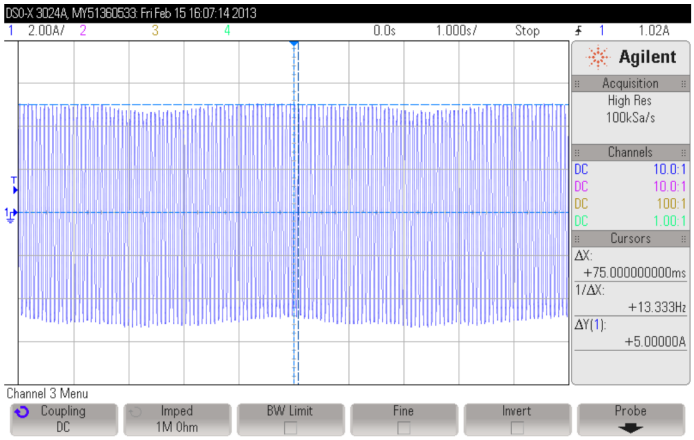


Figure 14-64. Overloading and Motor Overheating Plot

Zooming into the current (Figure 14-65), we can clearly see the frequency to be 13.33 Hz, which is exactly what we command for the speed reference, which in this case is 200 RPM. The conversion is well known, which depends on the number of pole pairs. Speed (RPM) = Speed (Hz) * 60 / Pole Pairs = 13.33 Hz * 60 / 4 = 200 RPM.

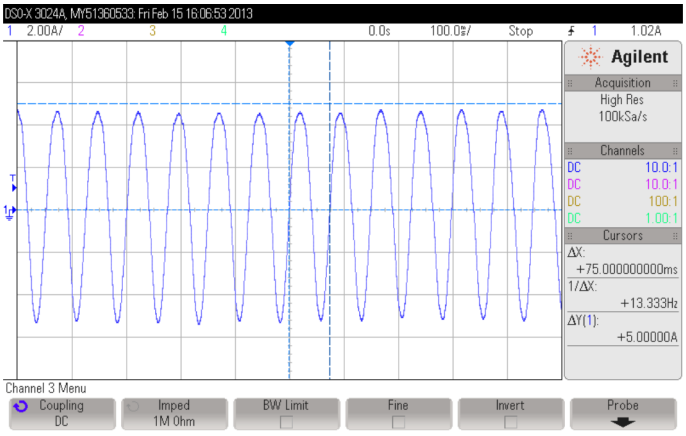


Figure 14-65. Zoom-in on Overloading and Motor Overheating Plot

The stator resistance is captured for a period of 5 minutes (300 seconds) and it is shown in Figure 14-66. It can be seen that the value we start with is 2.8 Ohms, and during a period of about 200 seconds it reaches about 3.45 Ohms, stabilizing at this value.

Copyright © 2013–2019, Texas Instruments Incorporated

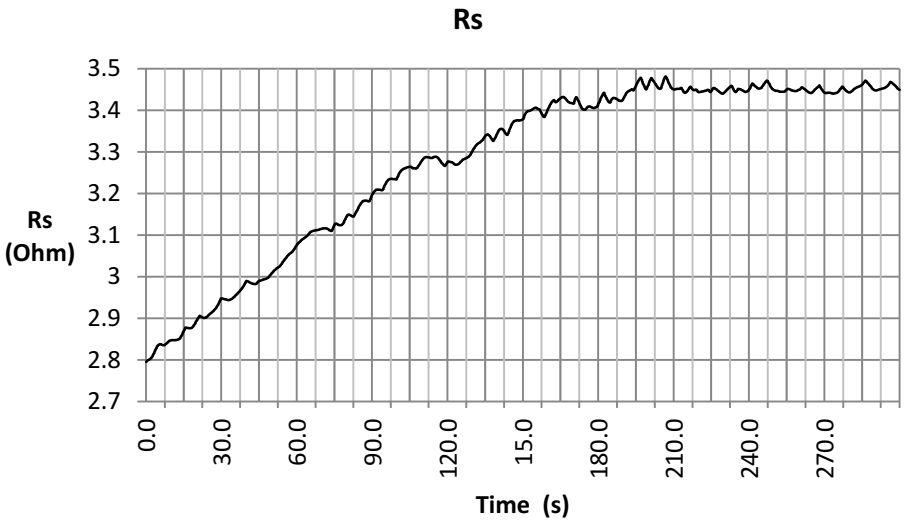


Figure 14-66. Stator Resistance Plot

With this increase, we have a difference in resistance of (3.45-2.8)/2.8 * 100% = 23% increase. Applying the equations we use in Chapter 15, this difference in resistance represents a motor temperature of:

$$T = T_0 + \frac{\frac{R}{R_0} - 1}{\alpha}$$

$$T = 30°C + \frac{\frac{3.45\Omega}{2.8\Omega} - 1}{0.00393°C^{-1}}$$

$$T = 89°C$$

The following estimated values were taken after 10 minutes of working, where the resistance measures about 3.5 Ohms. The estimated flux (Figure 14-67) measures a value very close to the rated flux of 0.5 V/Hz.

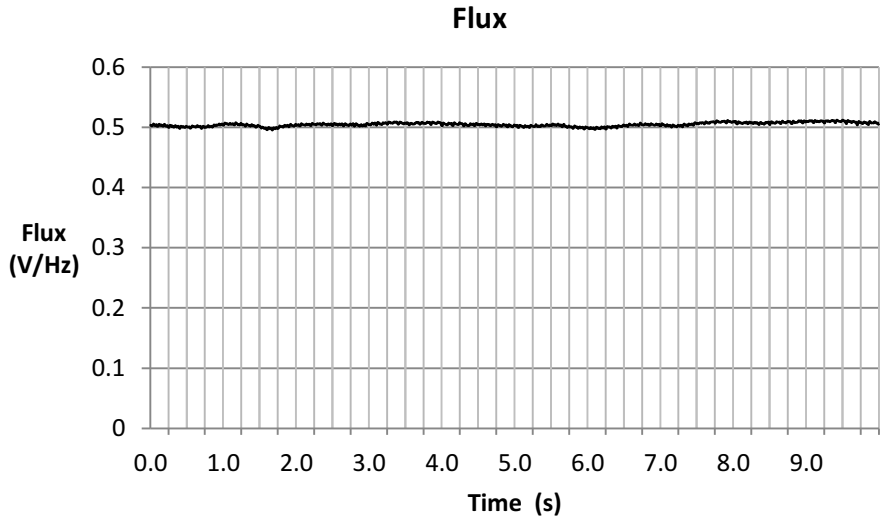


Figure 14-67. Flux Plot

We plot the angle (Figure 14-68) when the motor is 30% overloaded.

Copyright © 2013–2019, Texas Instruments Incorporated

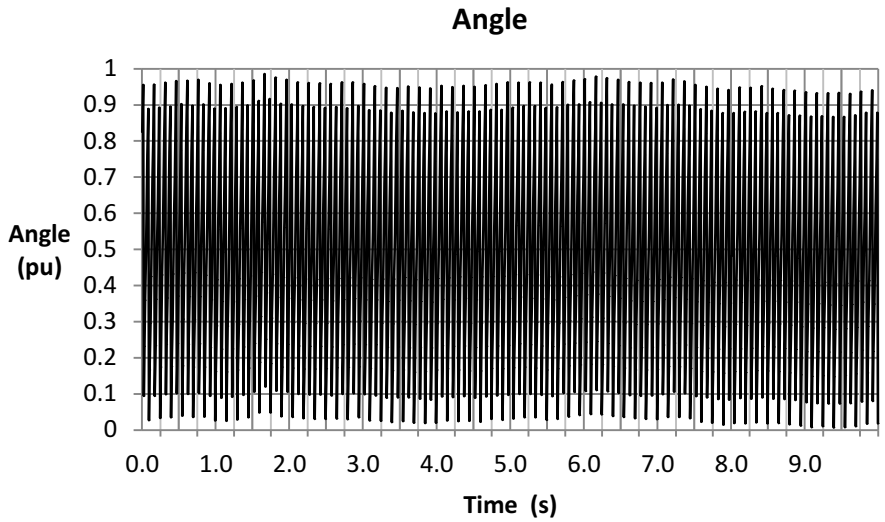


Figure 14-68. Angle Plot

Zooming into the angle (Figure 14-69), expanding one second of information, we see a clean and continuous ramp. Keep in mind that the data here was taken every 200 samples, so there is a discontinuity at the end of every cycle.

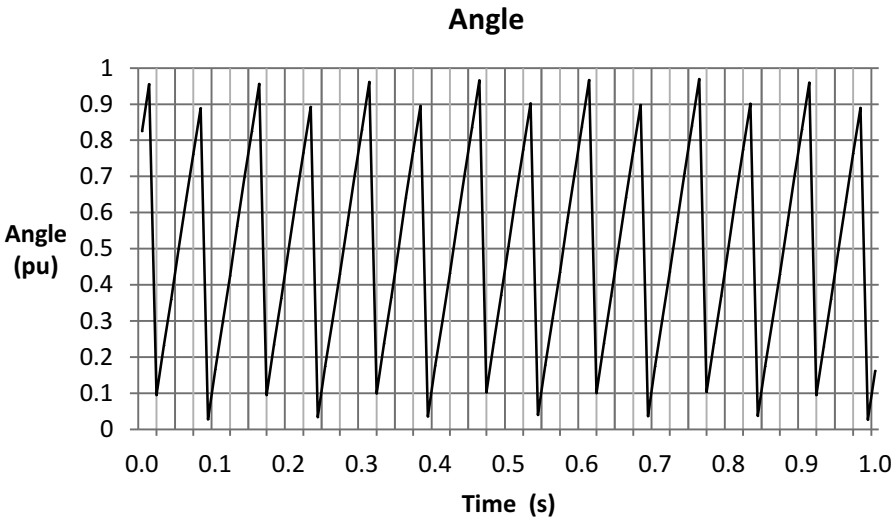


Figure 14-69. Zoom-in on Angle Plot

The estimated speed is around 200 RPM, as shown in Figure 14-70.

Copyright © 2013–2019, Texas Instruments Incorporated

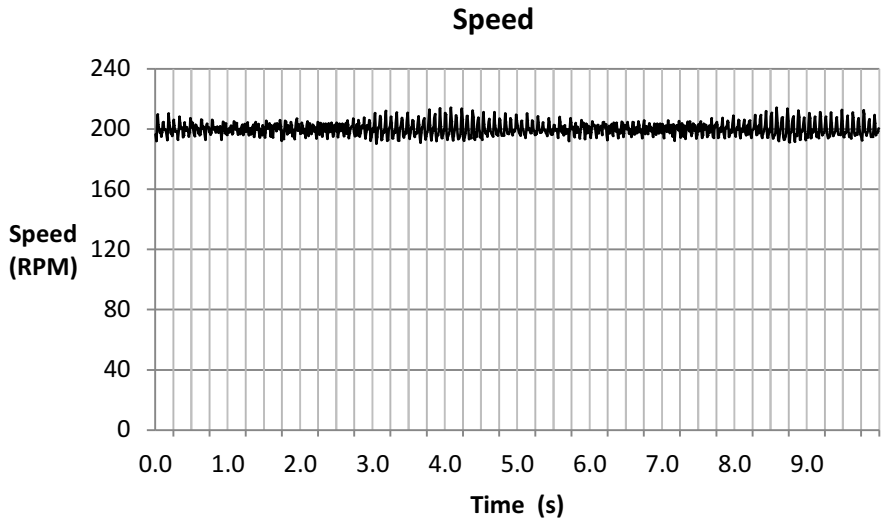


Figure 14-70. Speed Plot

The estimated torque is around 2.5 Nm. Estimated torque, when overloading the motor, is also accurate and fairly constant as can be seen in Figure 14-71.

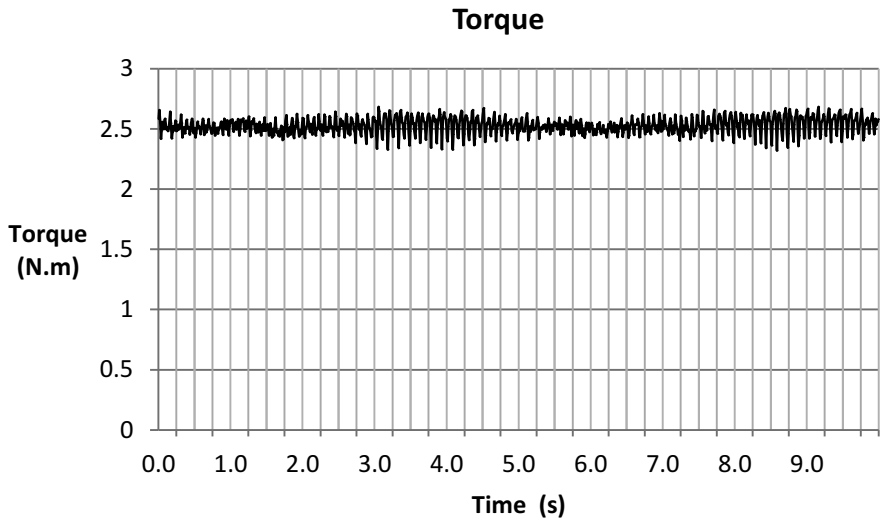


Figure 14-71. Torque Plot

The current in this example is about 5.2 A or so (Figure 14-72), due to the 30% of overloading for a long period of time, the current must be higher for higher stator resistance value in order to produce the commanded torque.

Copyright © 2013–2019, Texas Instruments Incorporated

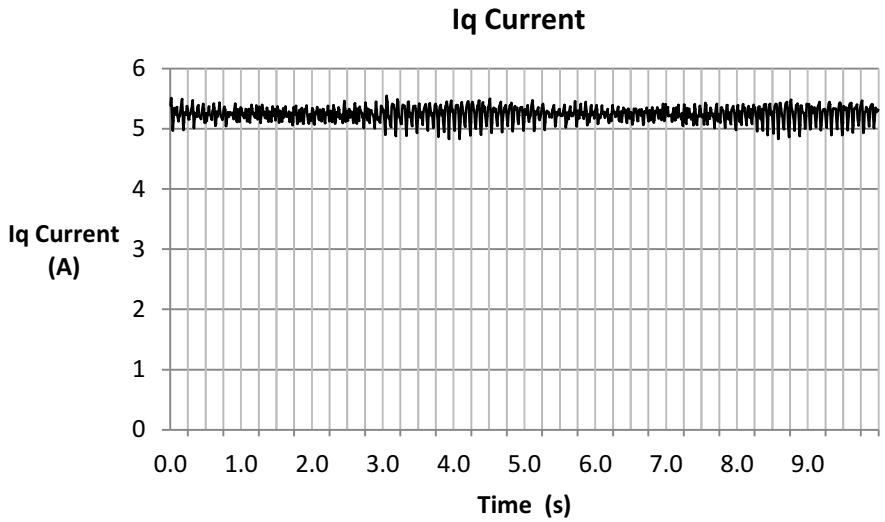


Figure 14-72. Iq Current Plot

14.6 InstaSPIN-MOTION and Low-Speed Considerations

One of the considerations when operating at low speeds is that the speed controller must be tuned to avoid the motor stalling. The SpinTAC speed controller provided in InstaSPIN-MOTION is an advanced speed controller that provides a single parameter tuner and a wide operating range. These combine to make it easy to tune the SpinTAC speed controller to avoid motor stall when operating at low speed. The one important note when tuning the SpinTAC speed controller for low speed operations is that the Bandwidth might need to be set higher than for rated speed operations. This needs to be done so that the SpinTAC speed controller will act more aggressivly to cancel disturbances and regulate low speeds. More information about the SpinTAC speed controller provided in InstaSPIN-MOTION can be found in Chapter 12.

Copyright © 2013–2019, Texas Instruments Incorporated

Rs Online Recalibration

Rs online recalibration is a feature of InstaSPIN-FOC that is used to recalibrate the stator resistance, Rs, while the motor is running in closed loop. The term online in this case is used to describe a system that is running a motor in closed-loop field-oriented control (FOC). This feature is implemented internally within the FAST estimator, and source code is not available, however, all the parameters within this feature can be modified according to applications requirements, as it will be explained in this document. Figure 15-1 shows the FAST estimator, highlighting the interface areas related to Rs Online.

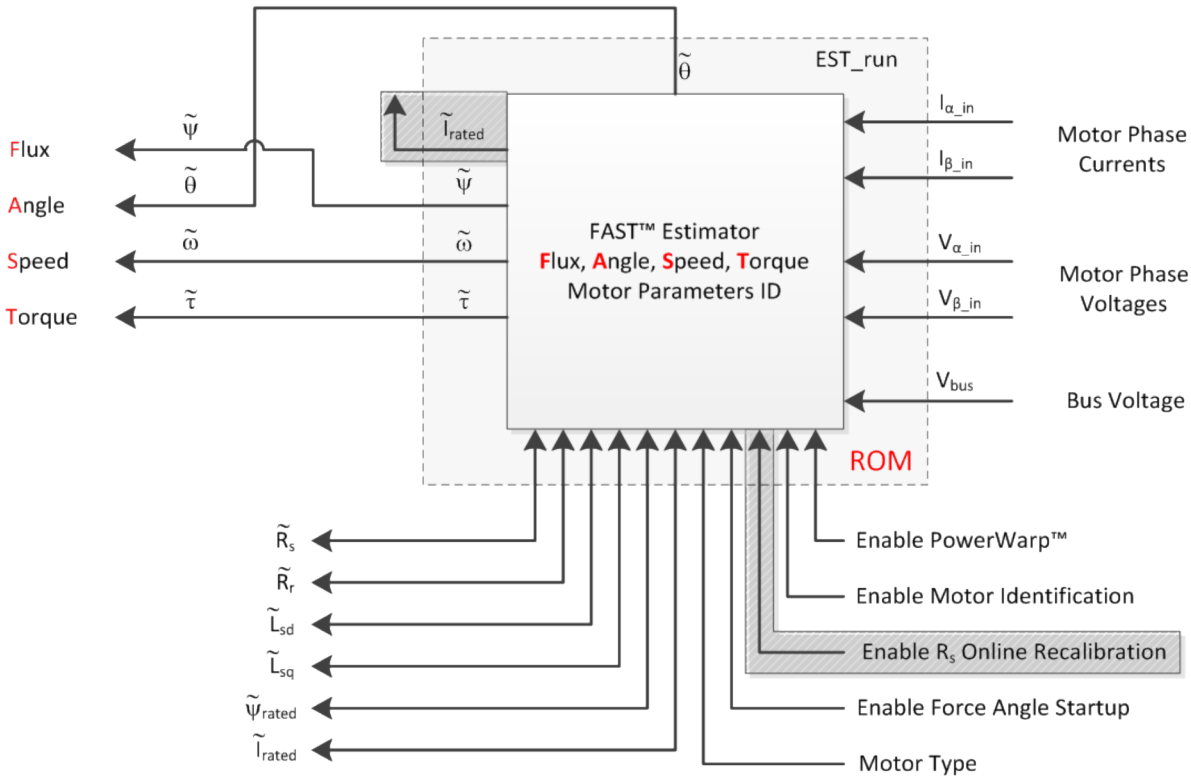


Figure 15-1. FAST Estimator - Rs Online Highlighted

Copyright © 2013–2019, Texas Instruments Incorporated

Topic **Page**

15.1 Overview 529
15.2 Resistance vs. Temperature 529
15.3 Accurate Rs Knowledge Needed at Low Speeds Including Startup 529
15.4 Introduction to Rs Online Recalibration 530
15.5 Rs Online vs. Rs Offline 532
15.6 Enabling Rs Online Recalibration 534
15.7 Disabling Rs Online Recalibration 536
15.8 Modifying Rs Online Parameters 536
15.9 Monitoring Rs Online Resistance Value 545
15.10 Using the Rs Online Feature as a Temperature Sensor 546
15.11 Rs Online Related State Diagrams (CTRL and EST) 546

Copyright © 2013–2019, Texas Instruments Incorporated

15.1 Overview

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.

15.2 Resistance vs. Temperature

A common material for the windings is copper. The following equation represents a linear approximation of the resistance and temperature relationship:

$$R = R_0[1 + \alpha(T - T_0)] \tag{80}$$

Where:

- R: Resistance in Ohms at temperature T, in Ohms (Ω)
- R_0: Resistance in Ohms at temperature T_0, in Ohms (Ω)
- α: Temperature coefficient of the material, in inverse Celsius ($^oC^{-1}$)
- T: Final temperature of the material, in Celsius (^oC)
- T_0: Reference temperature of the material, in Celsius (^oC)

For example, consider a stator resistance, Rs, to be 10 Ω at 20^{o}C, and the windings are made out of copper, with temperature coefficient of 0.00393$^oC^{-1}$. If the motor heats up to 150^{o}C, the new stator resistance will be:

$$R = R_0[1 + \alpha(T - T_0)] \tag{81}$$

$$R = 10\ \Omega[1 + 0.00393^oC^{-1}(150^oC - 20^oC)] \tag{82}$$

$$R = 15.109\ \Omega \tag{83}$$

As can be seen, there is a significant resistance change depending on the temperature, in the example, almost 52% percent increase.

15.3 Accurate Rs Knowledge Needed at Low Speeds Including Startup

The motor model used for FAST is affected by this resistance change, especially at low speeds. This is because at low speeds the majority of the voltage drop inside of the motor model is governed by the stator resistance and the DC component of the current:

$$R_s i_s \tag{84}$$

On the other hand, stator resistance changes outside of the low speed range do not affect the performance of the motor model significantly since at medium to high speeds the internal voltage drop of the model is governed by the back EMF and the inductance times the derivative of the current, or:

$$L_s \frac{di_s}{dt} + e_s \tag{85}$$

It is then required for low speed performance to have an accurate knowledge of the stator resistance, especially when operating at full loads, including starting up the motor from stand still at full load. The following section will introduce the use of Rs Online recalibration in the context of InstaSPIN-FOC.

Copyright © 2013–2019, Texas Instruments Incorporated

15.4 Introduction to Rs Online Recalibration

Rs Online recalibration is added to InstaSPIN-FOC to provide an accurate stator resistance while the motor is operating in closed loop. The updates of the resistance are done in real time, and the motor's model is updated according to the new resistance, providing the best performance results when the motor is running in the entire operating range, from no load, up to full load capability of the motor.

Taking a closer look to the InstaSPIN-FOC block diagram, Rs Online is enabled by setting a flag. Stator resistance is measured while the motor is running through a current injection on the direct component of the current, also known as D-axis current. Figure 15-2 highlights the areas used for Rs Online recalibration.

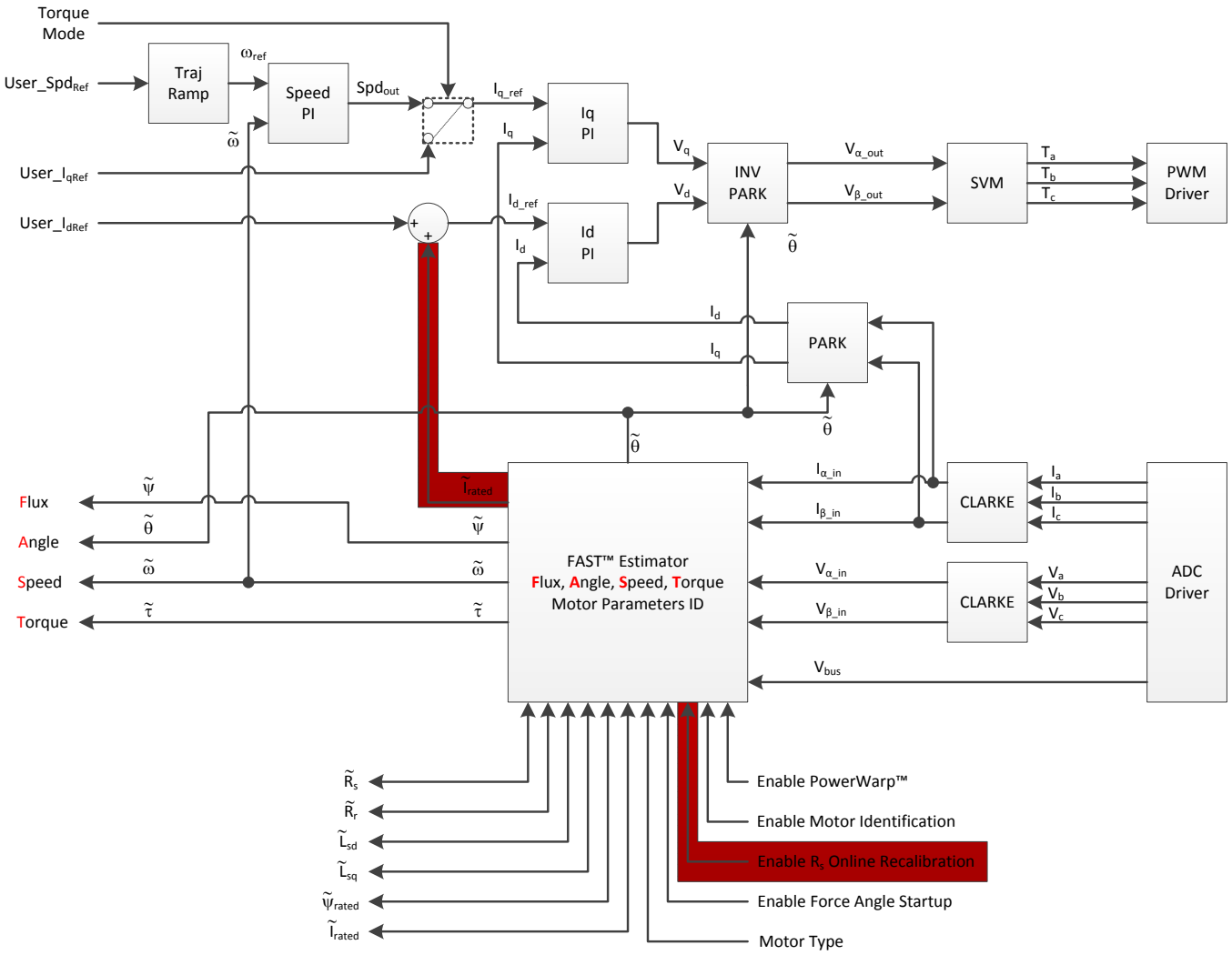


Figure 15-2. Rs Online Recalibration

As can be seen from the block diagram, the online resistance recalibration is done by adding an additional component to I_{rated}. This addition is performed within the FAST estimator, and I_{rated} that comes out of FAST already contains the current needed for Rs Online recalibration. This I_{rated} can be zero for permanent magnet motors, or the magnetizing current for induction motors. In the case of field weakening or field boosting, another current can be added as User_I_{dRef} and this additional current does not interfere with Rs Online recalibration.

Copyright © 2013–2019, Texas Instruments Incorporated

An important point to notice is that the Rs Online recalibration is calculated from the alternating value of the additional Id injected by the FAST module. These currents are alternating between positive and negative to allow the internal algorithm to function. In addition to that, the user can still command a user's I_d reference on top of the value coming from FAST. A typical scenario would be for example to have a negative I_d reference to operate in field weakening, and at the same time, have FAST provide a new I_{rated} to compute the online resistance recalibration. This is expected to be a typical use case and it would work giving expected results from both field weakening and online resistance recalibration.

Due to the current injection done, the phase current waveform going to the motor loses it sinusoidal shape depending on the ratio between the injected current and the mechanical load present on the shaft. For light loads, the sinusoidal shape is greatly affected, and for partial to full load the current shape change is barely perceivable. A few plots are shown below, with a motor running at 500 RPM and Rs Online enabled and disabled at various mechanical loads. It can be seen that in some cases the shape of the phase currents lose their sinusoidal shape.

Figure 15-3 shows that currents are sinusoidal at light loads when Rs Online is disabled.

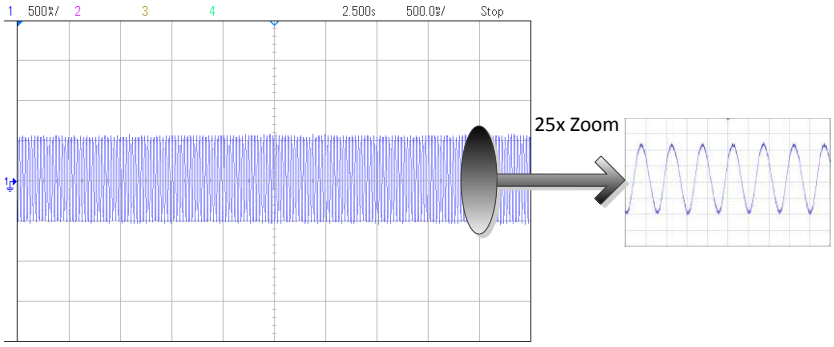


Figure 15-3. Phase Currents at Light Loads - Rs Online Disabled

Under the same mechanical loading conditions, when Rs Online is enabled it can be seen in Figure 15-4 how the currents shape is a distorted sinusoidal waveform.

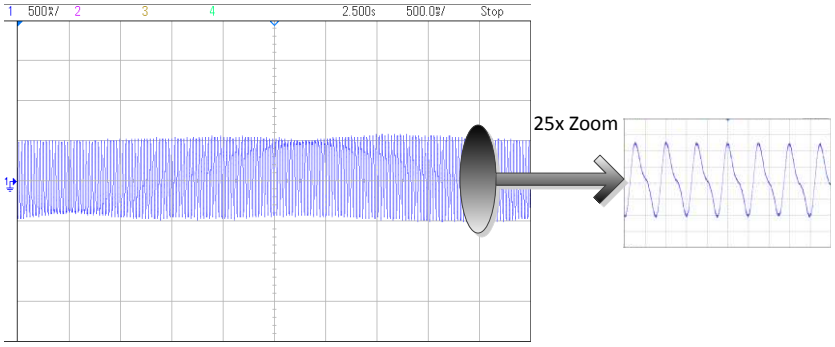


Figure 15-4. Phase Currents at Light Loads - Rs Online Enabled

On the other hand, a motor mechanically loaded, with and without Rs Online, the shape change is difficult to perceive. Figure 15-5 shows when Rs Online is disabled and there is a mechanical load present.

Copyright © 2013–2019, Texas Instruments Incorporated

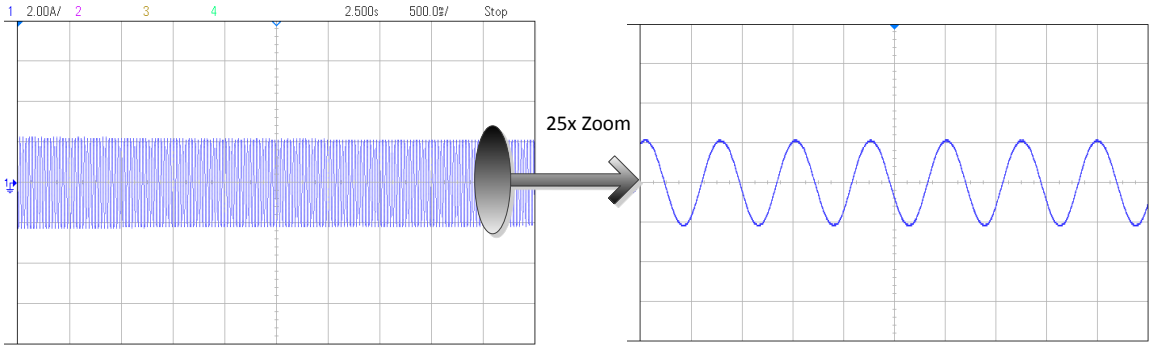


Figure 15-5. Phase Currents with Mechanical Load - Rs Online Disabled

And these are the currents under mechanical load and Rs Online enabled, as shown in Figure 15-6.

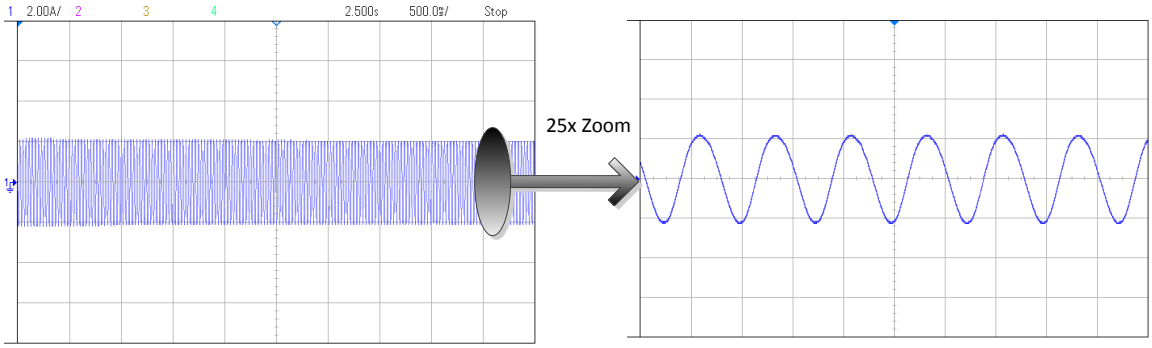


Figure 15-6. Phase Currents with Mechanical Load - Rs Online Enabled

As can be seen on the last plot, even though Rs Online is enabled, at medium to high loads the distortion is much less, and as the load increases, the distortion is no longer perceivable. In the following sections of this document, it will be shown how low the additional current is to allow Rs Online recalibration.

15.5 Rs Online vs. Rs Offline

InstaSPIN-FOC includes another resistance recalibration, which is done before the motor is spun, known as Rs Offline. Rs Offline requires the motor to be at standstill, injecting a DC current into I_d. On the other hand, Rs Online requires the motor to be spinning in order to recalibrate the resistance, injecting an AC current into I_d.

Both Rs Offline and Rs Online are critical portions of InstaSPIN-FOC to provide the best low speed performance. In a typical application, Figure 15-7 shows the use of both Rs offline and Rs online.

Copyright © 2013–2019, Texas Instruments Incorporated

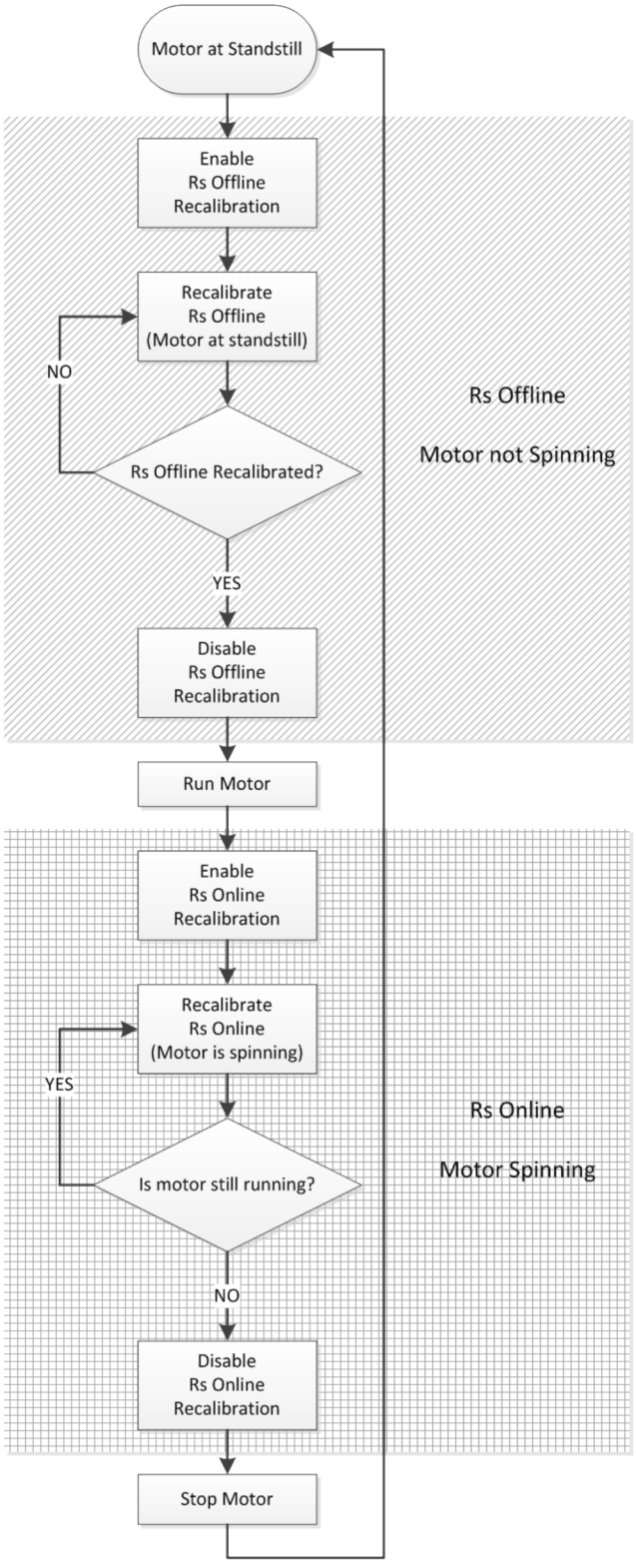


Figure 15-7. Rs Online and Rs Offline Flowchart

Copyright © 2013–2019, Texas Instruments Incorporated

15.6 Enabling Rs Online Recalibration

In order to enable Rs Online, a couple of parameters need to be setup. One of the most important parameters is how much current will be injected into the D-axis current (Id) in order to perform Rs Online recalibration. Usually, what is recommended is to have a minimum of 5% of the rated current in order to have measurable current to get a good recalibration of the resistance while the motor is running.

```
_iq RsOnLineCurrent_A = _IQ(USER_MOTOR_MAX_CURRENT * 0.05);
```

Note that the multiplication is done by the pre-compiler, using a floating point for the USER_MOTOR_MAX_CURRENT define, times 0.05, representing the 5%, and then converting the floating point results into a global IQ value. For more information about the IQmath library, see the *C28x IQMath Library – A Virtual Floating Point Engine – Module User's Guide* (SPRC990)

Also, before enabling Rs Online, user must set:

- Initial Q format value of the resistance representation. This is done by using the function: EST_setRsOnLine_qFmt (). The initial value must be taken from the value measured by the Rs Offline, while the motor is at stand still. This value can be read by calling the following function: EST_getRs_qFmt ().
- Id magnitude used for Rs Online set to zero. Done by using function: EST_setRsOnLineId_mag_pu ()
- Id in per-units value set to zero. Done by using function: EST_setRsOnLineId_pu ()
- Both enable Flag and update flag to FALSE, done by calling these two functions: EST_setFlag_enableRsOnLine () and EST_setFlag_updateRs ().

Both Id_mag_pu and Id_pu values need to be set to zero in order to prevent the estimator to keep any residual current references when it is disabled. In future releases of InstaSPIN, this will not be needed, and only the Id_mag_pu will be required to be reset to zero, but for the 2806xF devices, both require a value of zero to be written before Rs Online is enabled, or right before Rs Online is disabled.

The two flags perform different tasks inside the estimator. The enableRsOnLine flag allows the entire Rs Online feature to run, updating an internal variable holding the most recent resistance value, and injecting current into Id. The second flag, updateRs, allows the resistance value to be used by the motor model. If the updateRs flag is never set, but the enableRsOnLine flag is set, the resistance can still be used to monitor how the resistance changes, but the internal motor model will not use this varying resistance. If the motor temperature increases drastically, and the resistance is not updated in the motor model (by setting the updateRs flag to TRUE), the performance of InstaSPIN will be affected, and the low speed performance will not be as desired. Also, the motor might not startup under full load.

The following code example shows how to set the initial values as well as how to check the condition to make sure the initial values are set when the state machine is in the proper state. This is done prior to enabling Rs Online recalibration:

```
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;

// get the controller state
gMotorVars.CtrlState = CTRL_getState(ctrlHandle);

// get the estimator state
gMotorVars.EstState = EST_getState(obj->estHandle);

if((gMotorVars.CtrlState <= CTRL_State_OffLine) ||
       ((gMotorVars.CtrlState == CTRL_State_OnLine)    &&
        (gMotorVars.EstState == EST_State_Rs)))
  {
   EST_setRsOnLine_qFmt(obj->estHandle,EST_getRs_qFmt(obj->estHandle));
   EST_setRsOnLineId_mag_pu(obj->estHandle,_IQ(0.0));
   EST_setRsOnLineId_pu(obj->estHandle,_IQ(0.0));
   EST_setFlag_enableRsOnLine(obj->estHandle,FALSE);
   EST_setFlag_updateRs(obj->estHandle,FALSE);
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
  }
```

This code example can be executed outside of the interrupt, in the main forever loop, since it only involves global variables and no time critical code execution. Keep in mind that the code inside the "if" condition is executed before the Rs Online is enabled.

One of the conditions to reset all the parameters of Rs Online as shown in the previous code example is when CtrlState is less than or equal to CTRL_State_OffLine. This condition means that the Rs Online should be disable and reset when the state machine is either Idle (motor not being energized), or when doing the offsets recalibration. The other state where these initial values have to be done is when the control state is Online (CTRL_State_OnLine), and the estimator state is EST_State_Rs, in other words, when the Rs Offline recalibration is being done. All these conditions represent a motor at stand still. In order to relate these states to the entire state machine within InstaSPIN, see Chapter 6. The CTRL_State_OffLine state is shown as ***Offline*** in the CTRL state machine diagram, the CTRL_State_OnLine state is shown as ***Online*** in the CTRL state machine diagram and the EST_State_Rs state is shown as ***Rs*** in the EST state machine diagram

When the motor is running, we need to make sure that the resistance given by the Rs Online recalibration feature is close enough to the initial resistance, given by the Rs Offline feature. This is done to ensure a smooth transition between both resistance values causing no disturbances to the closed loop system. This can be done with the following code example, which includes the "else" condition from the previous code example. The following condition will be executed whenever the motor is not at standstill, in other words, when the motor is spinning. This condition will be used to enable Rs Online as shown below.

```
else
  {

   // Scale factor to convert Amps to per units.
   // USER_IQ_FULL_SCALE_CURRENT_A is defined in user.h
   _iq sf = _IQ(1.0/USER_IQ_FULL_SCALE_CURRENT_A);

   Rs_pu = EST_getRs_pu(obj->estHandle);
   RsOnLine_pu = EST_getRsOnLine_pu(obj->estHandle);
   Rs_error_pu = RsOnLine_pu -    Rs_pu;

   EST_setFlag_enableRsOnLine(obj->estHandle,TRUE);
   EST_setRsOnLineId_mag_pu(obj->estHandle,_IQmpy(RsOnLineCurrent_A,sf));

   // Enable updates when Rs Online is only 5% different from    Rs Offline
   if(_IQabs(Rs_error_pu) <    _IQmpy(Rs_pu,_IQ(0.05)))
     {
      EST_setFlag_updateRs(obj->estHandle,TRUE);
     }
  }
```

Notice that in this example we are enabling the Rs Online recalibration by calling:

```
EST_setFlag_enableRsOnLine(obj->estHandle,TRUE)
```


Copyright © 2013–2019, Texas Instruments Incorporated

However, the Rs Online value will not be updated until the update flag is set to TRUE. The update flag is set to TRUE when the Rs Online value is within a desired proximity to the initial value provided by Rs Offline, in this case 5% of that value. For applications that require full torque at start up, it is recommended to have a smaller percentage of difference between Rs Online and Rs Offline, i.e. 3% or so instead of 5%. As soon as the Rs Online value and Rs Offline value are within 5% difference, the Rs Online update flag is set by calling:

```
EST_setFlag_updateRs(obj->estHandle,TRUE)
```

Once both flags are enabled, Rs Online recalculates the resistance in real time, and the estimator will update its internal motor model according to that new resistance.

Also, as noted in the code example, the magnitude of the current to be injected in order to estimate the resistance Online is set by calling the following function:

```
EST_setRsOnLineId_mag_pu(obj->estHandle,_IQmpy(RsOnLineCurrent_A,sf));
```

15.7 Disabling Rs Online Recalibration

In order to disable Rs Online recalibration the user can refer to the first code example, listed here also, where the initial values are written and flags are disabled. This code example, as discussed previously, also checks for the right state of the state machine in order to disable Rs Online appropriately:

```
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;

// get the controller state
gMotorVars.CtrlState = CTRL_getState(ctrlHandle);

// get the estimator state
gMotorVars.EstState = EST_getState(obj->estHandle);

if((gMotorVars.CtrlState <= CTRL_State_OffLine) ||
       ((gMotorVars.CtrlState == CTRL_State_OnLine)    &&
        (gMotorVars.EstState == EST_State_Rs)))
  {
   EST_setRsOnLine_qFmt(obj->estHandle,EST_getRs_qFmt(obj->estHandle));
   EST_setRsOnLineId_mag_pu(obj->estHandle,_IQ(0.0));
   EST_setRsOnLineId_pu(obj->estHandle,_IQ(0.0));
   EST_setFlag_enableRsOnLine(obj->estHandle,FALSE);
   EST_setFlag_updateRs(obj->estHandle,FALSE);
  }
```

15.8 Modifying Rs Online Parameters

Several parameters can be tuned and modified within the Rs Online feature of InstaSPIN-FOC. The following list of parameters will be discussed in further detail in this section:

- Injected Current Magnitude
- Slow Rotating Angle
- Delta Increments and Decrements of the Rs Online Value
- Filter Parameters

Copyright © 2013–2019, Texas Instruments Incorporated

15.8.1 Adjusting Injected Current Magnitude

The first parameter described in this section is the current injected in the D-axis (Id) to allow an Rs Online recalibration. This current is generated by the estimator module itself when the Rs Online is enabled, and its magnitude is user configurable. As discussed earlier, the recommended value for the injected current is around 5% of the motor's rated current to allow a measurable current back from the motor and hence allow an accurate Rs Online recalibration. For example, consider a motor with a rated current of 5 A. The injected current in this scenario is 0.25 A. The following code example sets the injected current for Rs Online to be 0.25 A:

```
// Scale factor to convert Amps to per units.
// USER_IQ_FULL_SCALE_CURRENT_A is defined in user.h
_iq sf = _IQ(1.0/USER_IQ_FULL_SCALE_CURRENT_A);

// Value corresponding to 0.25 Amps
_iq RsOnLineCurrent_A = _IQ(0.25);

CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;

// Scale value from Amps to per units and set through the use of an API
EST_setRsOnLineId_mag_pu(obj->estHandle,_IQmpy(RsOnLineCurrent_A,sf));
```

When Rs Online is running, Figure 15-8 shows how the amplitude in the current waveform can be seen, and measured to be approximately 0.25 A when no mechanical load is applied to the motor. As can be seen, the no load current in this case is 0.1 A, and when adding a 0.25 A for Rs Online, the peak current is $I_s = \sqrt{I_q^2 + I_d^2} = \sqrt{0.1^2 + 0.25^2} = 0.27\,A$

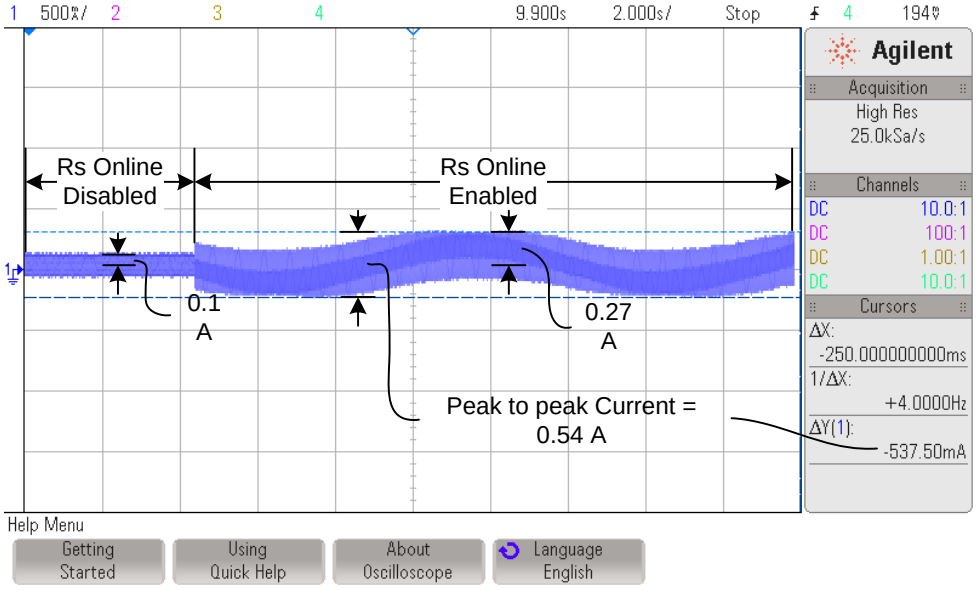


Figure 15-8. Result of Adding 0.25 A for Rs Online

As the load of the motor increases, the additional current required for Rs Online becomes proportionally smaller as can be seen in Figure 15-9. In this case we have a load current of 0.35 A, and we are keeping an Rs Online injection current of 0.25 A. Using the same equation as in previous example, the maximum current is: $I_s = \sqrt{I_q^2 + I_d^2} = \sqrt{0.35^2 + 0.25^2} = 0.43\,A$

Copyright © 2013–2019, Texas Instruments Incorporated

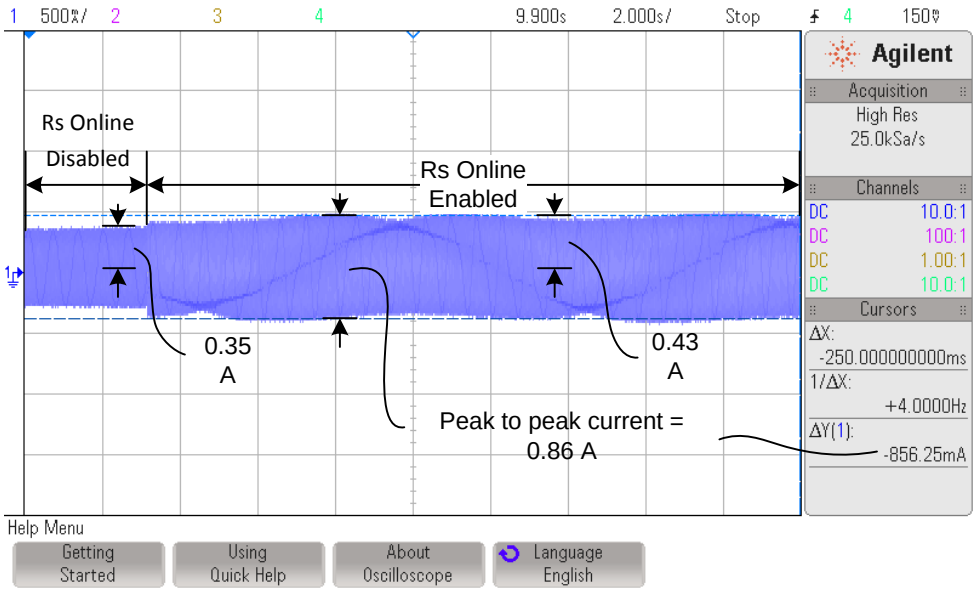


Figure 15-9. Result of Increasing Load for Rs Online

Another example is if a motor is 10 A, and the injected current for Rs Online is 0.5 A. The same code example with a different current value is used:

```
// Scale factor to convert Amps to per units.
// USER_IQ_FULL_SCALE_CURRENT_A is defined in user.h
_iq sf = _IQ(1.0/USER_IQ_FULL_SCALE_CURRENT_A);

// Value corresponding to 0.5 Amps
_iq RsOnLineCurrent_A = _IQ(0.5);

CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;

// Scale value from Amps to per units and set through the use of an API
EST_setRsOnLineId_mag_pu(obj->estHandle,_IQmpy(RsOnLineCurrent_A,sf));
```

Similar to the previous examples, the maximum current can be calculated by using the same equation:

$$I_s = \sqrt{I_q^2 + I_d^2} = \sqrt{0.6^2 + 0.5^2} = 0.78\,A \tag{86}$$

And the corresponding oscilloscope plot in this example is shown in Figure 15-10.

Copyright © 2013–2019, Texas Instruments Incorporated

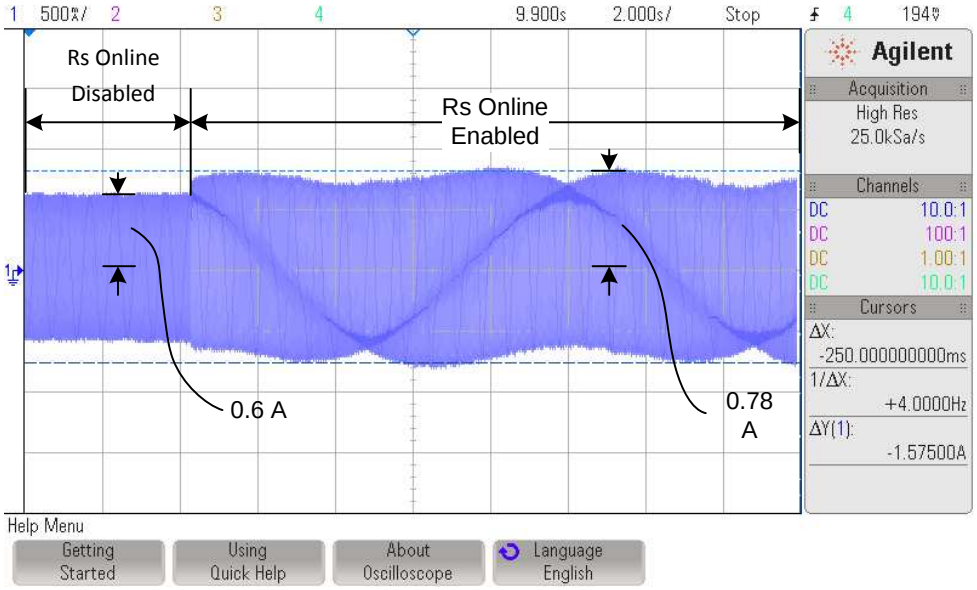


Figure 15-10. Maximum Current With Rs Online Enabled

In general, if we consider a case where the motor is fully loaded, in the case of a 10 A motor, and adding 5% of that current for Rs Online recalibration, the total maximum current will be:

$$I_s = \sqrt{I_q^2 + I_d^2} = \sqrt{10.0^2 + 0.5^2} = 10.0125\,A \quad (87)$$

In other words, the additional current supplied to the motor will only be 0.0125 A, representing only 0.125 % of the rated current. This current is usually not even perceived by the motor in terms of additional heating.

For example, the following scenario is for a 2.2 A motor, with an Rs Online current of 5% of the rated current, equal to 2.2*0.05 = 0.11 A. The additional current is only 0.125% compared to the rated current, which equals to 2.2*0.00125 = 0.0028 A. When we plot before and after Rs Online has been enabled, the additional current is not even noticeable on the oscilloscope (Figure 15-11). It is also important for the reader to notice that the vertical zoom of the oscilloscope changed from 500mA/ to 2.00A/ since now the amplitudes of the current are much higher than before due to the mechanical loading of the motor.

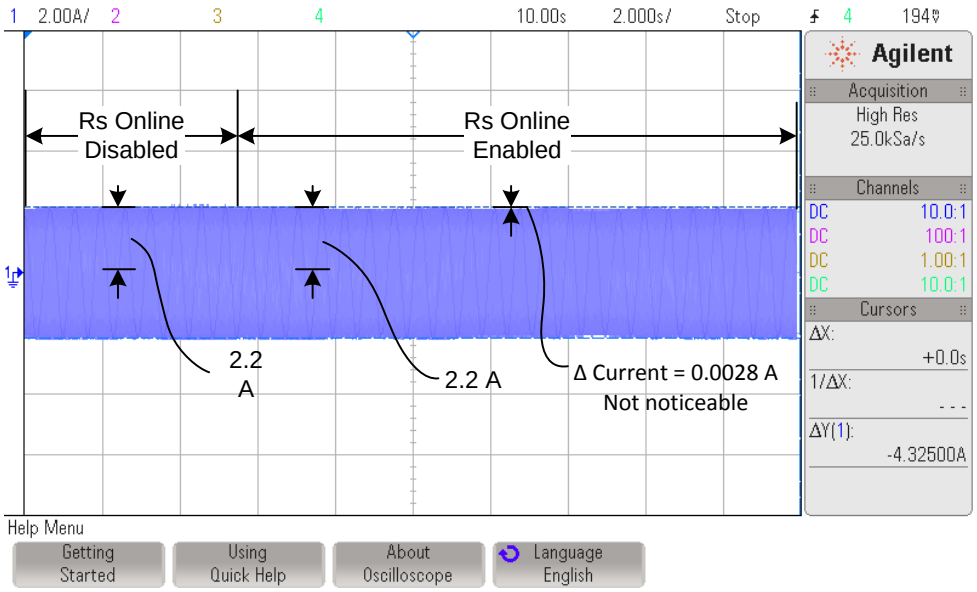


Figure 15-11. 2.2-A Motor With an Rs Online Current of 5%

Copyright © 2013–2019, Texas Instruments Incorporated

15.8.2 Adjusting Slow Rotating Angle

The implementation of the Rs Online feature requires a certain internal vector to be rotated slowly as the motor spins. This slowly rotating vector is set by default to a value of 0.00001, in per unit value, so it will generate a vector at a frequency of 0.00001 * Estimator Frequency in Hz. So if the estimator frequency is 10 kHz, then the rotating vector will be at a frequency of 0.1 Hz, or with a period of 10 seconds. This rotating vector is needed so that the Rs Online converges to an average resistance measured at different points of that vector.

Although the details are not disclosed on how Rs Online estimates a varying resistance, it is important to know that this rotating angle is used to estimate a resistance at various current vectors. Over time, the online resistance is the average resistance produced by the measurements at all the vectors as they are rotated slowly. In order for the user to know what the rotating vector is set to, besides looking at the currents in the oscilloscope, users can use the following code example.

```
// These defines are in user.h
#define USER_NUM_ISR_TICKS_PER_CTRL_TICK (1)
#define USER_NUM_CTRL_TICKS_PER_EST_TICK (1)
#define USER_PWM_FREQ_kHz (10.0)
#define USER_ISR_FREQ_Hz (USER_PWM_FREQ_kHz * 1000.0)
#define USER_CTRL_FREQ_Hz (uint_least32_t)(USER_ISR_FREQ_Hz \
                          /USER_NUM_ISR_TICKS_PER_CTRL_TICK)
#define USER_EST_FREQ_Hz (uint_least32_t)(USER_CTRL_FREQ_Hz \
                          /USER_NUM_CTRL_TICKS_PER_EST_TICK)

// Initialize obj to the controller handle
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;

_iq delta_pu_to_kHz_sf = _IQ((float_t)USER_EST_FREQ_Hz/1000.0);
_iq RsOnLine_Angle_Delta_pu = EST_getRsOnLineAngleDelta_pu(obj->estHandle);

// By default, the returned value in the following line will be close to:
// _IQ(0.00001), representing 0.0001 kHz, or 0.1 Hz
_iq RsOnLine_Angle_Freq_kHz = _IQmpy(RsOnLine_Angle_Delta_pu, \
                              delta_pu_to_kHz_sf);
```

For more information about the software execution clock trees used in InstaSPIN, as well as the decimation factors, also known as tick rates, see Chapter 9.

If the angle delta is never changed, a default value of 0.00001, in per unit value, is set by the library, which results in a slow rotating angle frequency of 0.00001 times the estimation frequency. If the estimation frequency is the same as the PWM frequency, and it is set to 10 kHz, then the slow rotating angle will have a frequency of 0.00001 * 10000 = 0.1 Hz (period of 10 seconds). The slow rotating angle can be seen from the current waveform, as the angle changes how the current is injected into I_d. Figure 15-12 shows how the current changes its shape at a frequency equal to the slow rotating angle frequency.

Copyright © 2013–2019, Texas Instruments Incorporated

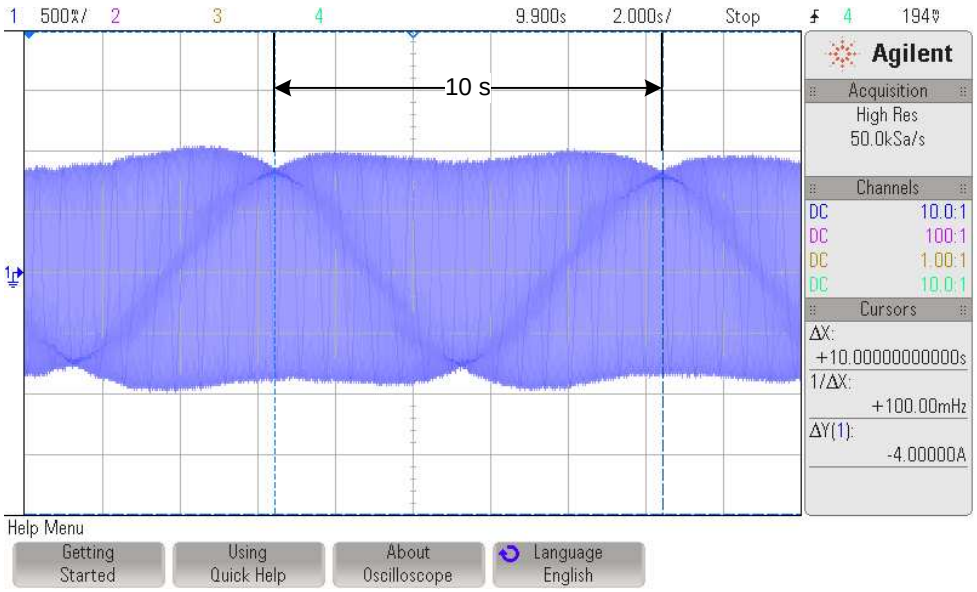


Figure 15-12. Current Shape Changes When Frequency Equals Slow Rotating Angle Frequency

An example of an application that might need a change in this rotating angle is when the motor's temperature increases at a much higher rate, so this rotating angle needs to be faster. The rotating vector does not need to be changed as the temperature increases. It only needs to be set once depending on the expected worst case temperature dynamics of the system, and there is no need to fine tune this value as temperature varies. For example, if the temperature dynamics of a system requires a rotating angle to be changed to 0.2 Hz (period of 5 seconds), the following code example is used to change the slowly rotating angle to the new value of 0.2 Hz:

```
// This new define represents the desired RsOnLine rotating angle frequency
#define RSONLINE_ANGLE_FREQ_Hz (0.2)

// Initialize obj to the controller handle
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;

// The scale factor (sf) calculation is done by the pre-compiler
_iq delta_hz_to_pu_sf = _IQ(1.0/(float_t)USER_EST_FREQ_Hz);
_iq RsOnLine_Angle_Freq_Hz = _IQ(RSONLINE_ANGLE_FREQ_Hz);
_iq RsOnLine_Angle_Delta_pu = _IQmpy(RsOnLine_Angle_Freq_Hz, \
                              delta_hz_to_pu_sf);

EST_setRsOnLineAngleDelta_pu(obj->estHandle,
          RsOnLine_Angle_Delta_pu);
```

As can be noticed in this code example, the function now sets a value for angle delta, so it expects a parameter to be written, which in this case is variable RsOnLine_Angle_Delta_pu.

The resulting oscilloscope plot related to the configuration done in the previous code example is shown in Figure 15-13.

Copyright © 2013–2019, Texas Instruments Incorporated

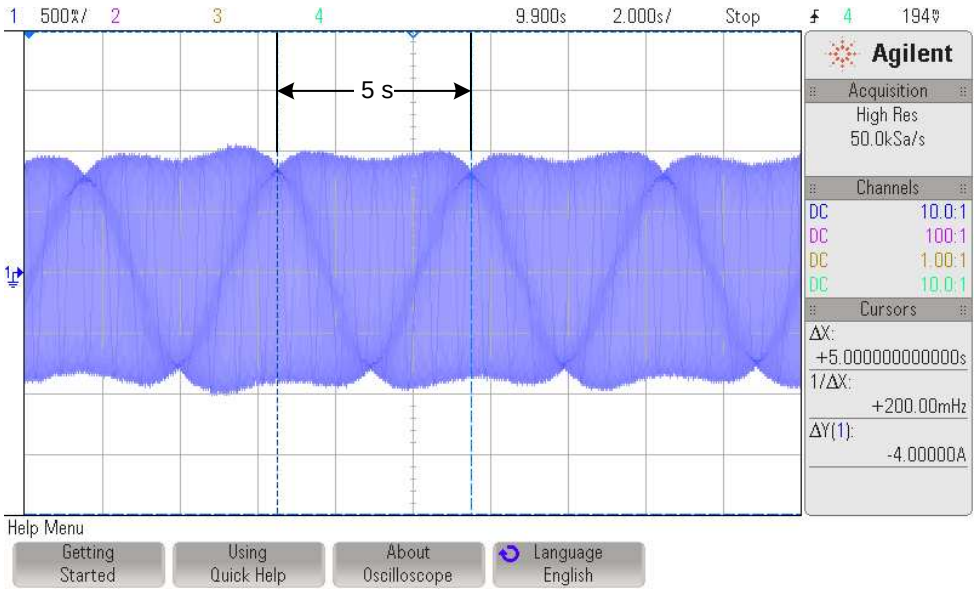


Figure 15-13. Result of RsOnLine_Angle_Delta_pu

15.8.3 Adjusting Delta Increments and Decrements of the Rs Online Value

Inside of the estimator, and in particular, the part of the estimator that runs the Rs Online feature, the actual value of the resistance is updated by adding and/or subtracting fixed delta values, depending on the direction where the resistance is going. In general this parameter does not need to be changed, unless the change in resistance is too fast, for example, due to rapid motor heating. By default both the delta increment and delta decrement are set to a value of 0.00001, represented in IQ30 format, which can be verified with the following code example:

```
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;
_iq30 delta_dec = EST_getRsOnLine_delta_dec_pu(obj->estHandle);
_iq30 delta_inc = EST_getRsOnLine_delta_inc_pu(obj->estHandle);
```

In order to change those deltas, use the following code example, to for example, twice the default value, or 0.00002, in IQ30:

```
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;
EST_setRsOnLine_delta_dec_pu(obj->estHandle, _IQ30(0.00002));
EST_setRsOnLine_delta_inc_pu (obj->estHandle, _IQ30(0.00002));
```

Notice that these two functions set the deltas, as opposed to get the deltas, so they expect a parameter besides the handle.

Figure 15-14 shows how the resistance will respond to an initial value difference according to the delta values. For example, right at the beginning when Rs Online is first enabled, there is an initial resistance value which is different than the steady state value. Having a value of 0.00001 will lead to the following plot, which shows a slope = (0.77-0.4)/3.1 = 0.12 Ω/s.

Copyright © 2013–2019, Texas Instruments Incorporated

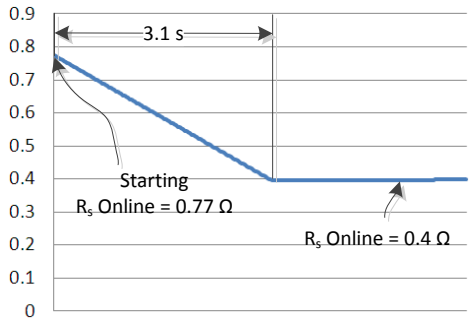


Figure 15-14. Resistance Response to Initial Value Difference

When the delta values are changed to double the default value, a much faster settling time is shown in Figure 15-15, with twice the slope = (0.71 - 0.4) / 1.3 = 0.24 Ω/s.

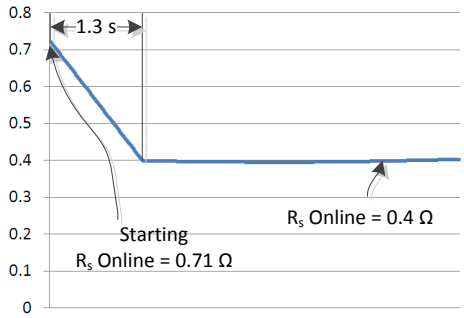


Figure 15-15. Delta Values Changed to Double Default Value

It is recommended that the rate of change selected for this delta is slow enough to provide smooth variations of the resistance, and fast enough to track the temperature changes of the system. Generally the initial value of 0.00001 will work, but keep the deltas in mind when fine tuning a particular application, especially when drastic temperature changes are expected.

15.8.4 Adjusting Filter Parameters

There are two first order cascaded filters inside of the Rs Online estimator which are run in order to get an accurate and steady value of the stator resistance. A total of four filters are run, one set of two cascaded ones for the currents, and one set of two for the voltages. Each set of two filters use the same coefficients by default. However, depending on the response required by the application, the coefficients of both filters can be read and written individually. The default cutoff frequency of these filters is set to 0.2 Hz. This cutoff frequency can be verified by the user with the following code example:

```
EST_getRsOnLineFilterParams(obj->estHandle, EST_RsOnLineFilterType_Current,
                            &pfilter_i0->b0, &pfilter_i0->a1, &pfilter_i0->y1,
                            &pfilter_i1->b0, &pfilter_i1->a1, &pfilter_i1->y1);
EST_getRsOnLineFilterParams(obj->estHandle, EST_RsOnLineFilterType_Voltage,
                            &pfilter_v0->b0, &pfilter_v0->a1, &pfilter_v0->y1,
                            &pfilter_v1->b0, &pfilter_v1->a1, &pfilter_v1->y1);

_iq pu_to_kHz_sf = _IQ((float_t)USER_EST_FREQ_Hz/1000.0);

cutoff_freq_kHz_i0 = _IQmpy(pfilter_i0->b0, pu_to_kHz_sf);
cutoff_freq_kHz_i1 = _IQmpy(pfilter_i1->b0, pu_to_kHz_sf);
cutoff_freq_kHz_v0 = _IQmpy(pfilter_v0->b0, pu_to_kHz_sf);
cutoff_freq_kHz_v1 = _IQmpy(pfilter_v1->b0, pu_to_kHz_sf);
```


Copyright © 2013–2019, Texas Instruments Incorporated

By default, the cutoff frequency variables will return a value of 0.0002, representing 0.0002 kHz, or 0.2 Hz. Generally, this cutoff frequency should be tuned so that the resistance value can grow depending on the expected temperature dynamics of the application. Figure 15-16 is an example of how the Rs Online varies depending on this cut-off frequency. We plot 0.1 Hz, 0.2 Hz and 0.5 Hz. Using a slow rotating angle of 0.1 Hz, if the cutoff frequency of the filters is faster than the rotating angle, the resistance will tend to follow the rotating angle, so it is recommended to have this filter setting to at least the same frequency of the slow rotating angle to get a better filtered resistance.

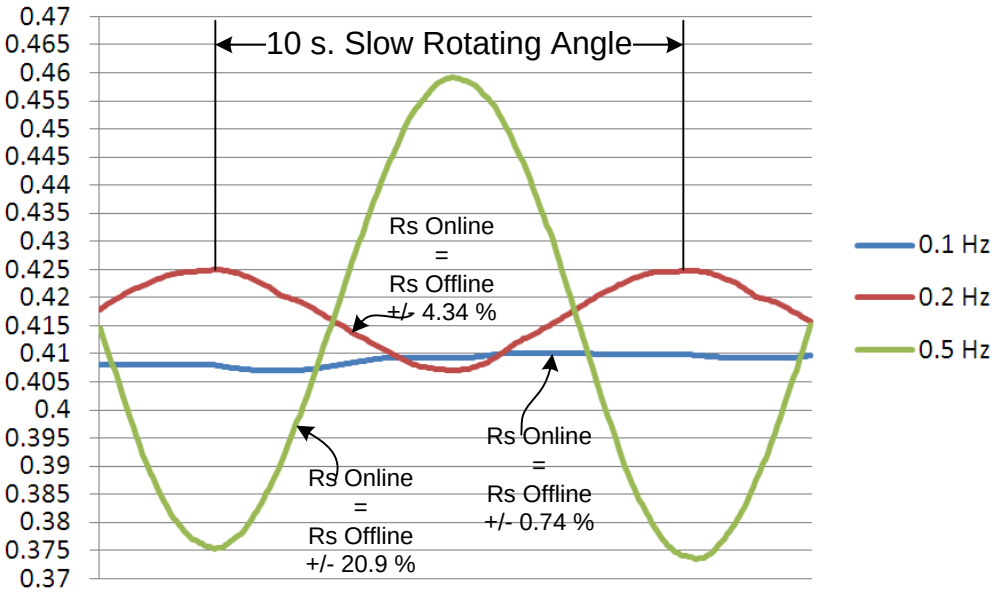


Figure 15-16. Rs Online Varies Depending on Cut-Off Frequency

As can be seen, having a lower cutoff frequency leads to a less varying resistance. However, if the temperature dynamics of the system allow the temperature to rise too rapid, having a low cutoff frequency might be a problem.

There can be a scenario where the temperature increases very rapidly, requiring a change in the cutoff frequency to a higher value. If the application requires a change in the cutoff frequency, the following code example shows how to do this. In the example we are changing the cutoff frequency on the fly, so we have to read it first, then modify the coefficients, and then write back with the same output, so we avoid affecting the filters' outputs.

```
EST_getRsOnLineFilterParams(obj->estHandle, EST_RsOnLineFilterType_Current,
                            &pfilter_i0->b0, &pfilter_i0->a1, &pfilter_i0->y1,
                            &pfilter_i1->b0, &pfilter_i1->a1, &pfilter_i1->y1);
EST_getRsOnLineFilterParams(obj->estHandle, EST_RsOnLineFilterType_Voltage,
                            &pfilter_v0->b0, &pfilter_v0->a1, &pfilter_v0->y1,
                            &pfilter_v1->b0, &pfilter_v1->a1, &pfilter_v1->y1);

// Use global variable desired_frequency_kHz to set the desired cutoff
// frequency of the filters. Use the same cutoff frequency for all filters
cutoff_freq_kHz_i0 = desired_frequency_kHz;
cutoff_freq_kHz_i1 = cutoff_freq_kHz_i0;
cutoff_freq_kHz_v0 = cutoff_freq_kHz_i0;
cutoff_freq_kHz_v1 = cutoff_freq_kHz_i0;

// Use the following scale factor to convert kHz to per unit value
_iq kHz_to_pu_sf = _IQ(1000.0/(float_t)USER_EST_FREQ_Hz);

// Calculate the per unit value for all filters
cutoff_freq_pu_i0 = _IQmpy(cutoff_freq_kHz_i0, kHz_to_pu_sf);
cutoff_freq_pu_i1 = _IQmpy(cutoff_freq_kHz_i1, kHz_to_pu_sf);
cutoff_freq_pu_v0 = _IQmpy(cutoff_freq_kHz_v0, kHz_to_pu_sf);
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
cutoff_freq_pu_v1 = _IQmpy(cutoff_freq_kHz_v1, kHz_to_pu_sf);

// Calculate coefficients for all filters
pfilter_i0->b0 = cutoff_freq_pu_i0;
pfilter_i0->a1 = cutoff_freq_pu_i0 - _IQ(1.0);
pfilter_i1->b0 = cutoff_freq_pu_i1;
pfilter_i1->a1 = cutoff_freq_pu_i1 - _IQ(1.0);
pfilter_v0->b0 = cutoff_freq_pu_v0;
pfilter_v0->a1 = cutoff_freq_pu_v0 - _IQ(1.0);
pfilter_v1->b0 = cutoff_freq_pu_v1;
pfilter_v1->a1 = cutoff_freq_pu_v1 - _IQ(1.0);

// Configure Rs Online to use the new filter coefficients
EST_setRsOnLineFilterParams(obj->estHandle, EST_RsOnLineFilterType_Current,
                            pfilter_i0->b0, pfilter_i0->a1, pfilter_i0->y1,
                            pfilter_i1->b0, pfilter_i1->a1, pfilter_i1->y1);
EST_setRsOnLineFilterParams(obj->estHandle, EST_RsOnLineFilterType_Voltage,
                            pfilter_v0->b0, pfilter_v0->a1, pfilter_v0->y1,
                            pfilter_v1->b0, pfilter_v1->a1, pfilter_v1->y1);
```

15.9 Monitoring Rs Online Resistance Value

There are two ways of checking the Rs Online resistance in a watch window or in a global variable. This might be done as per user's requirements, so set thresholds on temperature, or simply to make sure there is a correct connection of the motor to the system.

15.9.1 Rs Online Floating Point Value

The first method is to simply call a function that returns a value in floating point. This would represent the resistance value in ohms (Ω), and this is a use example:

```
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;
float_t RsOnLine_Ohm = EST_getRsOnLine_Ohm(obj->estHandle);
```

This method is convenient for general monitoring. However it performs a few floating point instructions which won't be the most efficient way to do it in terms of execution time.

15.9.2 Rs Online Fixed Point Value

The second method to monitor Rs Online value is using a more detailed approach, but only using fixed point math, and bit shifting. The following code example shows how an Rs Online value in Ohms but in fixed point can be calculated from the functions available in InstaSPIN-FOC. This can be calculated even inside an interrupt, since execution time is optimized by avoiding floating point math.

```
#define VarShift(var,nshift) (((nshift) < 0) ? ((var)>>(-(nshift))) \
                                             : ((var)<<(nshift)))
#define USER_IQ_FULL_SCALE_VOLTAGE_V (48.0)
#define USER_IQ_FULL_SCALE_CURRENT_A (40.0)

CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;
uint_least8_t RsOnLine_qFmt = EST_getRsOnLine_qFmt(obj->estHandle);
_iq fullScaleResistance = _IQ(USER_IQ_FULL_SCALE_VOLTAGE_V \
                            /USER_IQ_FULL_SCALE_CURRENT_A);
_iq RsOnLine_pu = _IQ30toIQ(EST_getRsOnLine_pu(obj->estHandle));
_iq pu_to_ohms_sf = VarShift(fullScaleResistance, 30 - RsOnLine_qFmt);
_iq RsOnLine_Ohms = _IQmpy(RsOnLine_pu, pu_to_ohms_sf);
```


Copyright © 2013–2019, Texas Instruments Incorporated

15.10 Using the Rs Online Feature as a Temperature Sensor

This unique feature to monitor the resistance of the motor while spinning, allows the user to monitor the temperature of the coils based on the resistance increment. To show an example of a temperature sensor implementation, consider the values from Table 15-1.

Table 15-1. Temperature Sensor Implementation Values

Parameter	Value	Description
R	12.0 Ω	Resistance at temperature T. Value found with Rs Online.
R_0	10.0 Ω	Resistance at temperature T0. Value found with Rs Offline.
α	0.00393ºC^{-1}	Temperature coefficient of the material, in this case copper.
T_0	20ºC	Reference temperature of the material.
T	?	Final temperature of the material. To be calculated based on Rs Online.

Once the Rs Online feature is enabled, consider a resistance increase from 10.0 Ω to 12.0 Ω. The temperature of the motor windings can be calculated based on the following equation, derived from the equation listed in the previous section:

$$T = T_0 + \frac{\frac{R}{R_0} - 1}{\alpha}$$

$$T = 20°C + \frac{\frac{12.0\Omega}{10.0\Omega} - 1}{0.00393°C^{-1}}$$

$$T = 70.89°C \quad (88)$$

The following code example shows how to implement the temperature monitor:

```
#define COPPER_TEMP_COEF_INV_C (0.00393)
#define RS_AT_ROOM_TEMP_OHMS          (10.0)
#define ROOM_TEMP_C                   (20.0)

// Derived defines, pre-calculated by the compiler, not the CPU

#define INV_COPPER_TEMP_COEF_C          (1.0/COPPER_TEMP_COEF_INV_C)
#define INV_RS_AT_ROOM_TEMP_INV_OHMS    (1.0/RS_AT_ROOM_TEMP_OHMS)

CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;
float_t RsOnLine_Ohm = EST_getRsOnLine_Ohm(obj->estHandle);
float_t Temperature_C = \
                    (ROOM_TEMP_C) + \
                    (RsOnLine_Ohm * (INV_RS_AT_ROOM_TEMP_INV_OHMS) - 1.0) * \
                    (INV_COPPER_TEMP_COEF_C);
```

This code example can be executed in the background outside of the interrupts. The execution time is not critical at all, since temperature changes are much slower compared to the CPU timing.

So by using Rs Online, users can set a same temperature limit to their motor to avoid damage and malfunction of the system. For easier computation of the temperature, a look up table is recommended, to avoid the execution penalty of this equation in real time.

15.11 Rs Online Related State Diagrams (CTRL and EST)

There are several references to the controller (CTRL) and estimator (EST) state diagrams throughout the document. In this section, both state machines are shown for reference. For more details about the motor identification process, see Chapter 6.

Copyright © 2013–2019, Texas Instruments Incorporated

PowerWarp™

The PowerWarp algorithm of FAST adaptively reduces current consumption in order to minimize the combined (rotor and stator) copper losses to the lowest, without compromising the output power level of the AC Induction Motor.

Topic **Page**

16.1 Overview **548**
16.2 Enabling PowerWarp **549**
16.3 PowerWarp Current Slopes **550**
16.4 Practical Example **551**
16.5 Case Study **553**

Copyright © 2013–2019, Texas Instruments Incorporated

16.1 Overview

For applications that require minimum power consumption using AC Induction Motors (ACIM) PowerWarp presents a solution. By simply setting an enable flag in InstaSPIN-FOC, the FAST estimator will recalculate the magnetizing current, so that the minimum current is used to produce the torque needed for a given load and speed.

- PowerWarp algorithm is a capability of InstaSPIN-FOC designed to improve induction motor efficiency under partially loaded conditions
- Note that output power is maintained with PowerWarp algorithm enabled

PowerWarp has the greatest effect on motor efficiency at partial loads. However, as a result of de-fluxing the motor, the ability of the control system to respond to sudden transient conditions is diminished. However, flux angle tracking is not affected by this phenomenon

As shown in Figure 16-1, PowerWarp, when enabled, acts on the magnetizing current provided by the FAST estimator as highlighted. This magnetizing current is referred as I_{rated} in the diagram.

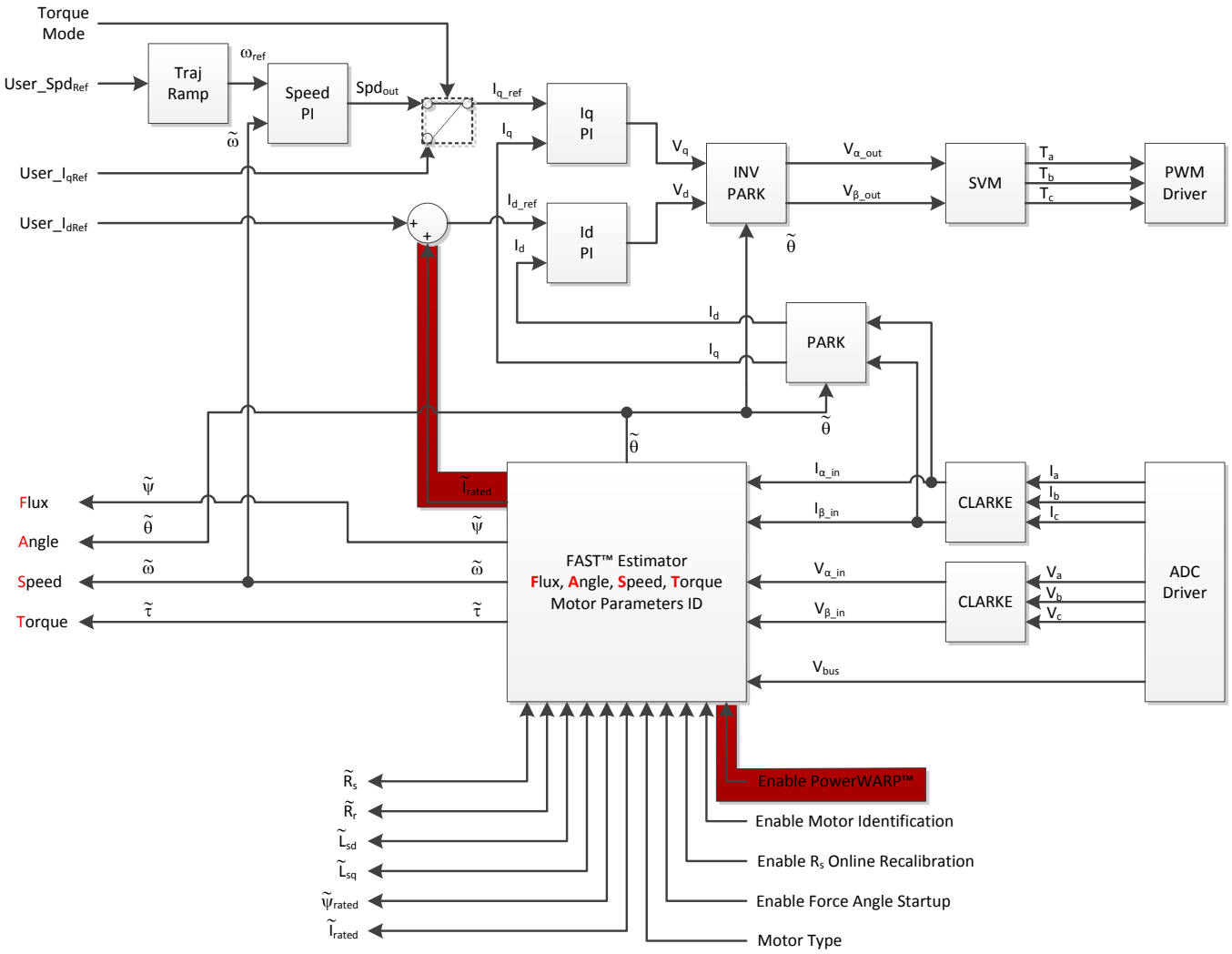


Figure 16-1. FAST Estimator with PowerWarp

Copyright © 2013–2019, Texas Instruments Incorporated

16.2 Enabling PowerWarp

In the code, all the functions related to PowerWarp are referred as POWERWARP. The following code example shows how this flag is enabled. As can be seen, POWERWARP is mentioned in the name of the function, which relates to PowerWarp.

```
CTRL_setFlag_enablePOWERWARP(ctrlHandle,TRUE);
```

The use of this flag does not take any effect unless the controller and estimator are running OnLine, in other words, not identifying the motor but running it in closed loop. Another condition is that the motor type has to be an AC Induction Motor. Nothing happens if the motor is a PM Motor. So, to summarize the conditions under PowerWarp enable will have an effect, these conditions must be met:

- The controller is running and motor is in closed loop.
- The estimator is running, and motor has been identified.
- Motor type must be induction motor.

In Figure 16-2, the highlighted state of the InstaSPIN controller is where the motor is running in closed loop. This state is also known as the Online State, and it is the state where PowerWarp algorithm can be executed (for complete details on the CTRL and EST states, see Chapter 6).

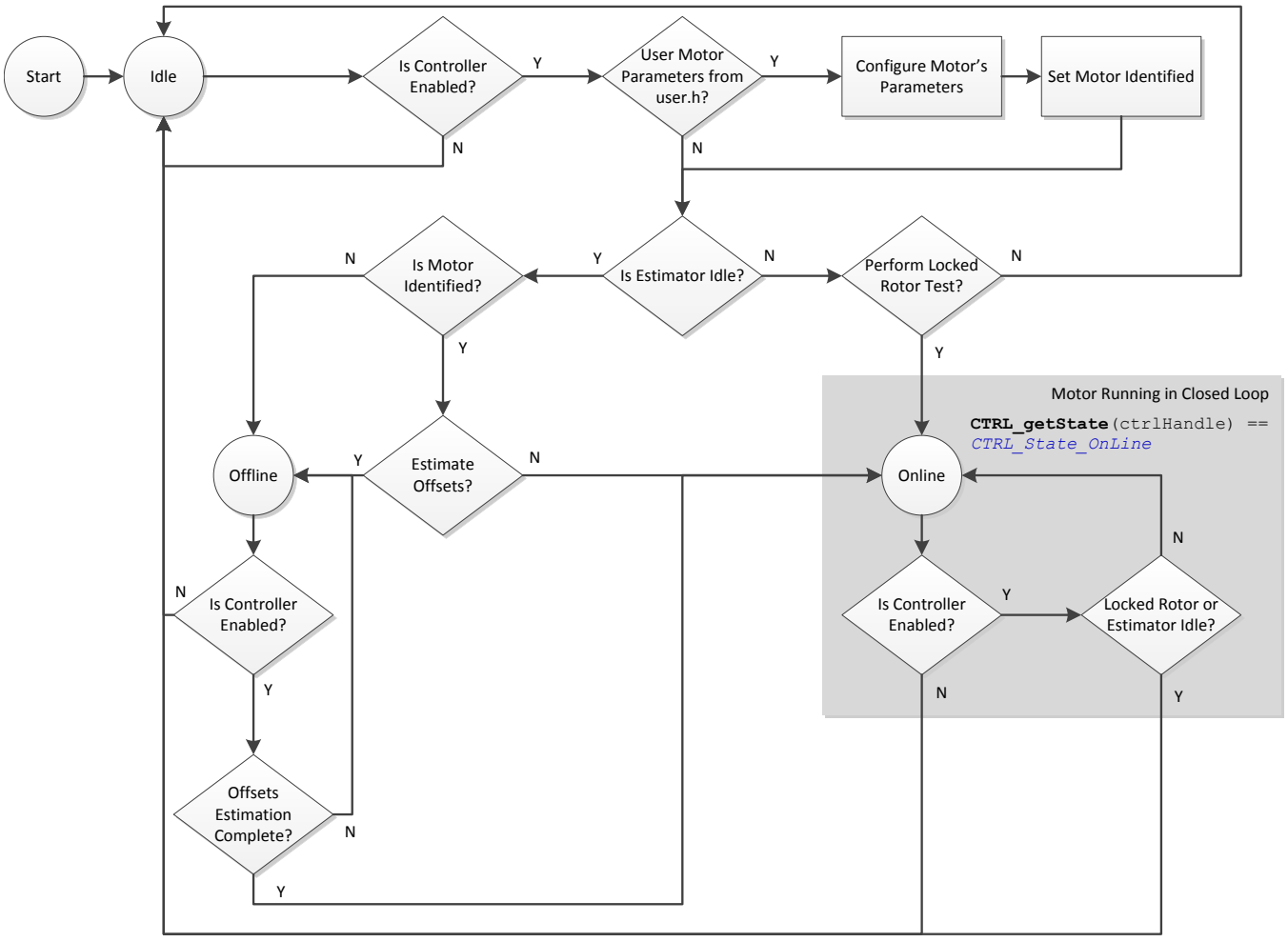


Figure 16-2. InstaSPIN Controller Flowchart - PowerWarp Executed in Closed Loop

Copyright © 2013–2019, Texas Instruments Incorporated

Similarly, the estimator's state machine where PowerWarp algorithm can be executed is highlighted from the following state machine. The state machine shown in Figure 16-3 represents the state where the motor is running in closed loop, from the estimator's perspective.

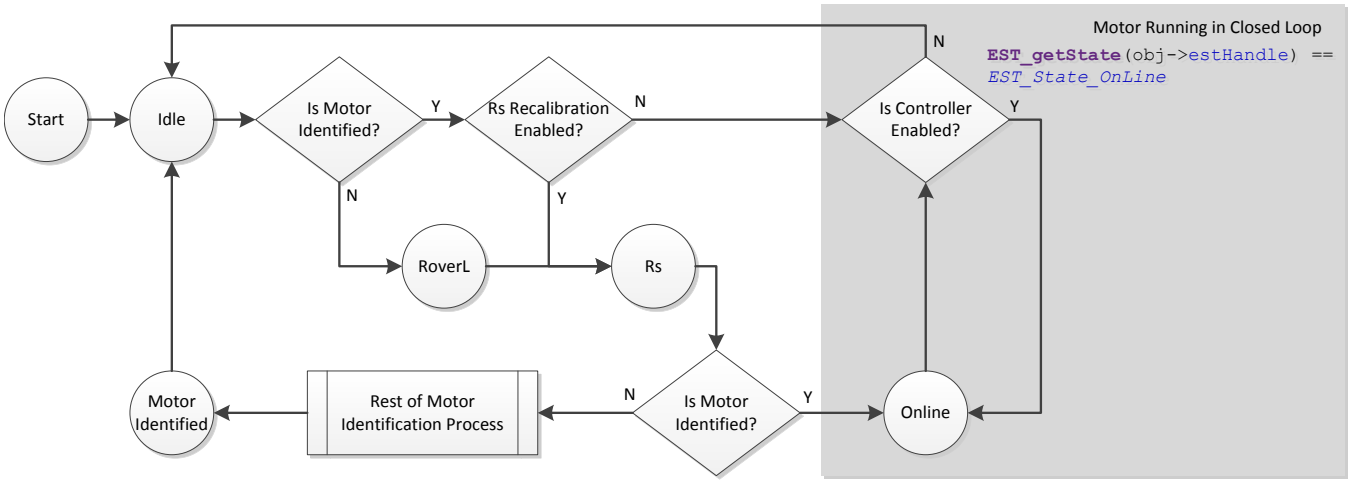


Figure 16-3. FAST Estimator State Machine Flowchart - PowerWarp Executed in Closed Loop

This code example shows how to check the state of state machines: Controller (CTRL) and Estimator (EST) estate machines, as well as the motor type:

```
CTRL_Obj *obj = (CTRL_Obj *)ctrlHandle;
ctrlState = CTRL_getState(ctrlHandle);
estState = EST_getState(obj->estHandle);
motorType = CTRL_getMotorType(ctrlHandle);

if( (ctrlState == CTRL_State_OnLine) &&
   (estState == EST_State_OnLine) &&
   (motorType == MOTOR_Type_Induction) )
  {
   CTRL_setFlag_enablePOWERWARP(ctrlHandle,TRUE);
  }
```

16.3 PowerWarp Current Slopes

In order to keep a smooth transition between rated magnetizing current and a reduced current provided by PowerWarp algorithm, a linear transition is generated when this mode is enabled and disabled. When PowerWarp is enabled, the rate at which the current is changed is set by the following defined in user.h:

```
#define USER_MAX_CURRENT_SLOPE_POWERWARP
(0.3 * USER_MOTOR_RES_EST_CURRENT
/USER_IQ_FULL_SCALE_CURRENT_A
/USER_TRAJ_FREQ_Hz)
```

The rate of change for the current after PowerWarp is enabled is equal to the current used for resistance estimation times 0.3 Hz. For example, if 1 Ampere is used for resistance estimation, the rate at which PowerWarp will change the rated current of the ACIM motor would be: 0.3 Amperes per second. So if PowerWarp reduced the rated current from 3 A to 1.5 A, it would take (3-1.5)/0.3 = 5 seconds to reach the new rated current.

Copyright © 2013–2019, Texas Instruments Incorporated

A second current slope is also defined, which is used whenever the I_{rated} is changed in the software, or when PowerWarp is disabled.

```
#define USER_MAX_CURRENT_SLOPE
(USER_MOTOR_RES_EST_CURRENT
/USER_IQ_FULL_SCALE_CURRENT_A
/USER_TRAJ_FREQ_Hz)
```

By default, this current slope is set to the current used for resistance measurement, per second. For example, if 1 A is used for resistance measurement, PowerWarp current is 1 A, and I_{rated} current is 3 A, then it will take (3 – 1)/1 = 2 seconds for the current to grow back to I_{rated}.

16.4 Practical Example

The following plot shows a practical example of PowerWarp, and the power savings related to this mode. The motor under test has the following parameters:

AC Induction Asynchronous Machine (GE 5K33GN2A)

- Rated power: ¼ Hp
- Rated torque: 9.22 Lb.in
- Rated voltage: 208~230 (v)
- Rated full load current: 1.3~1.4 (A)
- Rated full load speed: 1725 rpm
- Pole Pairs: 2
- Frequency: 60 Hz

Notice that the motor efficiency with PowerWarp is dramatically improved from 27% to 68% at 1 lb.in of load. Since PowerWarp reduces the ability of the ACIM motor to produce torque, at higher torque demands, energy savings with PowerWarp are also reduced. For the same reason, at rated torque, efficiency curves are identical when PowerWarp is enabled or disabled.

Even though PowerWarp reduces the ability to produce torque, the mechanical power delivered to the shaft is maintained with PowerWarp algorithm enabled, not affecting the mechanical system performance. In other words, from a purely mechanical standpoint, the output mechanical torque and speed are not changed when PowerWarp is enabled. What changes is the electrical power delivered to the motor to produce a certain mechanical output.

As shown in Figure 16-4, motor efficiency is boosted dramatically at lower loads, with a trade-off in dynamic torque and speed response, though the control system remains stable.

Copyright © 2013–2019, Texas Instruments Incorporated

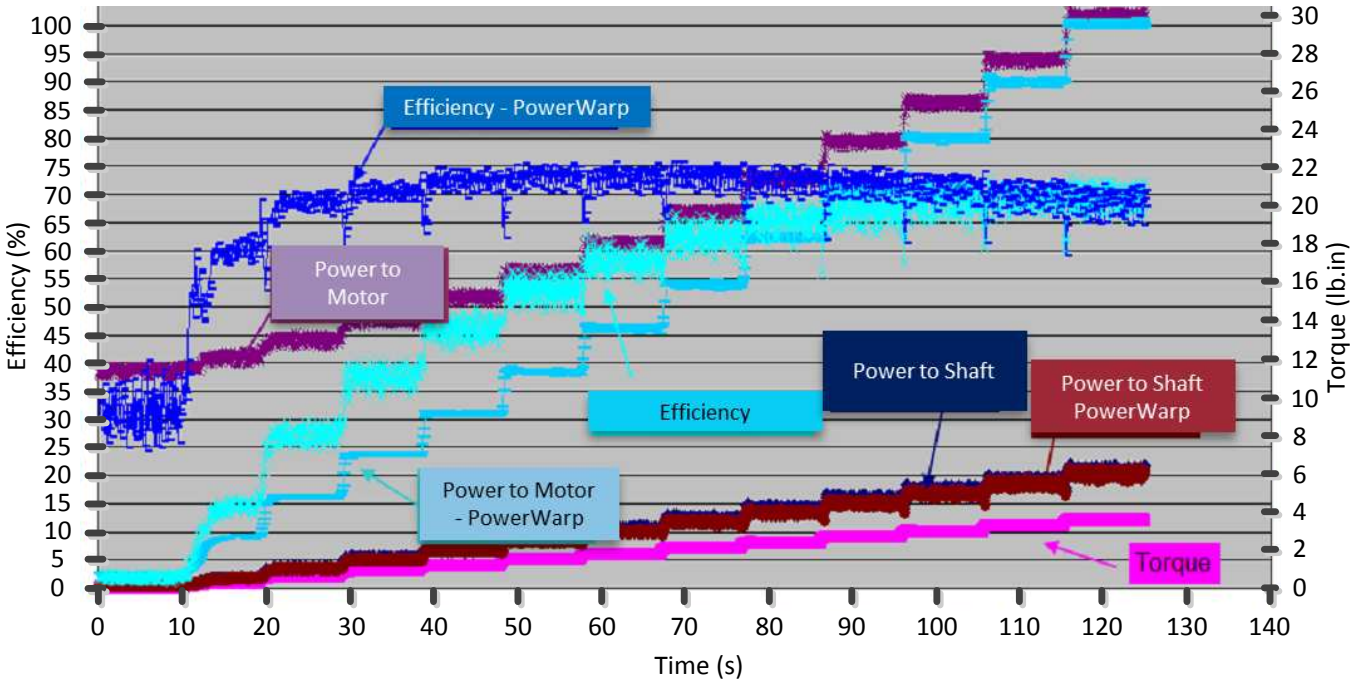


Figure 16-4. PowerWarp Improves Motor Efficiency

The default current slopes are used for this practical example.

```
#define USER_MAX_CURRENT_SLOPE_POWERWARP (0.3 * USER_MOTOR_RES_EST_CURRENT \
                                        / USER_IQ_FULL_SCALE_CURRENT_A \
                                        / USER_TRAJ_FREQ_Hz)
```

Figure 16-5 shows the time it takes for the FAST estimator to reduce the rated current when PowerWarp is enabled.

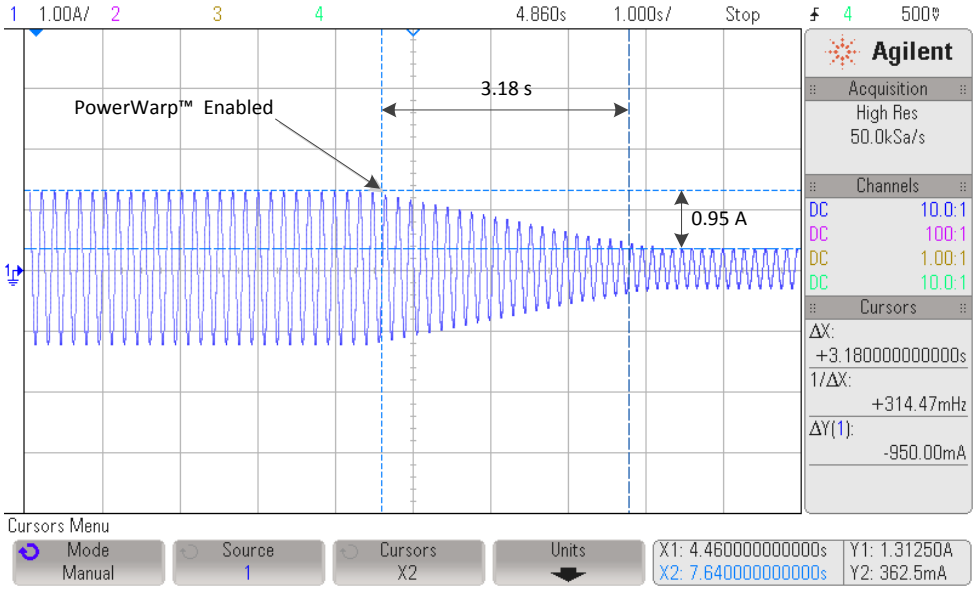


Figure 16-5. Current Reduced When PowerWarp Enabled

Copyright © 2013–2019, Texas Instruments Incorporated

It can be seen than the time it takes to reach the lowest current is:

$$(I_{rated} - I_{PowerWarp})/0.3 = (1.3125 - 0.3625)/0.3 = 3.167 \text{ seconds} \quad (89)$$

Also, the default has been used for current slopes when PowerWarp is disabled

```
#define USER_MAX_CURRENT_SLOPE
(USER_MOTOR_RES_EST_CURRENT
/USER_IQ_FULL_SCALE_CURRENT_A
/USER_TRAJ_FREQ_Hz)
```

Figure 16-6 shows this slope. The time it takes is: (1.3125 – 0.3625)/1 = 1.0 seconds.

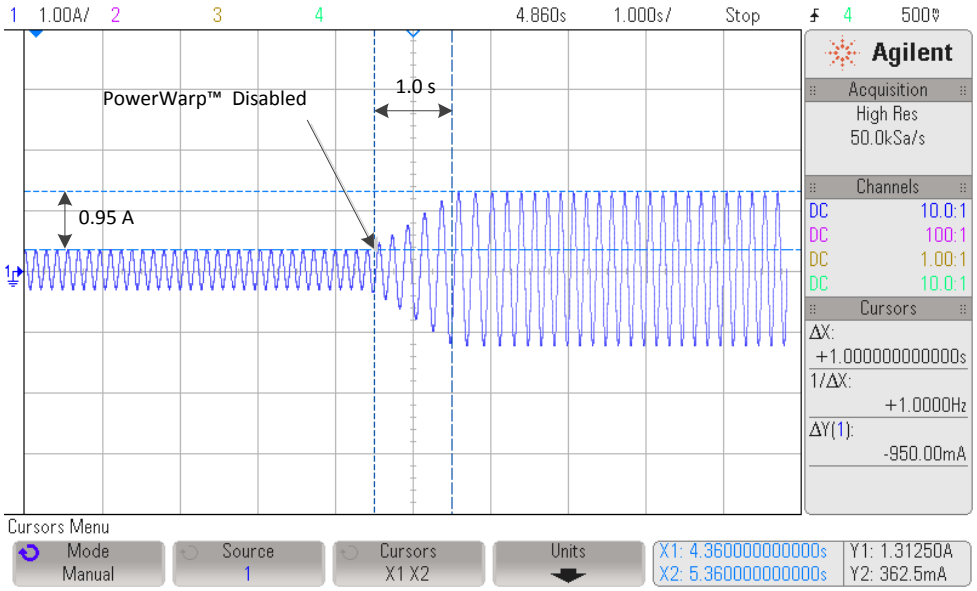


Figure 16-6. Current Slopes When PowerWarp Disabled

16.5 Case Study

Two pairs of motors were placed running the same load, which in this case were fans. One pair ran for 15 months side to side comparing energy consumption performance between InstaSPIN-FOC with PowerWarp algorithm enabled versus a TRIAC control of an Induction Motor. The energy savings over time are significant, average savings of about 81% of the energy when using InstaSPIN with PowerWarp enabled. In other words, with PowerWarp algorithm enabled, each motor only consumes about 19% of the power compared to a TRIAC controller. This percentage is calculated as follows:

Total Energy consumed by the TRIAC controller = 2096.6 kWh (90)

Figure 16-7 shows the energy consumption per month and an accumulative energy savings in kWh.

Copyright © 2013–2019, Texas Instruments Incorporated

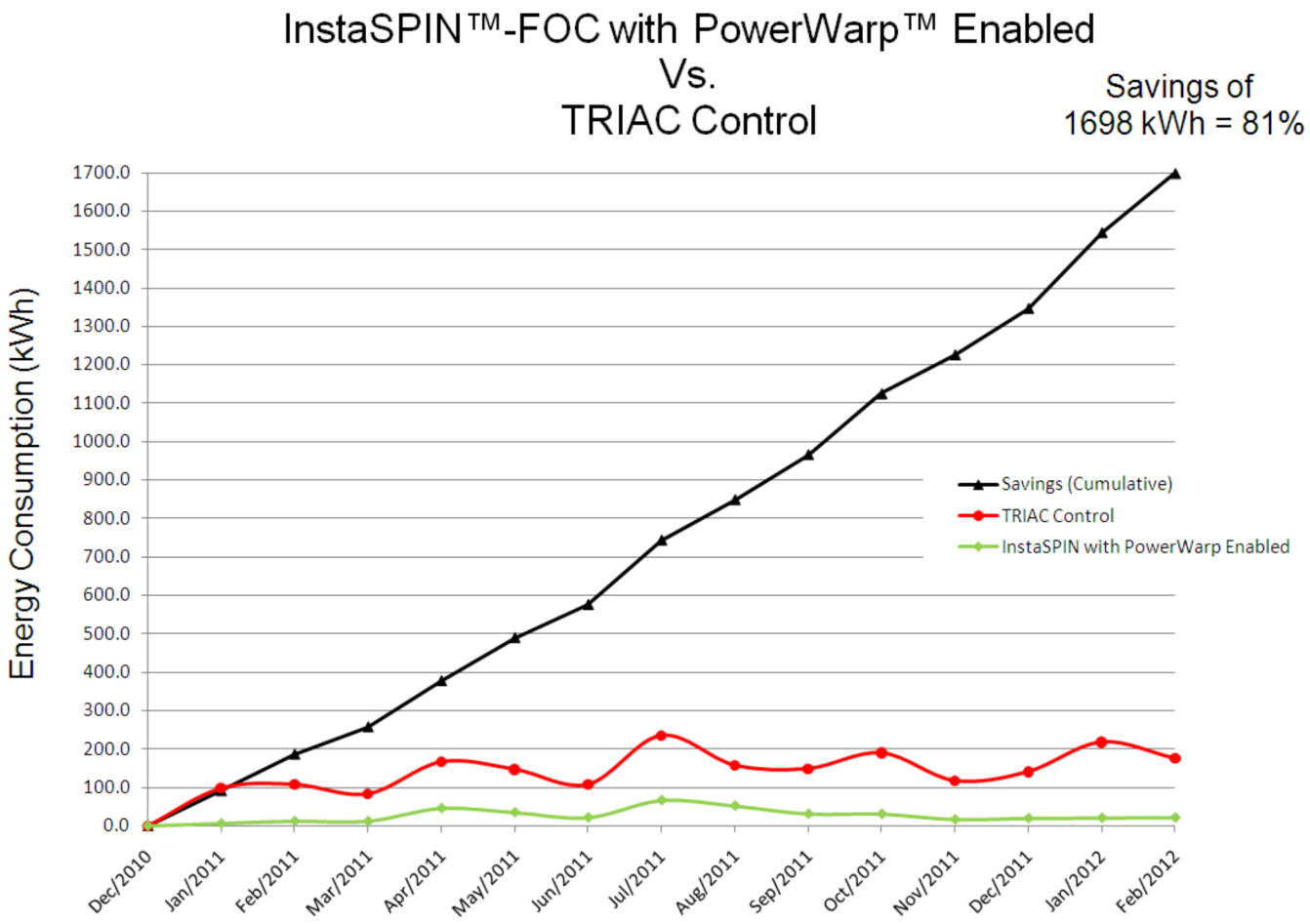


Figure 16-7. PowerWarp Algorithm Enabled vs. TRIAC Control of Induction Motor

The second pair of system was running InstaSPIN with PowerWarp enabled, but now versus a Volts-per-Hertz Control, also known as Frequency Control. The energy savings are also significant in this scenario, since PowerWarp optimizes the current consumption to minimize copper losses of the motor. In this case, the average energy savings were 48%, and it was calculated as follows:

Total Energy consumed by the Frequency controller = 916.5 kWh

Total Energy consumed by InstaSPIN with PowerWarp = 478.9 kWh

Average Energy Savings = 100% * (1 – 478.9/916.5) = 47.75%

Figure 16-8 shows these results.

Copyright © 2013–2019, Texas Instruments Incorporated

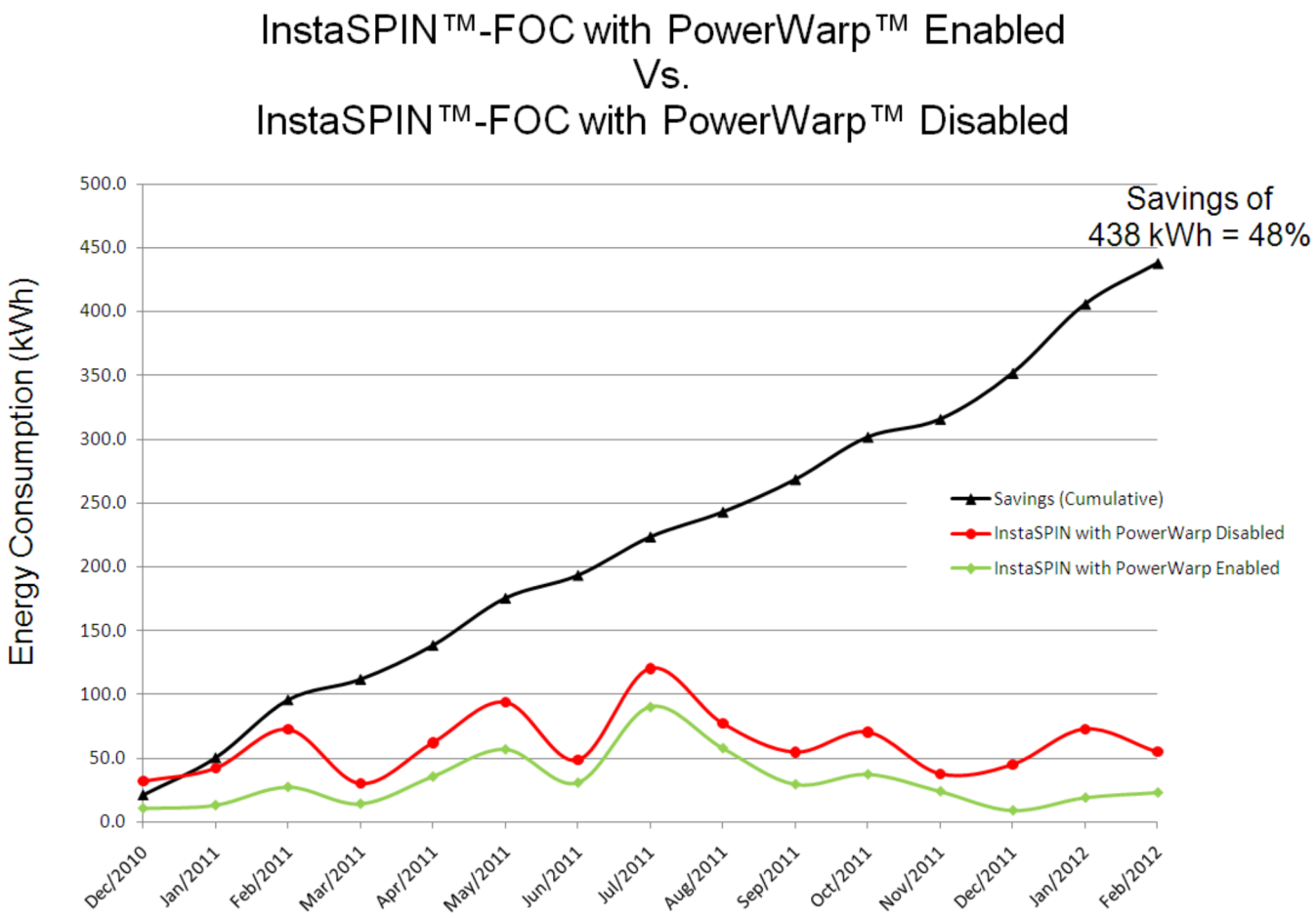


Figure 16-8. InstaSPIN-FOC with PowerWarp Enabled vs. InstaSPIN-FOC with PowerWarp Disabled

Copyright © 2013–2019, Texas Instruments Incorporated

Shunt Current Measurements

One-, two-, and three-shunt current measurement techniques are studied. We will show why the three shunt technique does not add significantly more cost to the current measurement circuit. And why it is much better to use the three shunt technique for motor control applications.

Topic **Page**

17.1 Introduction 557
17.2 Signals 557
17.3 1-Shunt 557
17.4 2-Shunt 561
17.5 3-Shunt 562
17.6 Development Kits 563
17.7 Conclusion 564

Copyright © 2013–2019, Texas Instruments Incorporated

17.1 Introduction

InstaSPIN can be used with many different types of current measurement techniques. The techniques include 3-shunt and 2-shunt resistor and LEM sensor measurements. For lower power drives, shunt resistors are the most widely used approach to measuring phase currents in a motor controller. Currently the InstaSPIN software suite does not provide a single shunt resistor solution. The reason for not providing single shunt current measurements and why a 3-shunt measurement is the ideal method for measuring currents will be provided below.

17.2 Signals

Before talking about the different types of resistor shunt current measurements, we will study the switching signals involved and where the measurement has to be taken to measure current. Most field oriented control (FOC) drives use the space vector modulation (SVM) technique to send the duty cycle commands to the inverter switches to power the motor. A typical SVM voltage waveform looks like the blue signal as shown in Figure 17-1. The SVM waveform is next sampled by a triangular waveform which is the red signal in Figure 17-1. Whenever the triangle is greater than the SVM, the lower switch of an h-bridge phase is turned on. The resulting PWM waveform that is used to control an h-bridge phase is also shown below.

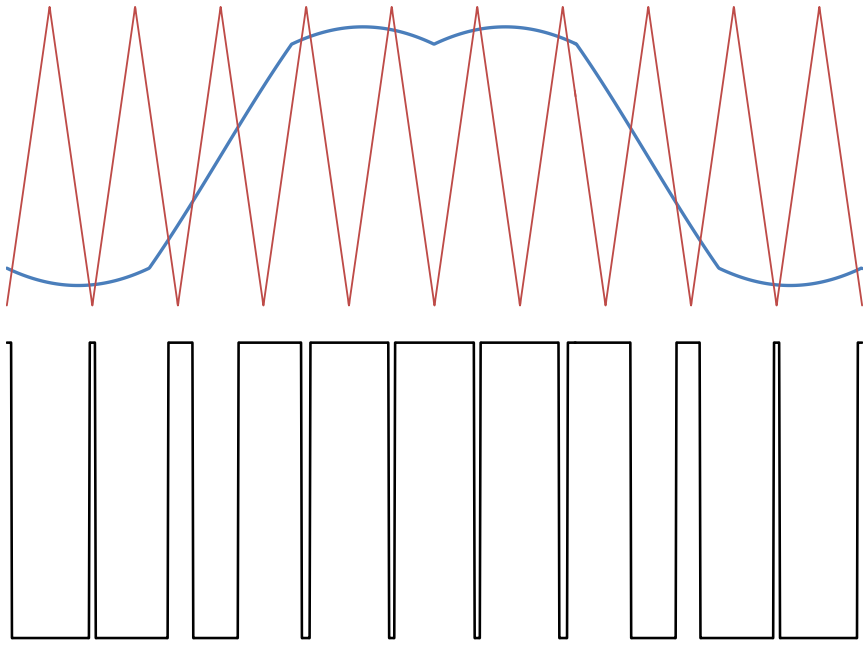


Figure 17-1. Typical SVM Waveform Sampled by Counter

To measure current that is flowing through a phase, the normal approach is to have a resistor located at the base of the phase. So no matter what resistor configuration there is, 1-shunt, 2-shunt or 3-shunt, current can only be measured when a lower switch is on. In the PWM waveform the lower switch is on when the square wave signal is low. To sample the current, it has to be clean. A clean current signal or representation of the current signal must have no ringing or noise. With that being said, let us start looking at the different resistor shunt current measurement techniques that are used in the field.

17.3 1-Shunt

The single shunt current measurement technique measures the power supply current and with knowledge of the switching states recreates each of the three phase currents of the motor. Figure 17-2 illustrates where the single shunt is located in the inverter circuit.

Copyright © 2013–2019, Texas Instruments Incorporated

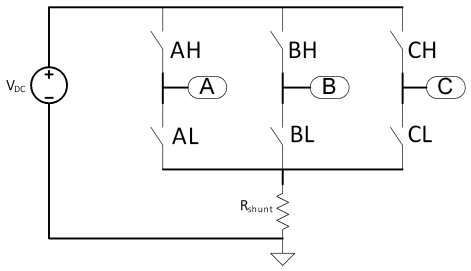


Figure 17-2. Single-Shunt Current Measurement Circuit With Inverter

There are eight different switch options in SVM. Table 17-1 explains each one and shows the direction of the voltage space vector and what current can be measured in that state. With the switches in states 0 and 7 only circulating current is present and there is no possibility to measure current with the single shunt technique. To properly measure current with the single shunt technique, the current measurement and switching state have to both be considered.

Table 17-1. The Eight SVM Switching States

Switch State	AH	BH	CH	Vector	Measure
0	0	0	0	•	offsets
1	1	0	0	→	Ia
2	1	1	0	↗	-Ic
3	0	1	0	↖	Ib
4	0	1	1	←	-Ia
5	0	0	1	↙	Ic
6	1	0	1	↘	-Ib
7	1	1	1	•	offsets

Figure 17-3 shows a SVMPWM waveform and the current measurement signal that result. In this case the current conduction times for I_C and I_A are on long enough so that the slew rate of the op-amp and settling time of the whole measurement system have enough time to go to steady state so that the ADC can have enough time to sample the current. As we will see shortly, when using the single shunt technique, it is mandatory to be able to measure current in the smallest time possible.

Copyright © 2013–2019, Texas Instruments Incorporated

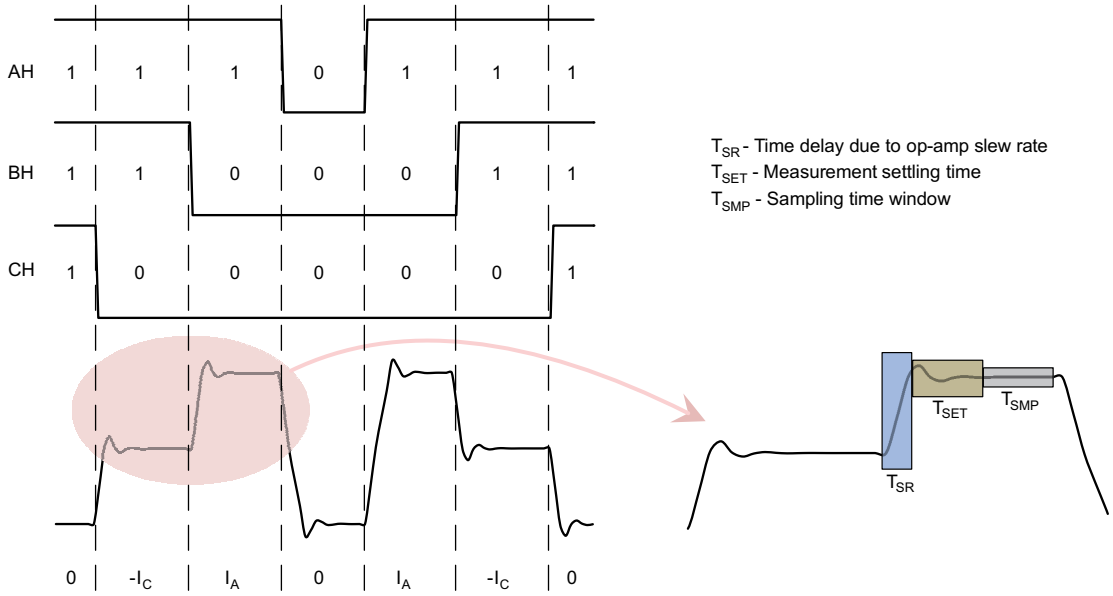


Figure 17-3. Single-Shunt Current Measurement When Sampling Times are Long Enough

In Figure 17-4, imagine the voltage space vector traversing counter clockwise around the circle. As the space vector points toward the corners of the hexagon, the time window for sampling current completely disappears. There are zones located at 0, 60, 120, 180, 240, and 300 degrees where only one current can be measured and the other two currents must be found in another fashion.

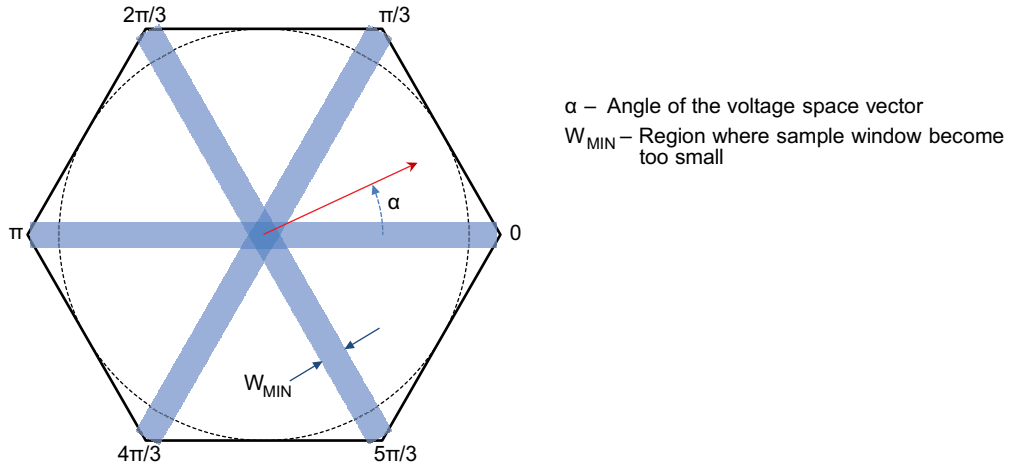


Figure 17-4. SVM and Regions Where Current Measurement is Not Allowed

In Figure 17-5, the space vector is pointing too close to π/3 and is causing the current measurement window to shrink for I_A. Because of slew rate of the op-amp and a long settling time, I_A will be missed and will cause an error in the FOC controller. One way of fixing this problem is to force a measurement window opening that lasts long enough to accommodate slew rate and settling time. An illustration of this technique is shown in Figure 17-6. The maximum duty cycle waveform is shifted to the right and the minimum duty cycle waveform is shifted to the left. The advantage of phase shifting the PWMs like this is that there is no distortion in the voltage waveform result per phase. Software still has to be written to compensate for the resulting current waveform and even though a current measurement window can be made as large as needed, it is best to keep the window as small as possible. As the space vector reaches the voltage limit set by the DC bus, there will be less room to shift the signals. So to get the best utilization of the DC bus and still have the ability to measure current requires that the chosen op-amp have very high slew rate and low settling time.

Copyright © 2013–2019, Texas Instruments Incorporated

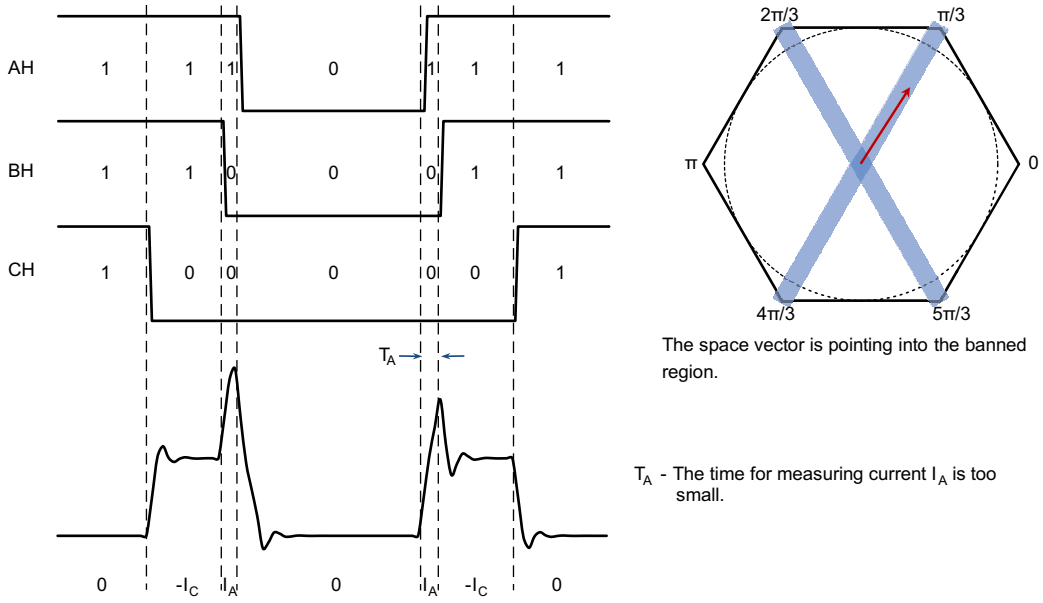


Figure 17-5. Example of When Current Sampling Window Disappears

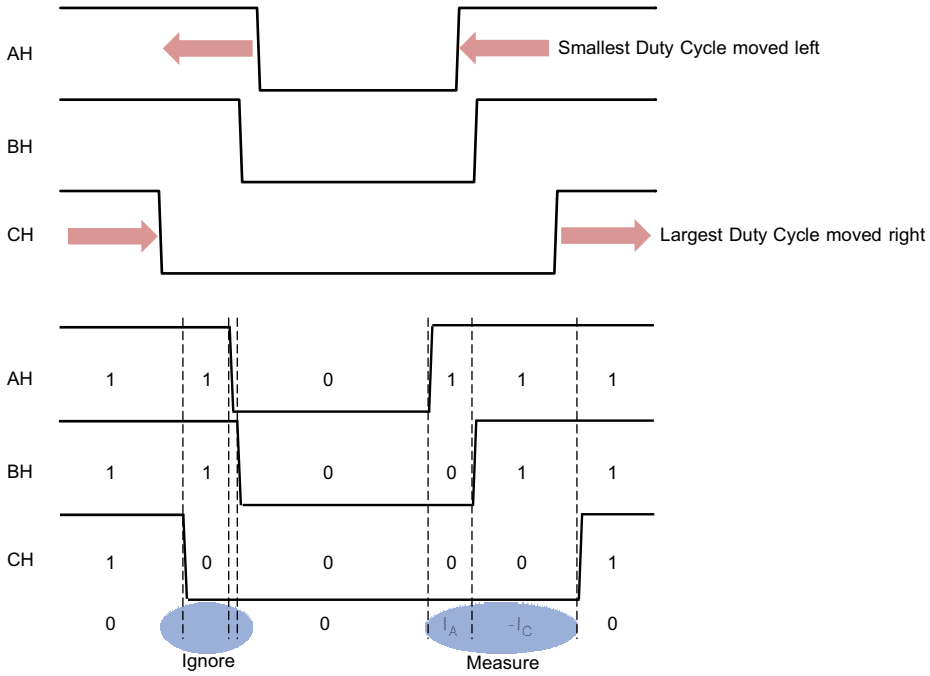


Figure 17-6. Phase Shifting the PWMs to Allow for a Large Enough Current Measurement Window

Current ripple is another problem that arises when using the single shunt technique. The motor is an inductive and resistive circuit element and therefore has an R/L time constant. On the current waveform that is shown in Figure 17-3, the motor's electrical time constant is large thus causing the current to be very level and the measured currents I_C and I_A can be considered the average current going to the motor. If the motor's R/L time constant is smaller, then the current will look more like a saw tooth wave. Now the current has to be sampled as close to the center of the total conduction time as possible to obtain the average motor current. This will cause an even stricter performance requirement for the chosen op-amp.

Copyright © 2013–2019, Texas Instruments Incorporated

Let's run through a quick calculation to see what types of op-amp parameters are needed. First, a normal PWM frequency is around 20 kHz which is a period of 50 µs. At 20 kHz when causing a deadtime or any non-symmetric adjustment of the PWM that is 0.5 µs or greater, current distortion will occur. The C2000 F2805xF and F2806xF family of processors has the capability of 90 MHz clock speeds which translates to a 45 MHz ADC clock. The minimum sampling window is 7 ADC clock cycles or 156 ηs. The worst case time delay when considering slew rate delay is during the maximum voltage transition in this case 3.3 V. Ignoring settling time, the slew rate that will keep the signal measurement below 0.5 µs is 3.3/344 ηs or 9.6 V/µs. Settling time will take up about half of the time, so to be safe the op-amp slew rate should be chosen at 20 V/µs.

17.4 2-Shunt

The two shunt current measurement technique uses the principle of Kirchhoff's Current Law (KCL) that the sum of the currents into a single node equals zero. By measuring only two phase currents, the third is calculated with KCL. A circuit for the two shunt current measurement technique is shown in Figure 17-7.

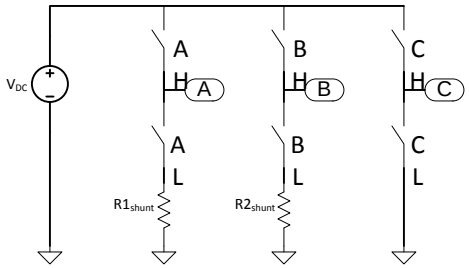


Figure 17-7. Two-Shunt Measurement Circuit With Inverter

The two shunt measurement circuit has an advantage over the single shunt circuit in that it can see circulating currents. Now all currents are measured only during switching state zero. Figure 17-8 shows an example of a switching waveform and where the ADC samples the current. The PWM for I_A is almost 100% duty cycle and in this example causes the I_A current to rise. The PWM for I_B is about 50% duty cycle and its current stays at about zero amps for this period. Phase current can only be measured when that phase's lower switch is conducting. In the example, I_A is measureable for a very short time while I_B has a long time to view. When the measured phase is operating at PWMs near 100%, this is the inherent problem when using the two shunt technique. For the example when I_A is sampled, the measured current signal has not yet stabilized giving an incorrect representation of the current signal. Another pointer to note about using the two shunt technique during motoring is that the current being measured is now bipolar. So zero amps is now represented as half of the ADC full scale.

Copyright © 2013–2019, Texas Instruments Incorporated

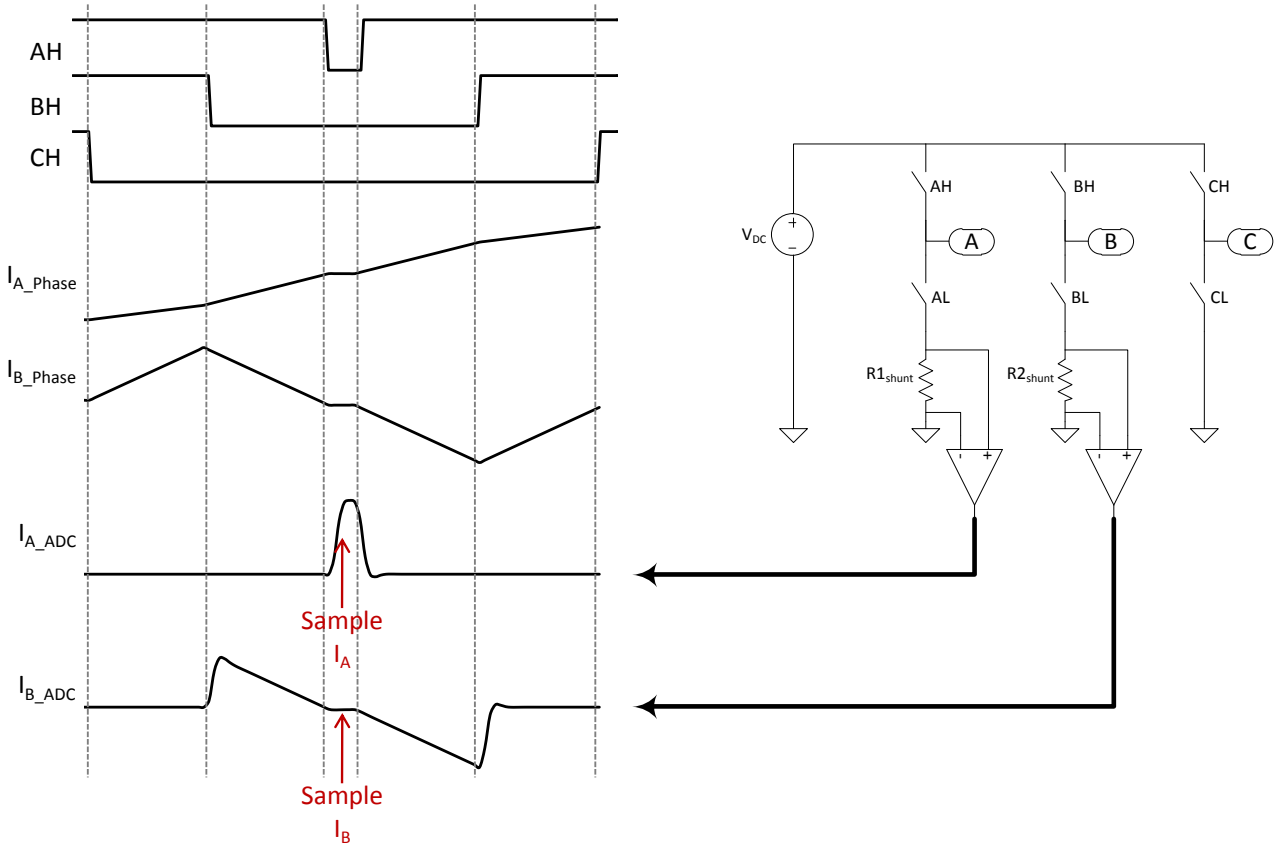


Figure 17-8. Sampling Current When Using Two-Shunt Measurement Technique

As the duty cycle increases the time to measure voltage across the shunt resistor for the phase needs to be quicker. For example if a duty cycle of 95% were commanded with a 20 kHz PWM waveform there will be 2.5 µs of on-time for the measured phase. Ideally the slew rate of the op-amp should be 1/10 of the on-time or 0.25 µs. A full scale output voltage transition of the op-amp is 1.65 V. The minimum slew rate is calculated to be 6.6 V/µs. As the duty cycle increases even more, the slew rate must be increased to capture the signal properly. Although the two shunt current measurement technique lessens the op-amp's speed requirement as compared to the single shunt measurement, there is a duty cycle where the slew rate has to be very large. For the single and two shunt measurement techniques there is no way of getting around the need for a fast and expensive op-amp.

17.5 3-Shunt

An example of the three shunt current measurement circuit is shown in Figure 17-9. The three shunt current measurement technique is very robust and surprisingly can be cost effective even when compared to using a single or two shunt measurement technique. First, with the single and two shunt circuits over-modulation is difficult to achieve. Second, higher priced fast slew rate op-amps must be selected for the one and two shunt techniques. The three shunt technique can bounce sampling between current signals, selecting two out of three phases each period, which allows for long times for the current signals to settle. If large current measurement windows are possible, then much slower and cheaper op-amps can be used. For example, Figure 17-10 shows three PWM switching signals and what shunt resistor will be sampled. As can be seen, there is plenty of time for the current signal to stabilize.

Copyright © 2013–2019, Texas Instruments Incorporated

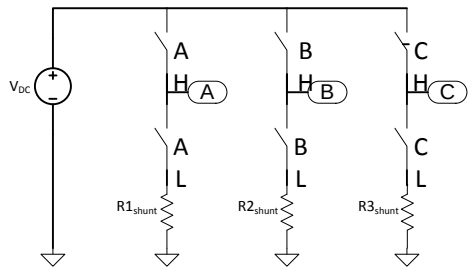


Figure 17-9. Three-Shunt Measurement Circuit With Inverter

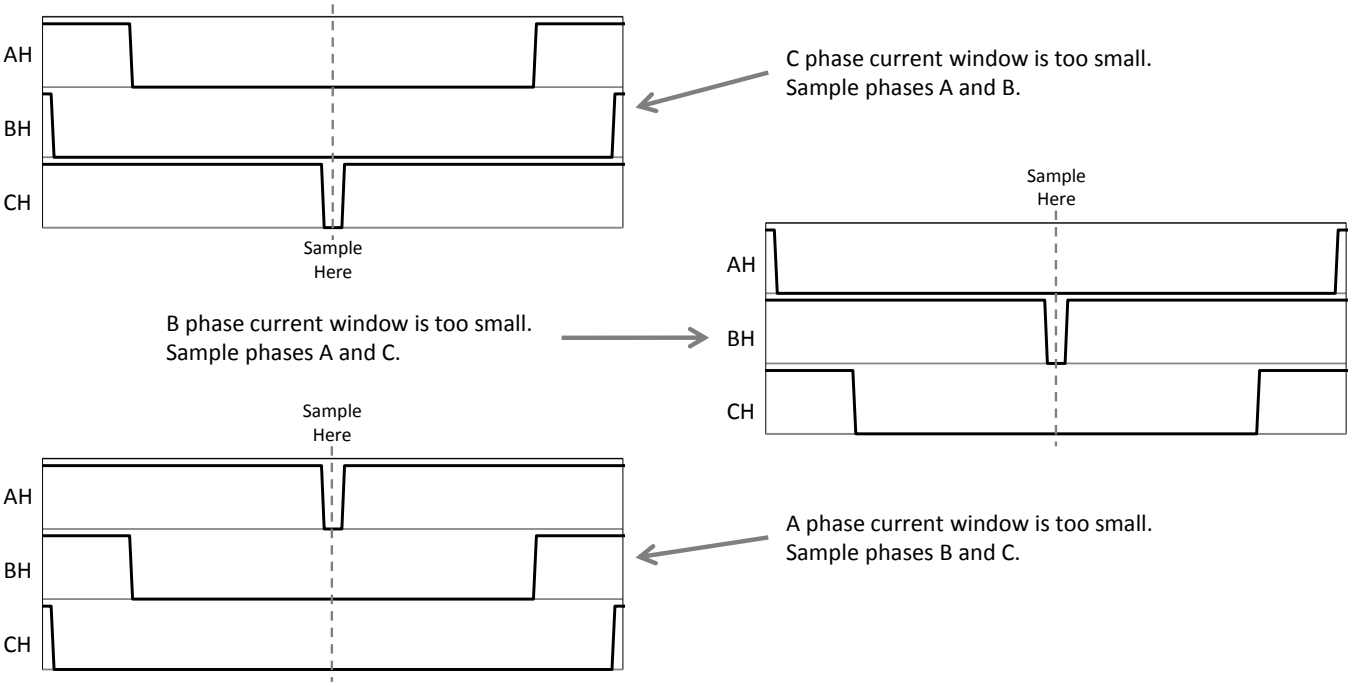


Figure 17-10. Using Three-Shunt Current Sampling Technique

With three shunts the op-amp slew rate can be well under 1 V/µs. There are many advantages to having slower amplifier circuits for current measurement. First cost will be less. Second a slower amplifier will have a lower bandwidth to pick up noise. Third in many of the current amplifier circuits there is crosstalk between the phase measurements. What will happen is a spike from the C phase switching will show up in the A phase measurement. A slow amplifier circuit will attenuate the crosstalk signal.

17.6 Development Kits

TI provides the hardware in three kits that allow for single, two and three shunt current measurement. Table 17-2 lists the current and voltage rating for each kit.

Table 17-2. Current and Voltage Ratings for TI Development Kits

Kit	Current Rating (A)	Voltage Rating (V)
DRV8312	6.5	50
DRV8301	82.5	50
HVKIT	10	350

Copyright © 2013–2019, Texas Instruments Incorporated

Below we will show pertinent schematics for the kits and how they can be used for the three different current measurement techniques.

17.6.1 DRV8312

The DRV8312 kit is the lowest power kit of all TI kits. It uses a TI DRV8312 power module which contains six power MOSFETs and their corresponding gate drivers. The kit accepts any C2000, 100 pin DIMM cards (Control Cards). A unique feature of the DRV8312 is that it can sustain switching frequencies up to 500 kHz with very high efficiency.

For the following discussion, refer to the schematic in the *Stellaris® DK-LM3S-DRV8312 Development Kit User's Manual* (SPMU276). OA3, OA4, and OA2 are op-amps that make up the Kelvin current connections for phases A, B, and C respectively. The differential gain of this circuit is 19.08 V/V. R52, R53, and R50 are shunt resistors for each phase and are 10 mM each. The amount of current that causes a 3.3 V output to the ADC is 3.3/.010/19.08 = 17.30 A/V. This current gain is used in the user.h file of InstaSPIN for the parameter USER_ADC_FULL_SCALE_CURRENT_A.

To select between the 2 current shunt method or the 3 current shunt method, set the user.h parameter USER_NUM_CURRENT_SENSORS to either 2 or 3 depending on the number of current shunts that are used.

17.6.2 DRV8301

The DRV8301 kit is a low voltage high current kit. It uses discrete MOSFETs that are switched by the DRV8301 gate driver.

17.7 Conclusion

Resistor shunt current measurement is a very reasonable technique for measuring current in a motor control inverter. There are three widely used examples, the 1-Shunt, 2-Shunt, and 3-Shunt resistor measurements. While at first the 1-Shunt and 2-Shunt techniques seem to be saving money, they require much faster and more expensive amplifier circuits (see Table 17-3). 1 and 2 Shunt current measurements also limit the capability of the current feedback which will limit the ability of the drive to use the full voltage that is provided to the inverter. The 3-Shunt technique is superior and not much different in cost due to the advantage of using cheap, slow, current amplifier circuits.

Table 17-3. Recommended Op-Amp Slew Rates for Corresponding Number of Sense Resistors

No. of Shunts	Op-Amp Slew Rate
1	>20 V/µs
2	>6 V/µs
3	>1 V/µs

Copyright © 2013–2019, Texas Instruments Incorporated

Sensored Systems

Sensored systems can also benefit from InstaSPIN-MOTION. Position control solutions rely upon an accurate electrical angle in order to control the angle of the motor. While sensorless estimators work well for velocity control applications, sensorless estimators do not provide an accurate enough motor angle for position control. Examples are provided to demonstrate how to use a quadrature encoder to provide an electrical angle feedback to InstaSPIN-MOTION for position control.

The FAST Software Encoder can still be used in position control applications to provide a backup encoder to detect that the primary electrical angle source is having a failure.

Topic **Page**

18.1 Hardware Configuration for Quadrature Encoder..... 566
18.2 Software Configuration for Quadrature Encoder..... 566
18.3 InstaSPIN-MOTION Position Convert..... 568

Copyright © 2013–2019, Texas Instruments Incorporated

18.1 Hardware Configuration for Quadrature Encoder

Quadrature encoders need to use the Enhanced Quadrature Encoder Pulse (QEP) peripheral found on the F2806xM and F2805xM devices. Your board design needs to feed the quadrature encoder signals into the microcontroller. If you are using a TI evaluation kit, the boards are setup to work correctly with a quadrature encoder. To attach the quadrature encoder, see the hardware manual for your specific evaluation kit.

18.1.1 Pin Usage

The QEP peripheral accepts A, B, and I inputs from the quadrature encoder. Table 18-1 lists the required pins for 1 QEP peripheral. These should be wired to the appropriate pins on your encoder.

Table 18-1. Pins Required to Connect Quadrature Encoder to eQEP Module

Pin Type	Pin Name	Pin Usage Per Motor
Digital	EQEPxA	3
	EQEPxB	
	EQEPxI	

18.2 Software Configuration for Quadrature Encoder

This software configuration will focus on using the MotorWare infrastructure to get your quadrature encoder working correctly. These steps need to be done for projects that use a quadrature encoder. Lab 12b — Using InstaSPIN-MOTION with Sensored Systems — is an example project that implements the steps required to use a quadrature encoder for feedback.

18.2.1 Configure Motor for EQEP Operation

An additional parameter needs to be defined in the user.h file. This parameter is USER_MOTOR_ENCODER_LINES. This value should be set to the number of lines (or pulses) that are on the motor's encoder. In the Lab 12b example project, this macro definition is included in user.h.

```
#elif (USER_MOTOR == Teknic_M2310PLN04K)
#define USER_MOTOR_TYPE                 MOTOR_Type_Pm
#define USER_MOTOR_NUM_POLE_PAIRS       (4)
#define USER_MOTOR_Rr                   (NULL)
#define USER_MOTOR_Rs                   (0.4076258)
#define USER_MOTOR_Ls_d                 (0.0001972132)
#define USER_MOTOR_Ls_q                 (0.0001972132)
#define USER_MOTOR_RATED_FLUX           (0.03975862)
#define USER_MOTOR_MAGNETIZING_CURRENT  (NULL)
#define USER_MOTOR_RES_EST_CURRENT      (1.0)
#define USER_MOTOR_IND_EST_CURRENT      (-0.5)
#define USER_MOTOR_MAX_CURRENT          (7.0)
#define USER_MOTOR_FLUX_EST_FREQ_Hz     (20.0)
// Number of lines on the motor's quadrature encoder
#define USER_MOTOR_ENCODER_LINES        (1000.0)
```

18.2.2 Initialize EQEP Handle

The HAL_init function, located in hal.c, initializes the handle for the QEP driver. This will provide the QEP handle with the location of the registers that it will modify. In the Lab 12b example project, this step is accomplished in the hal.c file.

```
// initialize EQEP handle
```


Copyright © 2013–2019, Texas Instruments Incorporated

```
obj->qepHandle[0] = QEP_init((void*)QEP1_BASE_ADDR,sizeof(QEP_Obj));
```

18.2.3 Set Digital IO to connect to QEP Peripheral

By default, the pins used by QEP driver are set to be general purpose digital IO. These need to be set in the HAL_setupGpios function, located in hal.c, to connect to the QEP driver. In the Lab 12b example project, this step is accomplished in the hal.c file.

```
// EQEPA
GPIO_setMode(obj->gpioHandle,GPIO_Number_20,GPIO_20_Mode_EQEP1A);

// EQEPB
GPIO_setMode(obj->gpioHandle,GPIO_Number_21,GPIO_21_Mode_EQEP1B);

//    EQEP1I
GPIO_setMode(obj->gpioHandle,GPIO_Number_23,GPIO_23_Mode_EQEP1I);
```

18.2.4 Enable Clock to eQEP

In the function HAL_setupPeripheralClks, located in hal.c, the clock needs to be enabled for the QEP driver you will be using. This will allow the QEP driver to work correctly. In the Lab 12b example project, this step is accomplished in the hal.c file.

```
CLK_enableEqep1Clock(obj->clkHandle);    // Enable clock to eQEP Module
```

18.2.5 Initialize ENC Module

The ENC module is used to convert the raw counts produced by the QEP driver into electrical angle that will be used by the FOC system. An ENC module object and handle need to be declared in the main source file of the project. In the Lab 12b example this is done in proj_lab12b.c.

```
ENC_Handle encHandle;
ENC_Obj enc;
```

Once the ENC module object and handle have been declared they need to be initialized. This will assign the handle to point to the specific memory used by the ENC module. This should be done prior to the main loop in the main source file.

```
// initialize the ENC module
encHandle = ENC_init(&enc, sizeof(enc));
```

18.2.6 Set Up ENC Module

Prior to the main loop in the main source file the ENC module needs to be setup. This will pass important values to the ENC module to allow it to interpret the raw quadrature counts from the QEP driver into electrical angle useable by the FOC.

```
// setup the ENC module
ENC_setup(encHandle, hal_obj->qepHandle[0], 1, USER_MOTOR_NUM_POLE_PAIRS,
USER_MOTOR_ENCODER_LINES, 0, USER_IQ_FULL_SCALE_FREQ_Hz, USER_ISR_FREQ_Hz, 8000.0);
```


Copyright © 2013–2019, Texas Instruments Incorporated

18.2.7 Call eQEP Function

In the main ISR we need to call the function to calculate the rotor angle from the EQEP driver. This function is called ENC_calcElecAngle. This needs to happen during every instance of the ISR.

```
// compute the electrical angle
ENC_calcElecAngle(encHandle);
```

18.2.8 Provide eQEP Angle to FOC

In the ctrl.h file the function CTRL_runOnline_User, we need to modify the source for the angle provided to FOC. As a default, FOC is set up to get the angle from the FAST estimator, but for sensored control, the FOC will receive the angle from the ENC module. In the Lab 12b example project, this step is accomplished in the ctrlQEP.h file.

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

// Update electrical angle from ENC module
angle_pu = ENC_getElecAngle(encHandle);
```

18.3 InstaSPIN-MOTION Position Convert

SpinTAC™ Position Convert is used to convert the electrical angle output from the ENC module into mechanical angle and speed feedback that is used in the rest of the system. The ENC module produces the rotor electrical angle used in the FOC. SpinTAC Position Convert will also estimate the slip velocity in AC induction motors. This is required to use AC induction motors with a physical sensor.

18.3.1 Software Configuration for SpinTAC Position Convert

Configuring SpinTAC Position Convert requires four steps. Lab 12b — Using InstaSPIN-MOTION with Sensored Systems — is an example project that implements the steps required to use the SpinTAC Position Convert. The header file spintac_velocity.h, included in MotorWare, allows you to quickly include the SpinTAC components in your project.

18.3.1.1 Include the Header File

This should be done with the rest of the module header file includes. In the Lab 12b example project, this file is included in the spintac_velocity.h header file. For your project, this step can be completed by including spintac_velocity.h.

```
#include "sw/modules/spintac/src/32b/spintac_pos_conv.h"
```

18.3.1.2 Declare the Global Structure

This should be done with the global variable declarations in the main source file. In the Lab 12b project, this structure is included in the ST_Obj structure that is declared as part of the spintac_velocity.h header file.

```
ST_Obj   st_obj;    // The SpinTAC Object
ST_HandlestHandle;  // The SpinTAC Handle
```


Copyright © 2013–2019, Texas Instruments Incorporated

This example is if you do not wish to use the ST_Obj structure that is declared in the spintac_velocity.h header file.

```
ST_PosConv_t      stPosConv;        // The SpinTAC Position Converter Object
ST_POSCONV_Handle stPosConvHandle;  // The SpinTAC Position Converter Handle
```

18.3.1.3 Initialize the Configuration Variables

This should be done in the main function of the project ahead of the forever loop. This will load all of the default values into SpinTAC Position Convert. This step can be completed by running the functions ST_init and ST_setupPosConv that are declared in the spintac_velocity.h header file. If you do not wish to use these two functions, the code example below can be used to configure the SpinTAC Position Converter. This configuration of SpinTAC Position Converter represents the typical configuration that should work for most motors.

```
// init the ST PosConv object
stPosConvHandle = STPOSCONV_init(&stPosConv, sizeof(stPosConv));

// Setup the SpinTAC Position Converter
// Sample time [s], (0, 1]
STPOSCONV_setSampleTime_sec(stPosConvHandle, _IQ(ST_SPEED_SAMPLE_TIME));
// The upper [0, 16] and lower [-16, 0] bounds of the input position signal [ERev]
STPOSCONV_setERevMaximums_erev(stPosConvHandle, _IQ(1.0), 0);
// Sets the unit conversions used in the SpinTAC Position Converter
STPOSCONV_setUnitConversion(stPosConvHandle, USER_IQ_FULL_SCALE_FREQ_Hz,
                                        ST_SAMPLE_TIME, USER_MOTOR_NUM_POLE_PAIRS);
// The Rollover bound of the output position signal [MRev]
STPOSCONV_setMRevMaximum_mrev(stPosConvHandle, _IQ(10.0));
// Low-pass time constant [tick]
STPOSCONV_setLowPassFilterTime_tick(stPosConvHandle, 3);
// ST_PosConv should start enabled
STPOSCONV_setEnable(stPosConvHandle, true);
// ST_PosConv should not be in reset
STPOSCONV_setReset(stPosConvHandle, false);
```

18.3.1.4 Call SpinTAC Position Convert

This should be done in the main ISR. This function needs to be called at the proper decimation rate for this component. The decimation rate is established by ST_ISR_TICKS_PER_SPINTAC_TICK declared in the spintac_velocity.h header file; for more information, see Section 4.7.1.4. Before calling SpinTAC Position Converter function the electrical angle computed by the ENC module needs to be passed into the SpinTAC Position Converter.

```
// update the electrical angle
STPOSCONV_setElecAngle_erev(stPosConvHandle, ENC_getElecAngle(encHandle));
// run the SpinTAC Position Converter
STPOSCONV_run(stPosConvHandle);
```

18.3.2 Troubleshooting SpinTAC Position Convert

18.3.2.1 ERR_ID

ERR_ID provides an error code for users. A list of errors defined in SpinTAC Position Convert and the solutions for these errors are shown in Table 18-2.

Copyright © 2013–2019, Texas Instruments Incorporated

Table 18-2. SpinTAC Position Convert ERR_ID Code

ERR_ID	Problem	Solution
1	Invalid sample time value	Set cfg.T_sec within (0, 0.01]
13	Invalid position rollover bound value	Set cfg.ROMax_mrev within [2, 100]
21	Invalid value for the scaling factor from [MRev] to [ERev]	Set cfg.PolePairs within [1, 32]
25	Invalid value for the scaling factor from position in [MRev] to velocity in [pu/s]	Set cfg.erev_TO_pu_ps as a positive IQ24 value
26	Invalid input sawtooth position upper bound value	Set cfg.ROMax_erev within [0, 16]
27	Invalid input sawtooth position lower bound value	Set cfg.ROMin_erev within [-16, 0]
37	Invalid input cfg.OneOverFreqTimeConst	Set cfg.OneOverFreqTimeConst to a positive value
38	Invalid input cfg.SampleTimeOverTimeConst	Set cfg.SampleTimeOverTimeConst to a positive value
1010	Invalid velocity feedback low pass filter time constant	Set cfg.LpfTime_tick within [1, 100]
4003	Invalid ROM Version	Use a chip with a valid ROM version or use the SpinTAC library that is compatible with the current ROM version.

Copyright © 2013–2019, Texas Instruments Incorporated

Definition of Terms and Acronyms

ACIM — Alternating current induction motor.

ADRC — Active Disturbance Rejection Control. Estimates and compensates for system disturbance, in real-time.

CCStudio — Code Composer Studio.

FAST — Unified observer structure which exploits the similarities between all motors that use magnetic flux for energy transduction, automatically identifying required motor parameters and providing the following motor feedback signals:

- High-quality Flux signal for stable flux monitoring and field weakening.
- Superior rotor flux Angle estimation accuracy over wider speed range compared to traditional observer techniques independent of all rotor parameters for ACIM.
- Real-time low-noise motor shaft Speed signal.
- Accurate high bandwidth Torque signal for load monitoring and imbalance detection.

FOC — Field-oriented control.

Forced-Angle — Used for 100% torque at start-up until the FAST rotor flux angle tracker converges within first electrical cycle.

InstaSPIN-FOC — Complete sensorless FOC solution provided by TI on-chip in ROM on select devices (FAST observer, FOC, speed and current loops), efficiently controlling your motor without the use of any mechanical rotor sensors.

InstaSPIN-MOTION — A comprehensive motor-, motion- and speed-control software solution that delivers robust system performance at the highest efficiency for motor applications that operate in various motion state transitions. InstaSPIN-MOTION builds on and includes InstaSPIN-FOC, combined with SpinTAC™ Motion Control Suite from LineStream Technologies.

IPM — Interior permanent magnet motor.

LineStream Technologies — Pioneers in the world of embedded controls software. Boasting a team of motor control experts from six different countries cumulatively speaking fifteen languages and possessing over eighty years of industry experience, LineStream is fast becoming the world's preeminent stronghold of embedded motor control knowledge.

Motor Parameters ID or Motor Identification — A feature added to InstaSPIN-FOC, providing a tool to the user so that there is no barrier between running a motor to its highest performance even though the motor parameters are unknown.

PI — Proportional-integral regulator.

PMSM — Permanent magnet synchronous motor.

PowerWarp™ — A mode of operation for AC induction motors (ACIM) that minimizes motor losses under lightly loaded conditions.

Rs-Offline Recalibration — InstaSPIN-FOC feature that is used to recalibrate the stator resistance, Rs, when the motor is not running.

Rs-Online Recalibration — InstaSPIN-FOC feature that is used to recalibrate the stator resistance, Rs, while the motor is running in closed loop.

Copyright © 2013–2019, Texas Instruments Incorporated

SpinTAC™ Motion Control Suite — Includes and advanced speed controller, a motion engine, and a motion sequence planner. The SpinTAC disturbance-rejecting speed controller proactively estimates and compensates for system disturbances in real-time, improving overall product performance. The SpinTAC motion engine calculates the ideal reference signal (with feed forward) based on user-defined parameters. SpinTAC supports standard industry curves, and LineStream's proprietary "smooth trajectory" curve. The SpinTAC motion sequence planner operates user-defined state transition maps, making it easy to design complex motion sequences.

SVM — Space-vector modulation.

Copyright © 2013–2019, Texas Instruments Incorporated

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from G Revision (February 2017) to H Revision	Page
• Table was added to the Preface of the document.	20
• Update was made in Chapter 1.	23
• Description was changed for *EST_setMaxCurrentSlope_pu ()* in Section 3.3.2.2.	107
• Updates were made in Section 4.2.3.	207
• Updates were made in Section 5.2.4.	234
• Figure 17-2 was updated in Chapter 17.	558

Copyright © 2013–2019, Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

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

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

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

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2019, Texas Instruments Incorporated