Sensorless field-oriented control (FOC) using PSoC™ 6 MCU

About this document

Scope and purpose

This application note shows how to control a permanent magnet synchronous motor (PMSM) with the sensorless field-oriented control (FOC) algorithm, using an Arm® Cortex®-M4-based PSoC™ 6 device.

Intended audience

This application note is intended for designers of motor control systems.

This application note assumes that you are familiar with PSoC™ 6 and the ModusToolbox™. If you are new to PSoC™ 6, see **AN228571 - Getting started with PSoC™ 6 MCU on ModusToolbox™**. If you are new to ModusToolbox™, see the **ModusToolbox™ home page**.

You should also understand motor control fundamentals; start with "**electric motor**" on Wikipedia.

Table of contents

Table of contents

1 Introduction

The FOC algorithm is frequently used in motor control applications because it allows motors to operate with less noise and more stable torque output than other algorithms. Sensorless FOC adds the advantage of reducing the cost due to the absence of rotor position sensors. Sensorless FOC is used in many applications including consumer (air conditioner, refrigerator), industrial (blower, pump), and commercial (elevator, escalator) products.

Sensorless FOC is calculation-intensive, and thus has been traditionally implemented with expensive digital signal processing (DSP) devices. However, with 32-bit Arm® Cortex®-M cores, it is possible to implement sensorless FOC with more cost-effective 32-bit MCUs.

This application note includes a code example to be used with the Infineon **CY8CKIT-037 motor control evaluation kit** which includes a 24-V 53-W PMSM motor.

Note: The CY8CKIT-037 kit board can operate at voltages as high as 48 VDC, and some components may operate at high temperatures. Use this kit with caution to avoid personal injury or equipment damage.

1.1 Abbreviations and definitions

Table 1 Abbreviations

Abbreviation	Meaning
BLDC drum	Brushless DC drum
DD drum	Direct drive drum
FOC	Field-oriented control
SVPWM	Space vector pulse width modulation
HVIC	High-voltage IC
CW	Clockwise
CCW	Counterclockwise
PMSM	Permanent magnet synchronous motor

2 Sensorless FOC basics

This section introduces the hardware structure of a typical sensorless FOC system as well as a firmware overview of the FOC algorithm. If you are familiar with these concepts, you can skip this section and go to the **Code example** section.

Figure 1 shows the diagrams of the two types of the PMSM motor; they differ in how magnets are placed in the rotor:

- Surface PMSM (SPMSM) – Left
- Interior PMSM (IPMSM) – Right

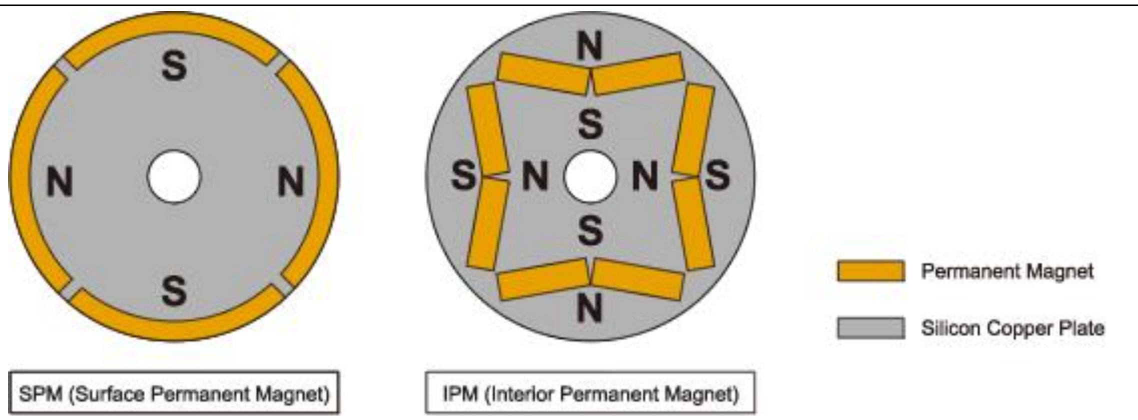


Figure 1 Rotor structure for SPMSM and IPMSM

SPMSM is widely used due to the ease of manufacture and assembly, while IPMSM has a larger torque output with the same-sized motor. The sensorless FOC algorithm varies depending on the motor type; this application note uses SPMSM, referred to as just "PMSM".

Figure 2 shows the hardware block diagram of a typical sensorless FOC system. It consists of:

- MCU
- Inverter
- PMSM
- Current sampling and signal conditioning circuit to determine the rotor position
- Communication interface

These components can be on the same controller board or separated in the system such as on an MCU board and an inverter board.

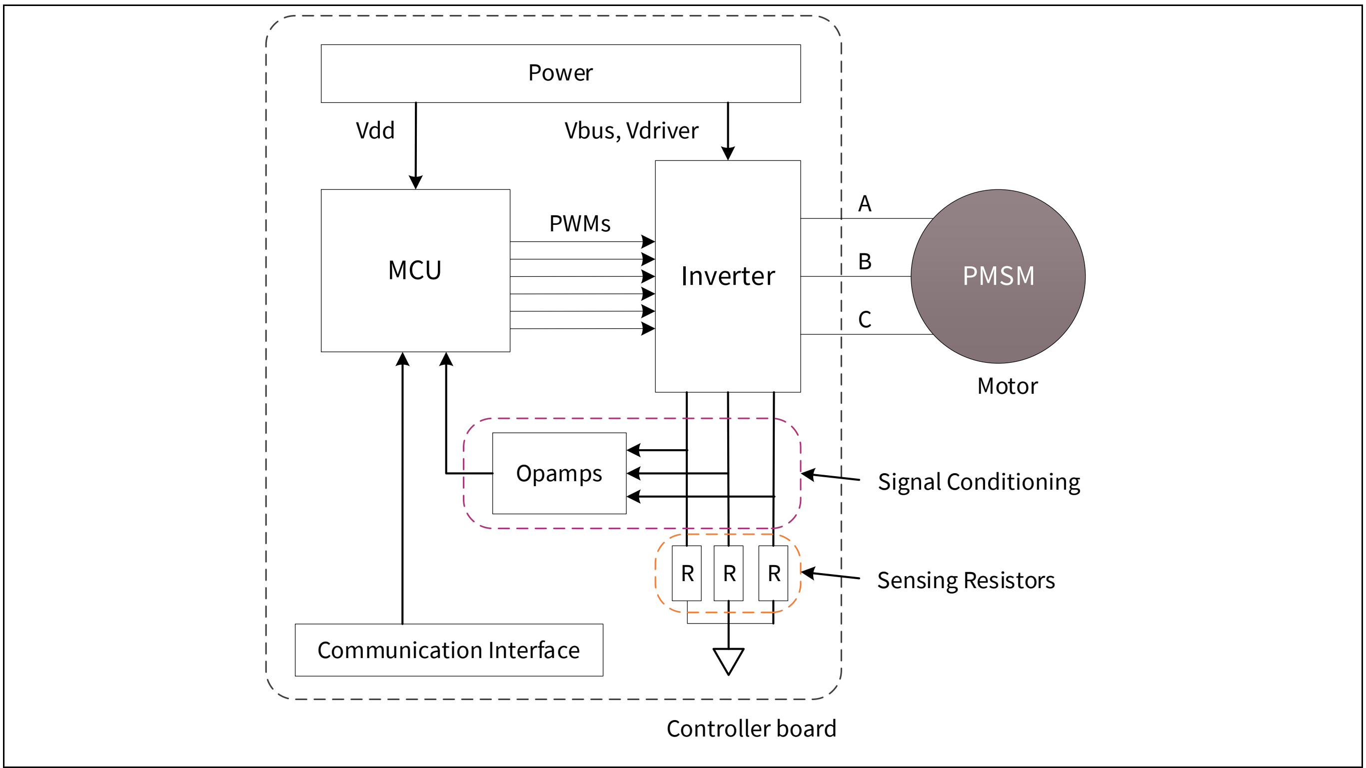


Figure 2 **Overview of a typical sensorless FOC system**

Figure 3 shows the details of the Inverter block shown in **Figure 2**. The inverter is composed of gate drivers and six MOSFETs (two for each motor phase). Turning different MOSFETs ON or OFF changes the current direction through the motor's stator windings or phases.

For example, turning on Q1 and Q4 generates a current from phase A to phase B, while turning on Q3 and Q2 reverses the current direction in those phases. Changing the current direction changes the stator flux direction and makes the rotor rotate.

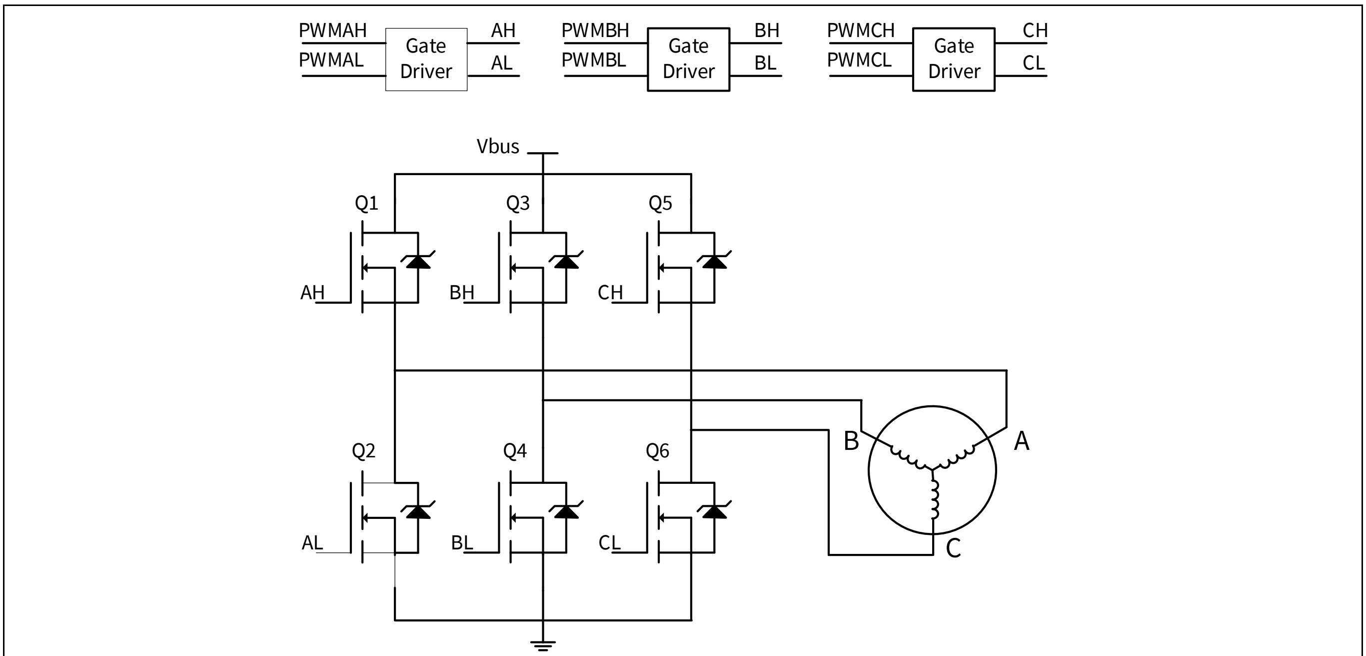


Figure 3 **Details of inverter block**

Vbus is a higher-voltage DC supply to power the motor. For example, it is 24 V in the **CY8CKIT-037 kit**.

Note that the pairs of MOSFETs on the same phase (for example, Q1 and Q2) must not be turned ON at the same time – the resultant low resistance causes high currents that can damage the MOSFETs.

Figure 4 and **Figure 5** show diagrams of the sensorless FOC algorithm and its calculation flow. The algorithm controls either the motor speed or motor torque using a proportional-integral (PI) controller based on a mathematical model of the PMSM (see **Appendix A: PMSM model**). The control result is sent to a Space Vector Pulse Width Modulation (SVPWM) block (see **SVPWM theory**). The SVPWM block generates three-phase voltages that change the stator currents.

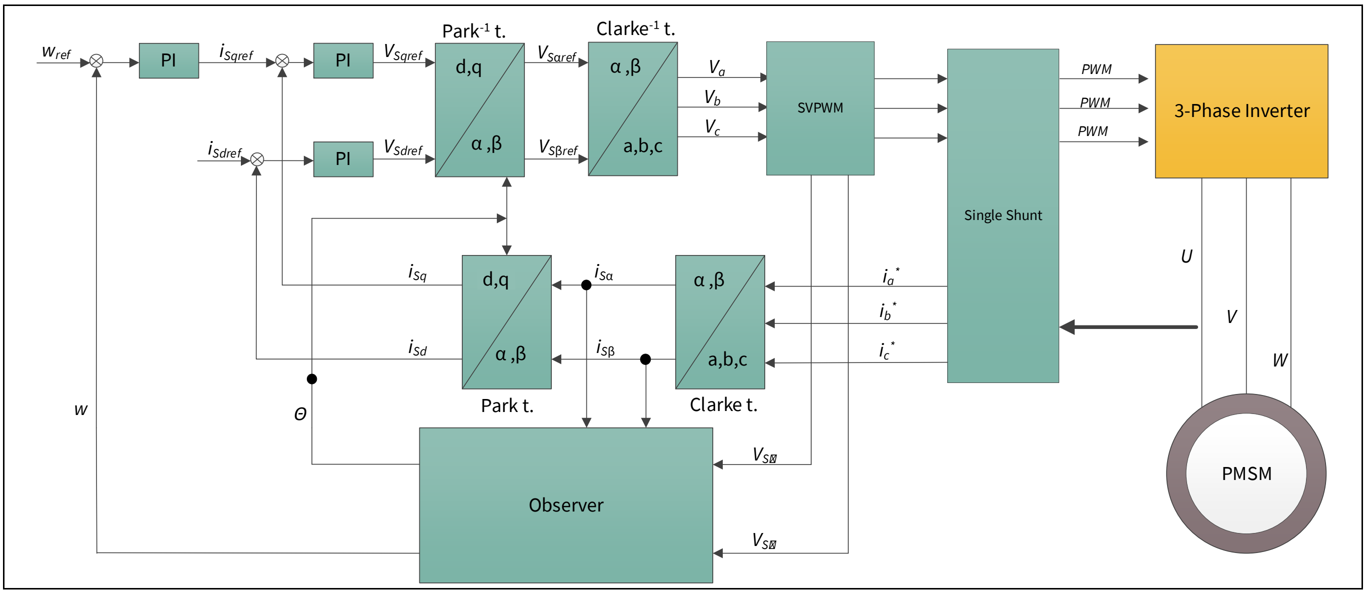


Figure 4 **Sensorless FOC control block diagram**

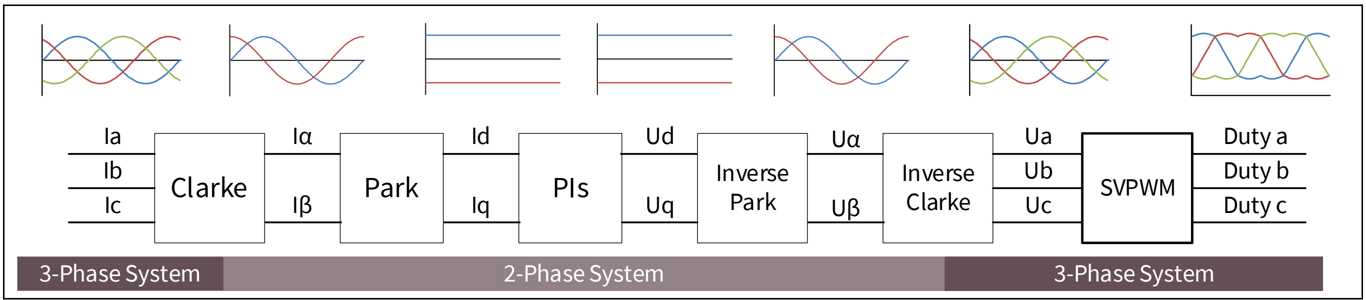


Figure 5 **Sensorless FOC calculation flow**

The Clarke and Park transformation calculations convert these three sampled motor phase currents into two values that are used by the PI controller. The Inverse Clarke and Inverse Park transformations are the opposites of the Clarke and Park transformations, respectively.

Figure 6 shows the Clarke transformation, where the three motor phase currents ($\mathbf{i_a}$, $\mathbf{i_b}$, $\mathbf{i_c}$) are converted to $\mathbf{i_\alpha}$ and $\mathbf{i_\beta}$. The ($\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$) frame is a three-phase stator reference frame, where the axes are 120° apart from each other. The transformation method is to project ($\mathbf{i_a}$, $\mathbf{i_b}$, $\mathbf{i_c}$) onto the ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$) axes, which produce the outputs $\mathbf{i_\alpha}$ and $\mathbf{i_\beta}$.

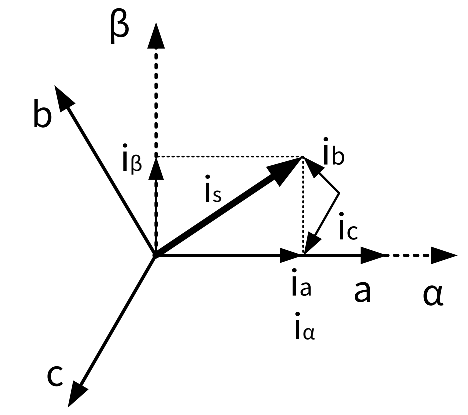


Figure 6 Clarke transformation

Figure 7 shows the details of the Park transformation. This transformation converts the current vectors from the Clarke transformation, $\mathbf{i_\alpha}$ and $\mathbf{i_\beta}$, to a frame on the rotating part of the motor. The axes of the rotating frame are called (d, q). The current vectors on these axes are called $\mathbf{i_d}$ and $\mathbf{i_q}$.

$\mathbf{\Psi_f}$ is the flux linkage vector of the rotor magnet. The d axis is always aligned with $\mathbf{\Psi_f}$, and the q axis is at 90° to the d axis. The rotor rotates at an angular speed $\mathbf{\omega_r}$, and $\mathbf{\theta_r}$ is the angle between the α and d axes.

In the (d, q) frame, the motor torque is proportional to $\mathbf{i_q}$. You can control $\mathbf{i_q}$ to achieve the desired torque by using the **PI controllers**. For details on the Clarke and Park transformations as well as the torque output, see **Appendix A: PMSM model**.

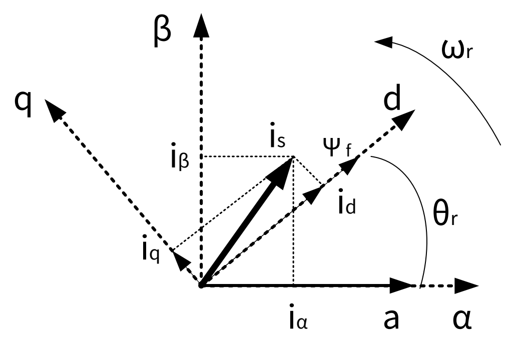


Figure 7 Park transformation

The angle $\mathbf{\theta_r}$ used in the Park transformation is derived from the speed and position estimation. An algorithm called "Slide Mode Observer" (SMO) uses $\mathbf{i_\alpha}$ and $\mathbf{i_\beta}$ to derive the $\mathbf{\theta_r}$ value. Then, the angular speed $\mathbf{\omega_r}$ is calculated based on $\mathbf{\theta_r}$. For more information, see the **Slide mode observer (SMO)** section.

3 Code example

3.1 Features

- Implements the sensorless FOC algorithm and closed-loop speed control in a multilayer, extensible, binary library architecture
- Estimates the rotor position with the Slide Mode Observer (SMO) algorithm
- Uses PSoC™ 6 internal opamps and the 1-Msps successive approximation register (SAR) ADC for signal conditioning and measuring the motor phase current
- Employs open-loop control at startup, which is changed to closed-loop control after the rotor position is determined
- Supports motor speeds from 500 to 4000 rpm by default. Can support higher speeds in other motors by modifying the tuning parameters in the code example
- Can adjust the motor speed by using the potentiometer on the kit
- Provides control accuracy 5% over the default speed range. Using high-resolution sensing resistors and advanced control algorithms can improve the accuracy; this topic is outside the scope of this application note.

3.2 Design overview

Figure 8 illustrates the sensorless FOC implementation in PSoC™ 6. A 12-bit SAR ADC and two opamps are used to sample the motor phase currents (only two phase currents need be sampled; the third phase can be calculated from the other two.) The three TCPWMs generate six PWM outputs applied to the inverter. A serial communication block (SCB) implements a UART to communicate with the host. See **Design details** in Chapter 3.

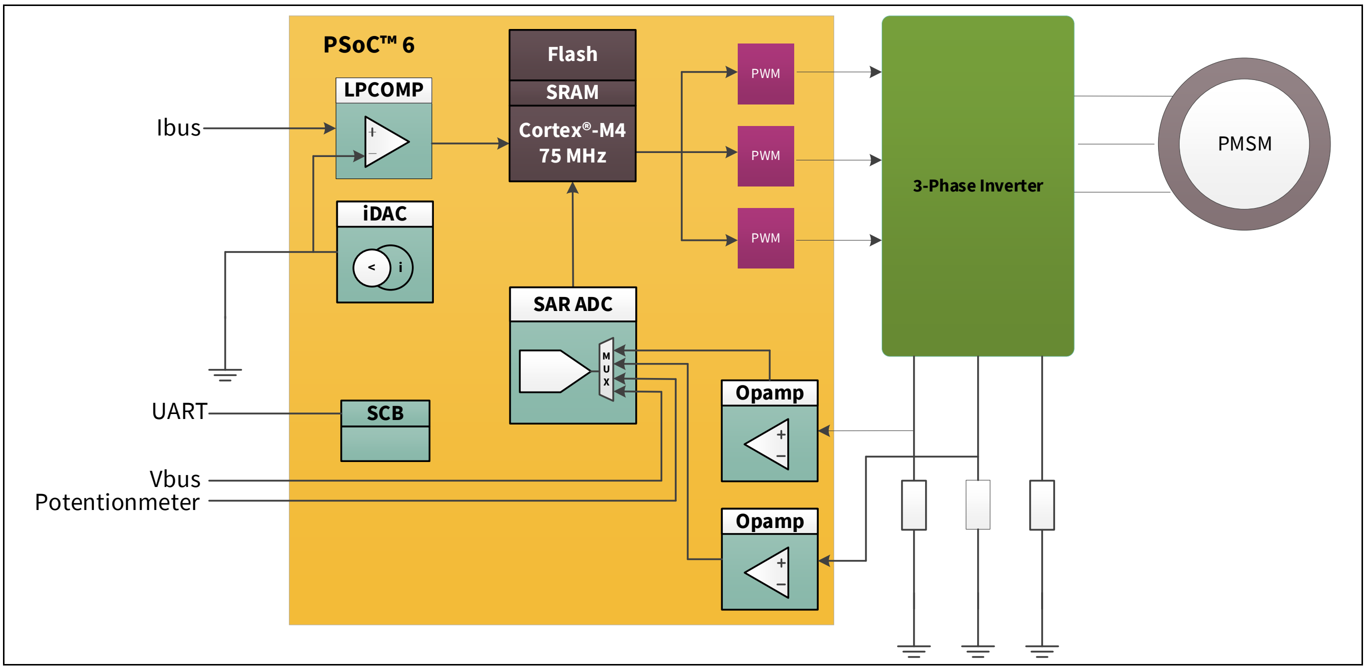


Figure 8 **PSoC™ 6 sensorless FOC implementation**

Table 2 shows the PSoC™ 6 resources that are used by this code example:

Table 2 Resource usage summary

Item	Used	Available	Usage
CPU frequency	75 MHz	150 MHz	Internal system clock
PWM frequency	10 kHz	5 kHz~20 kHz	NA
Flash	37084 bytes	256 KB	NA
SRAM	9135 bytes	128 KB	NA
Interrupts	3	140	Generates interrupts that system need
TCPWM blocks	4	12	Three TCPWM are used to generate 3-phase signals to control the PMSM drive. Another TCPWM is used to generate a trigger pulse for the ADC for current measurement.
Opamp	1	2	Used to amplify the voltage from the current sense resistors prior to feeding the voltages to ADC inputs
UART	1	7	Reserved for communication with the host
Low-power comparators	1	2	Used for overcurrent protection
8-bit current DAC (IDAC)	1	2	Generates the source current for overcurrent protection
12-bit SAR ADC channels	4	16	Used to transfer the phase current sampled value to digital signals
Other Pins	2	10	Pin_Led: The GPIO to control LED; Pin_Dir: The GPIO to control the motor running direction

3.3 Firmware

Figure 9 shows the firmware execution flow. The FOC algorithm requires the PWM duty cycle to be updated every control cycle. Therefore, FOC calculations must be done in a periodic interrupt service routine (ISR). The ISR is triggered by the PWM every 100 µs (10-kHz PWM) – this is the control cycle period.

The cycle period can be decreased by increasing the PWM frequency. A shorter control period results in a higher-bandwidth control system with two benefits:

- Motor can be run faster
- Better response to load changes

The communication and other functions that do not require real-time processing are executed in the main loop.

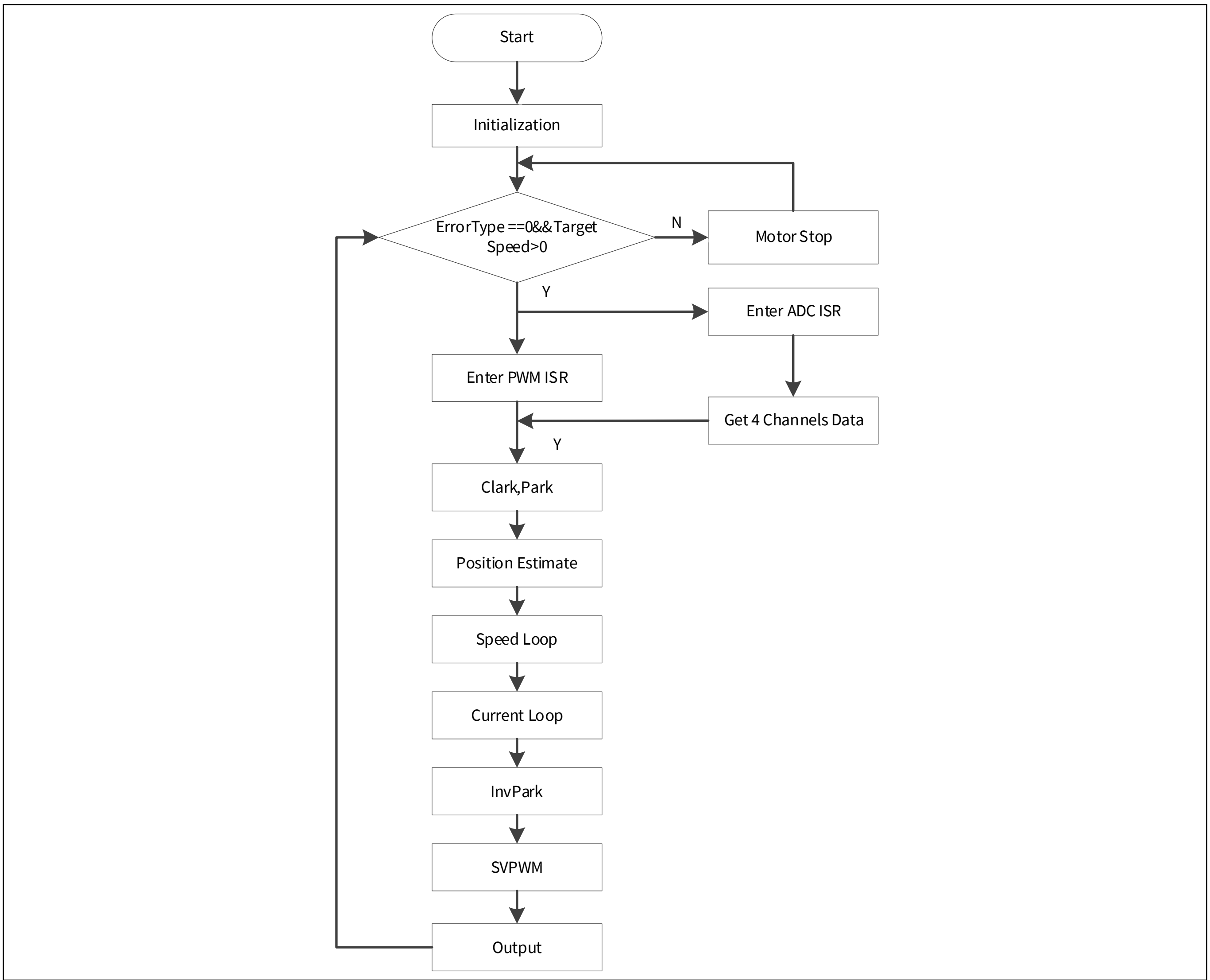


Figure 9 Firmware execution flow

3.4 CY8CKIT-037 kit

The CY8CKIT-037 kit is a motor-driver board designed to support three control algorithms: trapezoidal, FOC, and microstepping control for stepper motors. It has no MCU; it is a peripheral board to be used with the **CY8CKIT 062** (**Figure 11**), through the interface compatible with Arduino. For more information, see the **CY8CKIT-037 user guide**.

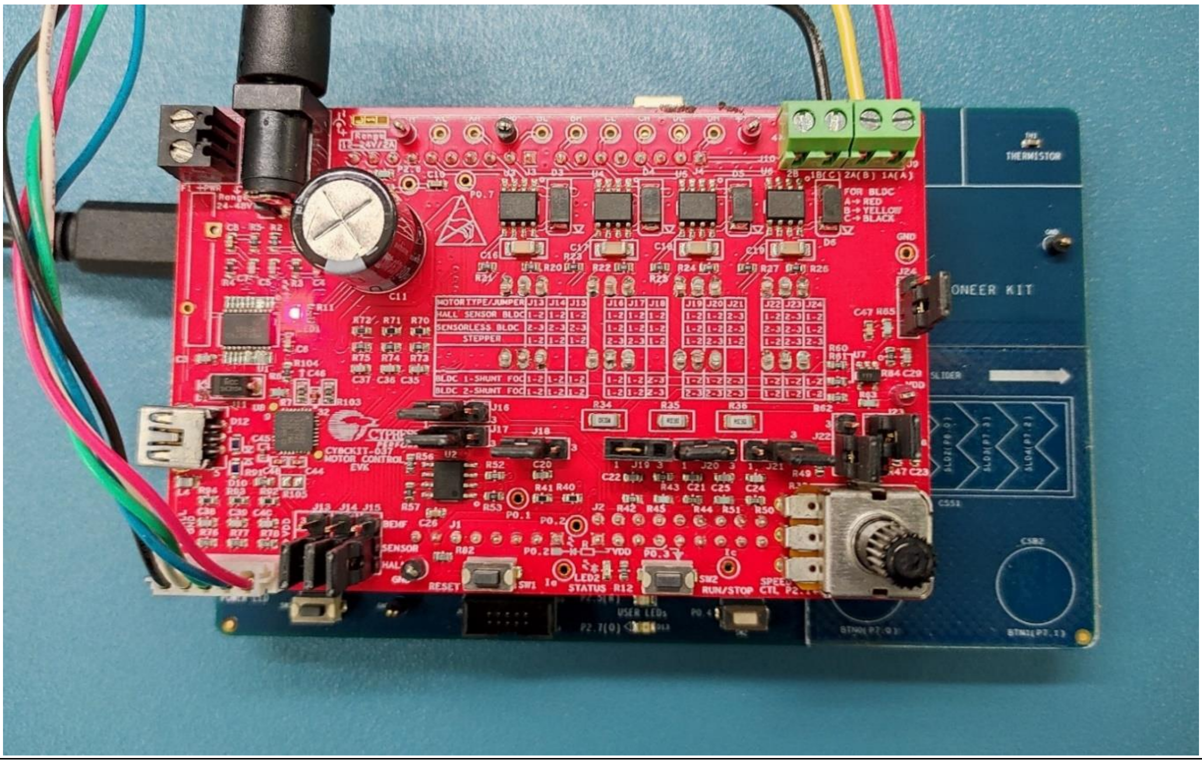


Figure 10 **CY8CKIT-037 kit**

A PMSM, manufactured by **Anaheim Automation**, is included in this kit. **Table 3** lists the motor parameters. See **Appendix B: Adapting the design to other motors** for information on how to change the code example by changing the motor parameters listed in this table.

Table 3 **Parameters for the motor in CY8CKIT-037**

Item	Parameter
Part number	BLY172S-24V-4000
Rated torque (Newton.meter)	1.26
Rated voltage (V)	24
Rated power (watts)	52
Rated speed (RPM)	4000
Torque constant (Newton.meter/A)	0.35
Back EMF voltage (V/kRPM)	3.72
Line-to-line resistance (ohm)	0.8
Line-to-line inductance (mH)	1.2
Rotor inertia (Newton.meter/sec^2)	0.000680
"L" length (cm)	6.02
Shaft	Single

3.5 Operation

3.5.1 Step 1 – Configure CY8CKIT-062S4

Select 3.3 V as the VDD power at jumper J9 on CY8CKIT-062S4, as **Figure 11** shows.

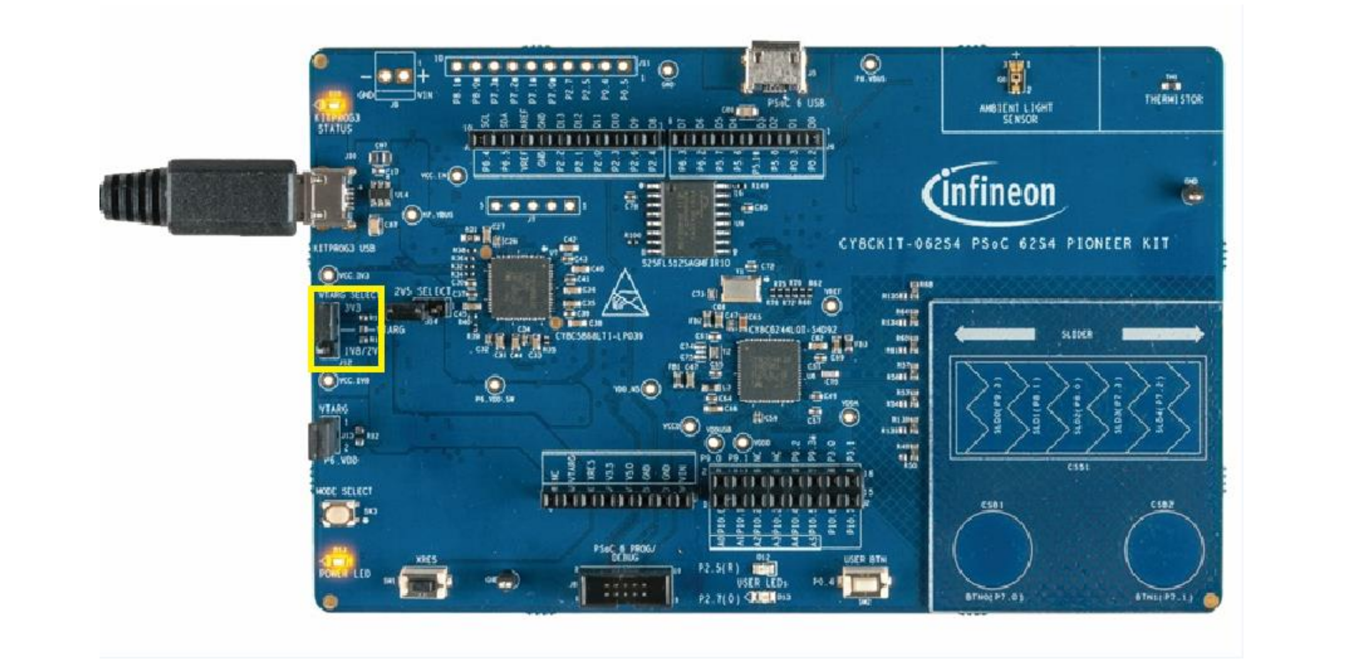


Figure 11 CY8CKIT-062S4 configuration

3.5.2 Step 2 – Configure CY8CKIT-037

Configure the board via jumpers J13-J24 as listed in the row "BLDC 2-SHUNT FOC" printed on the board. See **Figure 12** and **Figure 13**.

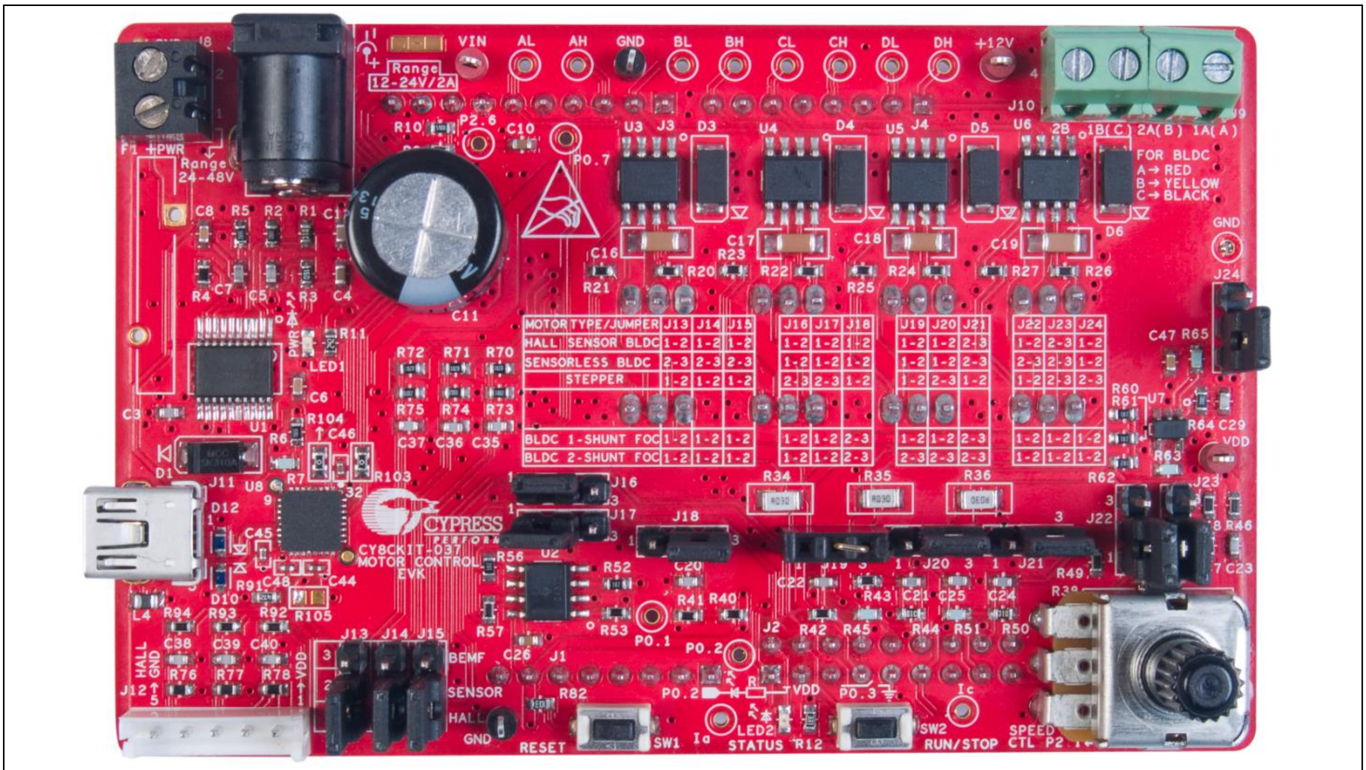


Figure 12 CY8CKIT-037 configuration for sensorless FOC motor control

MOTOR TYPE/JUMPER	J13	J14	J15	J16	J17	J18	J19	J20	J21	J22	J23	J24
HALL SENSOR BLDC	1-2	1-2	1-2	1-2	1-2	1-2	1-2	1-2	2-3	1-2	2-3	1-2
SENSORLESS BLDC	2-3	2-3	2-3	1-2	1-2	1-2	1-2	1-2	2-3	2-3	2-3	1-2
STEPPER	1-2	1-2	1-2	2-3	2-3	1-2	1-2	2-3	1-2	1-2	2-3	2-3
BLDC 1-SHUNT FOC	1-2	1-2	1-2	1-2	1-2	2-3	1-2	1-2	2-3	1-2	1-2	1-2
BLDC 2-SHUNT FOC	1-2	1-2	1-2	1-2	1-2	2-3	2-3	2-3	2-3	1-2	1-2	1-2

Figure 13 **Jumper table for CY8CKIT-037**

3.5.3 Step 3 – Plug CY8CKIT-037 into CY8CKIT-062S4

- Plug the CY8CKIT-037 into the CY8CKIT-062S4 via connectors compatible with Arduino, as **Figure 10** shows.

3.5.4 Step 4 – Connect the power supply and motor

- Connect the BLDC motor to J9 and J10 on CY8CKIT-037. The other motor cable routes the signals from the sensors inside the motor.

Note: *The kit hardware supports sensored BLDC motors and sensored FOC. Because this is a sensorless example, you do not need to connect this cable. Connect the 24-V power adapter to J7. See **Figure 14**.*

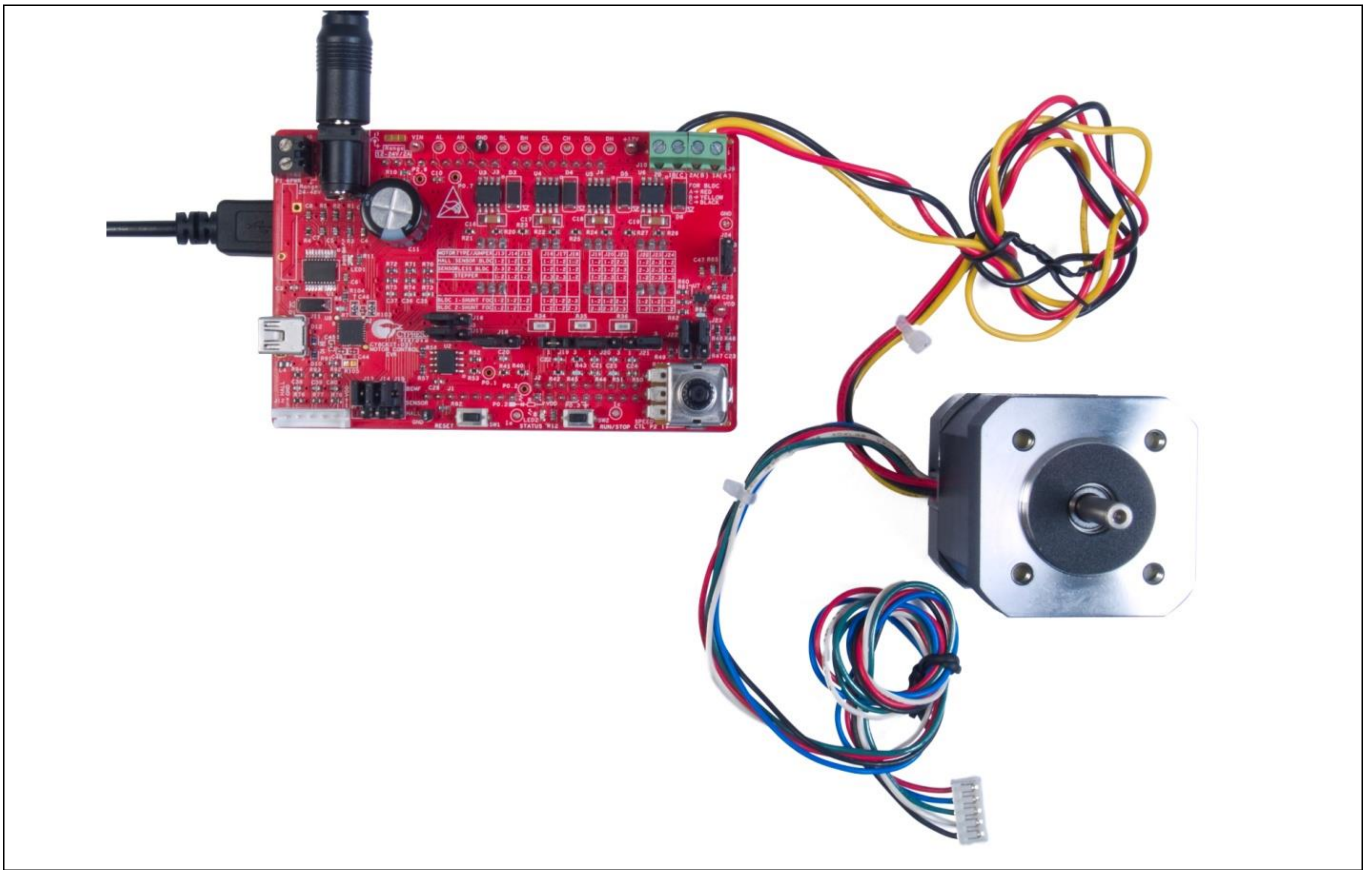


Figure 14 **Connect motor and power supply**

3.5.5 Step 5 – Build the project and program the PSoC™ 6 device

1. Open the sensorless FOC motor control code example project provided with this application note in ModusToolbox™ 2.4 or later.
2. Select Build Sensorless FOC Motor Control application in Quick Panel.
3. When the build is complete, select Generate launches for this project to generate the debug link in Quick Panel then choose your debug tools for program.

For more information about how to use ModusToolbox™, see to the **ModusToolbox™ home page**.

3.5.6 Step 6 – Rotate the potentiometer to start motor rotation

1. Rotate the potentiometer R38 to start and change the motor rotation speed (see **Figure 15**).
2. If the motor does not rotate, it indicates that an error has occurred. If so, first ensure that step 1 through step 5 have been executed correctly.
3. Then press the Reset button on CY8CKIT-062S4 and rotate the potentiometer R38 again.

   If the motor still does not rotate, there must be a problem in the hardware or software. Debug it using a multimeter or oscilloscope to observe the signals, or set breakpoints to monitor the variables. You can also contact Infineon for technical support.

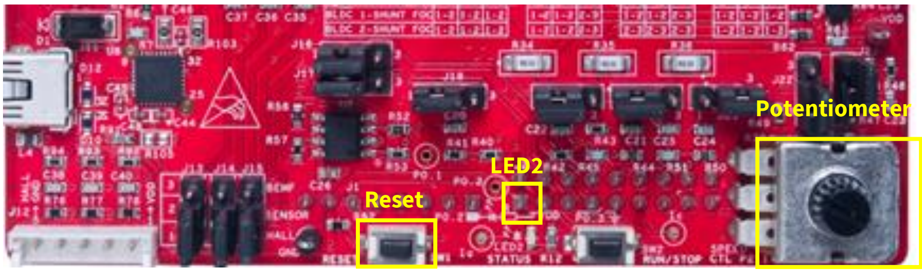


Figure 15 Buttons and status LED

3.6 Performance

Figure 16 to **Figure 18** show one of the phase currents for different motor speeds using the motor provided in the kit. **Figure 19** shows the phase current during startup.

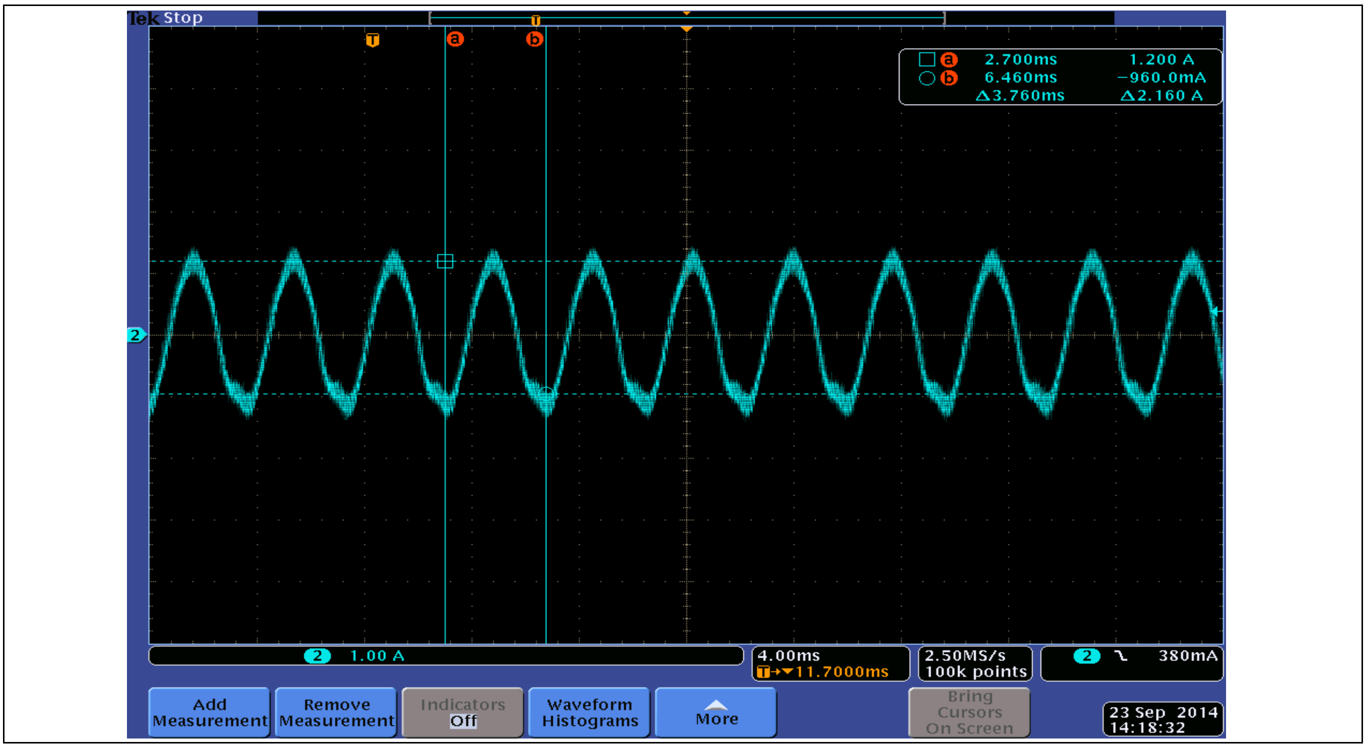


Figure 16 Phase current – 600 RPM

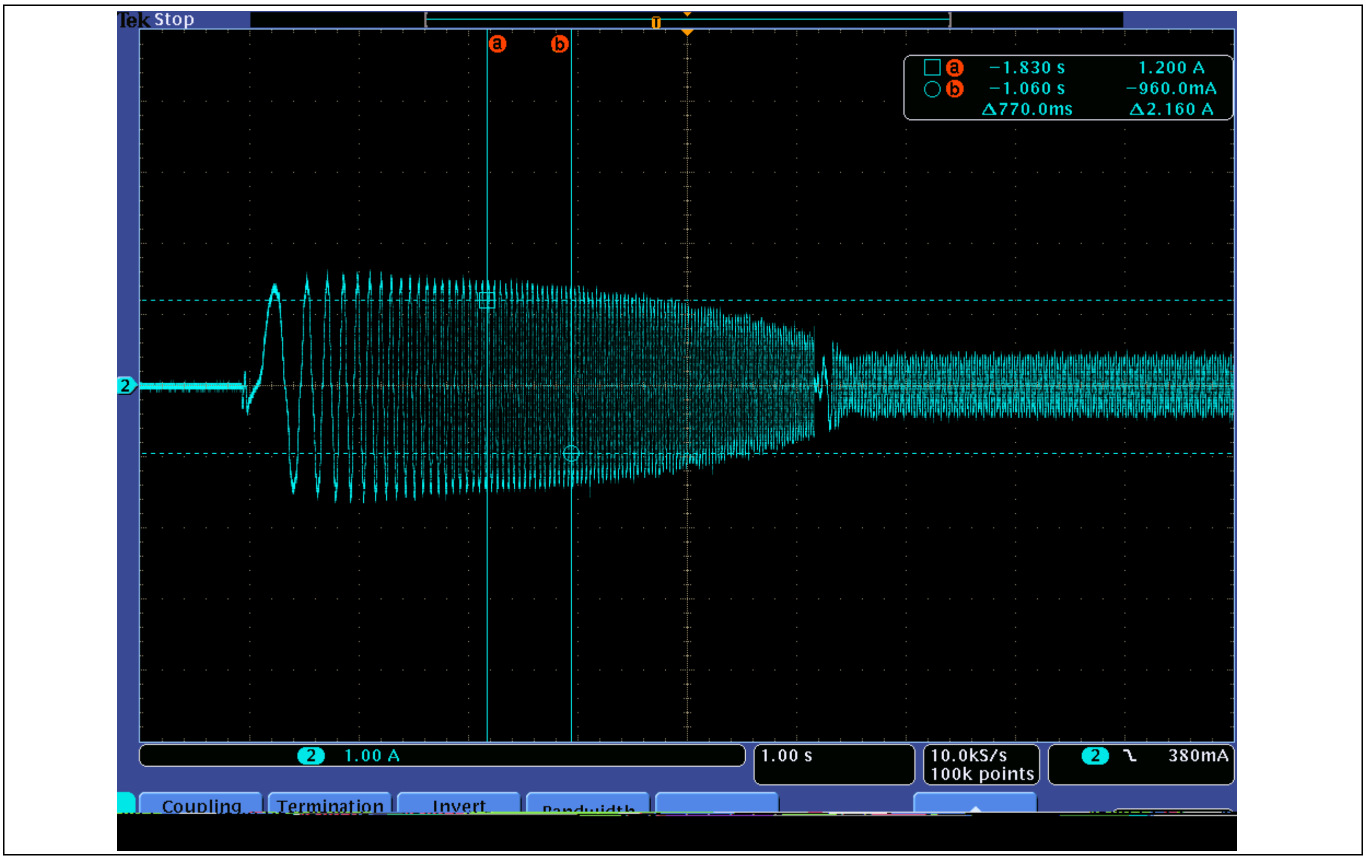


Figure 17 Phase current – 2000 RPM

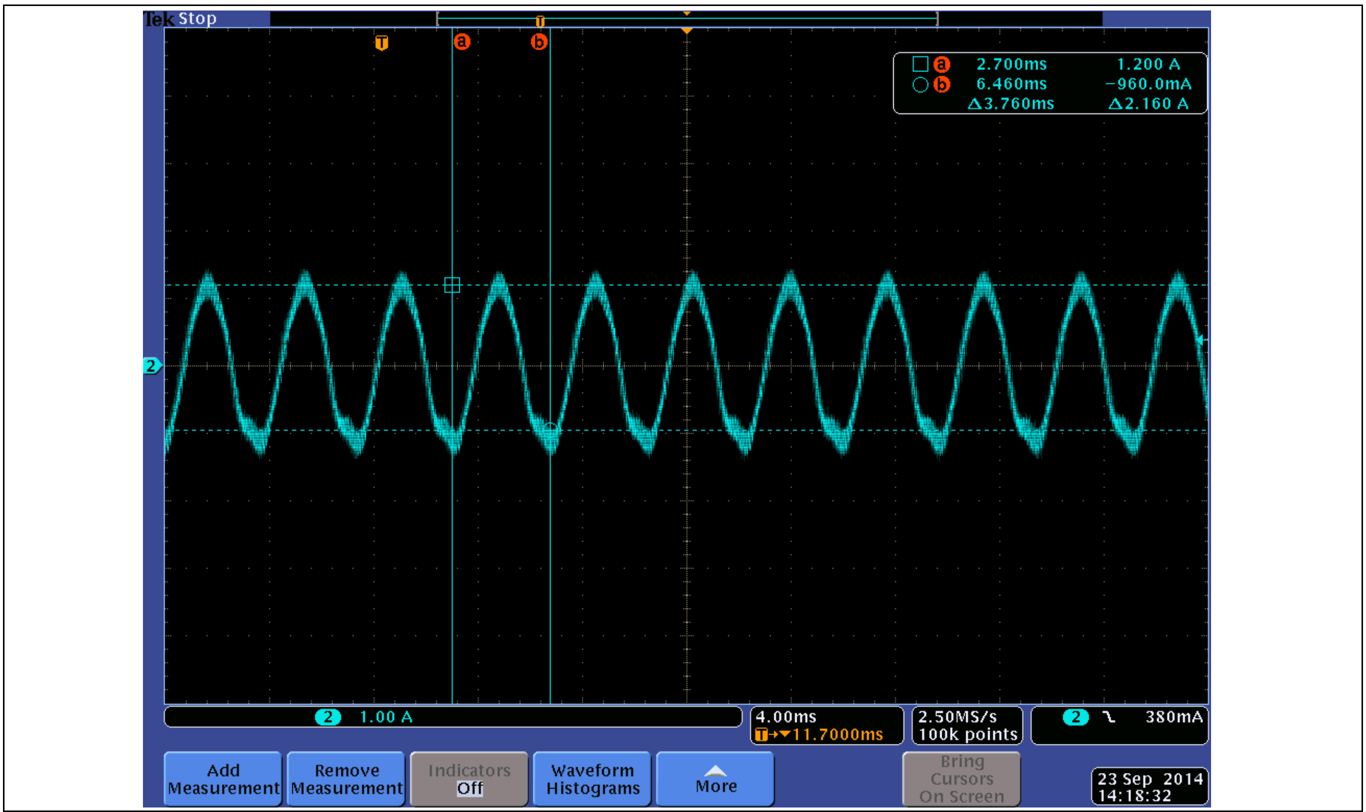


Figure 18 Phase current – 4000 RPM

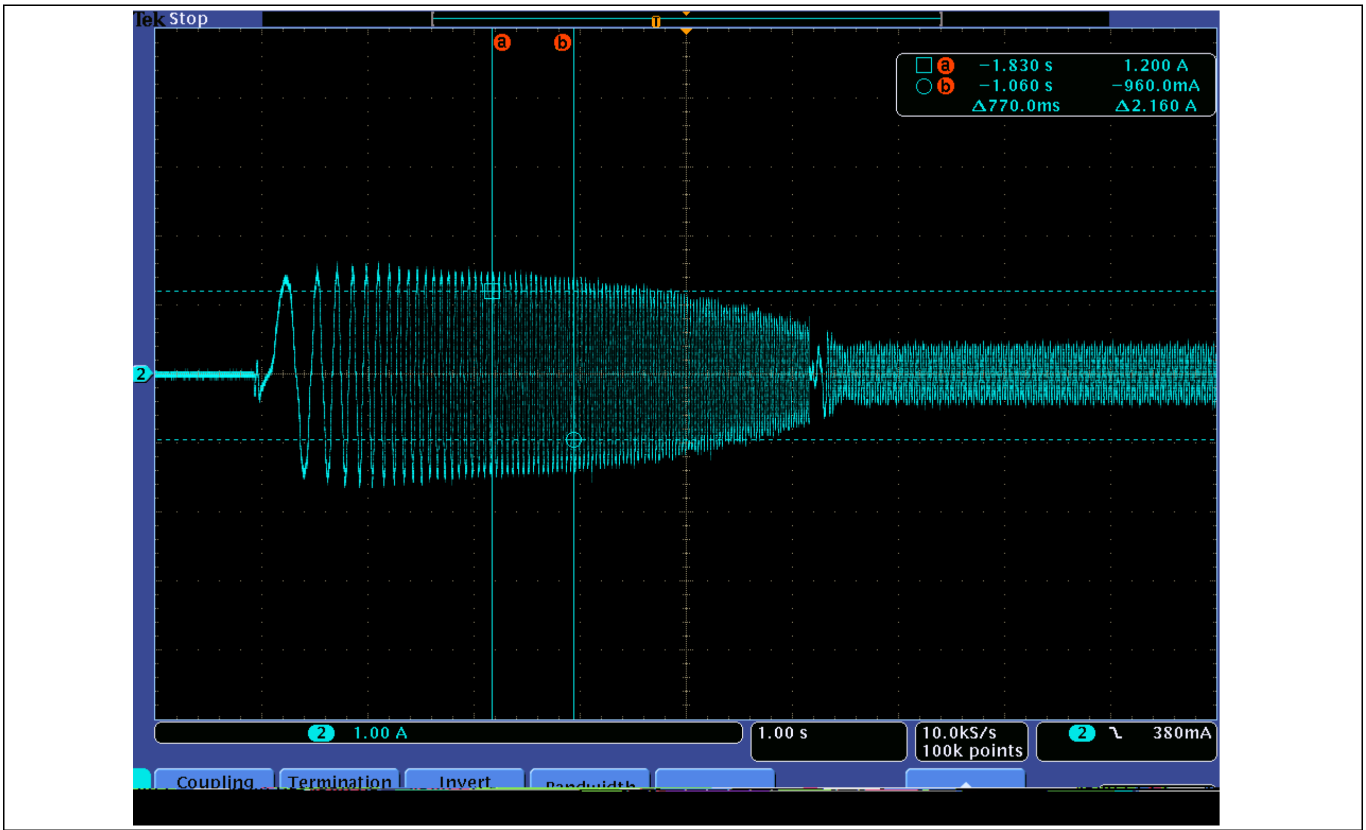


Figure 19 Phase current at startup

4 Design details

This section presents implementation details for each stage of the sensorless FOC processing listed in the PWM ISR (**Figure 9**), including current sampling, Clarke and Park transformations, SMO, PI controller, and SVPWM.

4.1 Current sampling

This section introduces the ADC sampling function in sensorless FOC motor control. In the project associated with this document, ADC sampling is realized by the internal SAR ADC component; there are several parameters need to be sampled:

- Phase winding currents: ADC0_Ia and ADC0_Ic
- Bus voltage
- Voltage input from the variable resistor (potentiometer)

Figure 20 and **Figure 21** show the SAR ADC configuration with the following features:

- 25-MHz sampling clock for a 1-Msps sampling rate
- Voltage reference as VDDA/2 to obtain a 0-to-VDDA input range
- All channels are single-ended.
- The sampling result is unsigned.
- A hardware trigger starts sampling. After four channels are sampled, the ADC stops and waits for the next trigger signal. The trigger frequency is 10 kHz. The PWMs provide a common timing for ADC sampling, CPU interrupt, and MOSFET control.

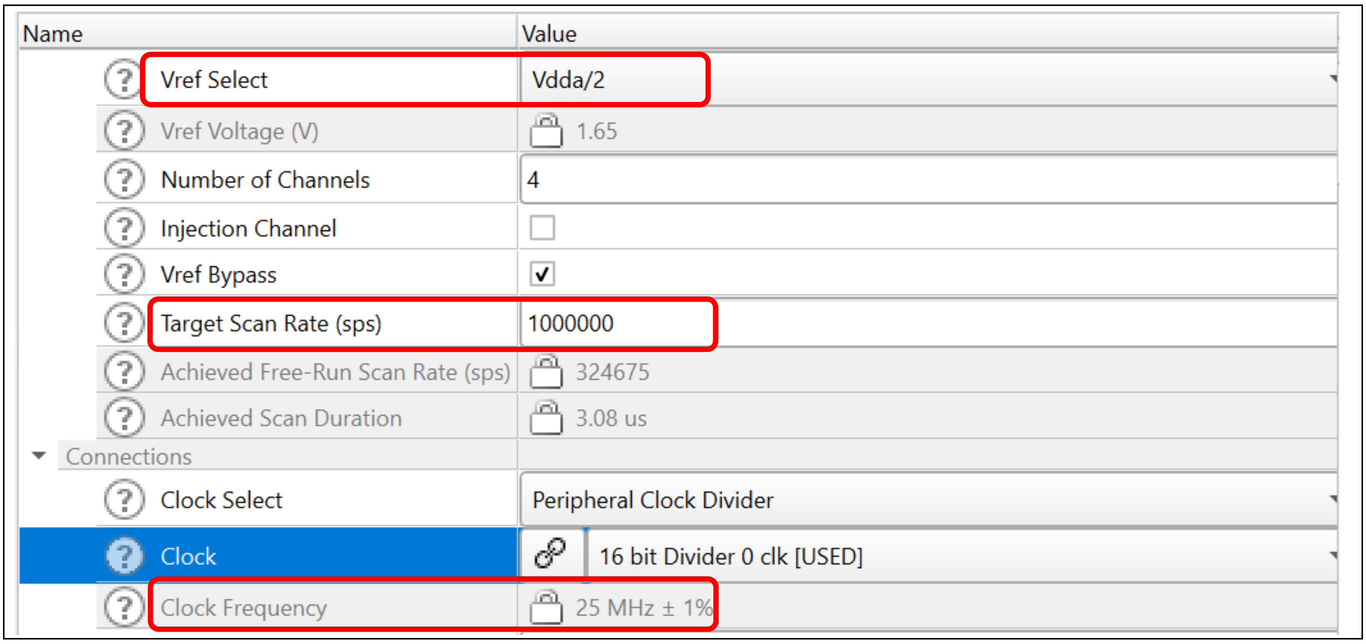


Figure 20 SAR ADC configuration(a)

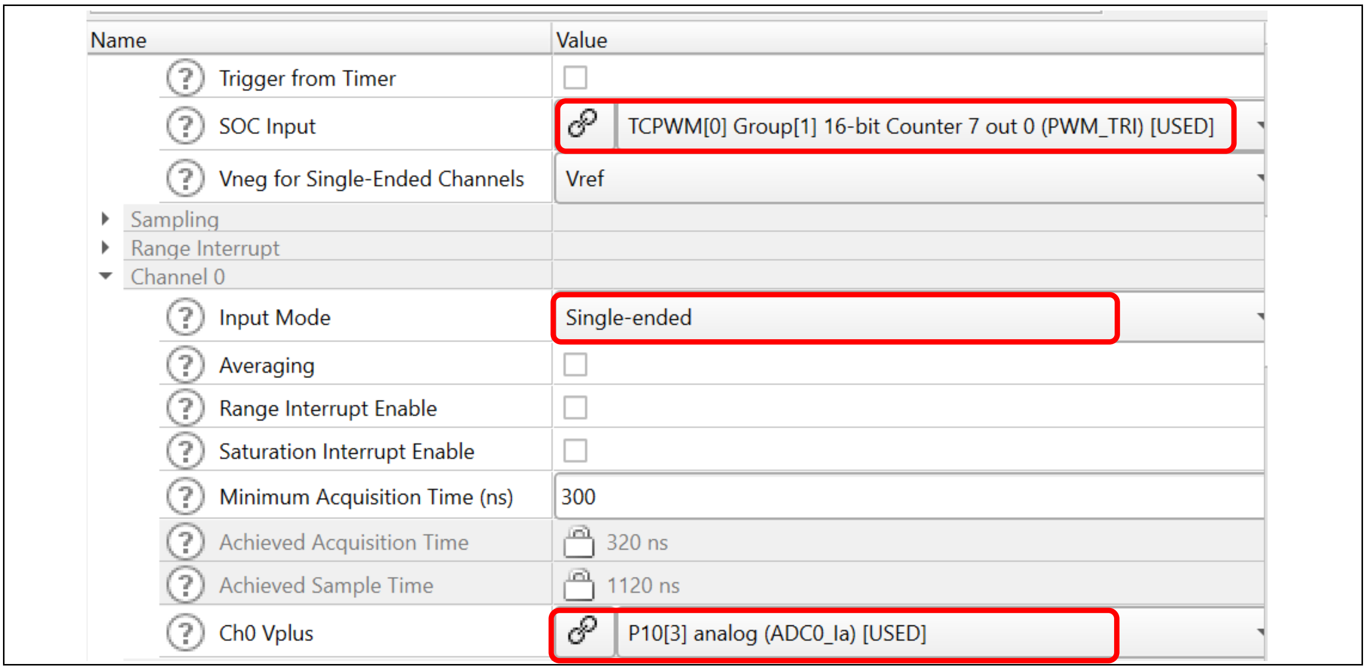


Figure 21 SAR ADC configuration(b)

The motor phase current is converted to a voltage by the sensing resistors, as **Figure 22** shows. The figure also shows that because the sum of the three currents must be zero at the sampling point, you can sample just two of the currents and calculate the third.

The opamp gains and the sensing resistor values are selected so that:

- The voltage stays in the ADC input range when the current is at the rated maximum. Sensing resistors are typically on the order of milliohms.
- The measurement of low currents is accurate. The sensing resistors have a tolerance of 1%.

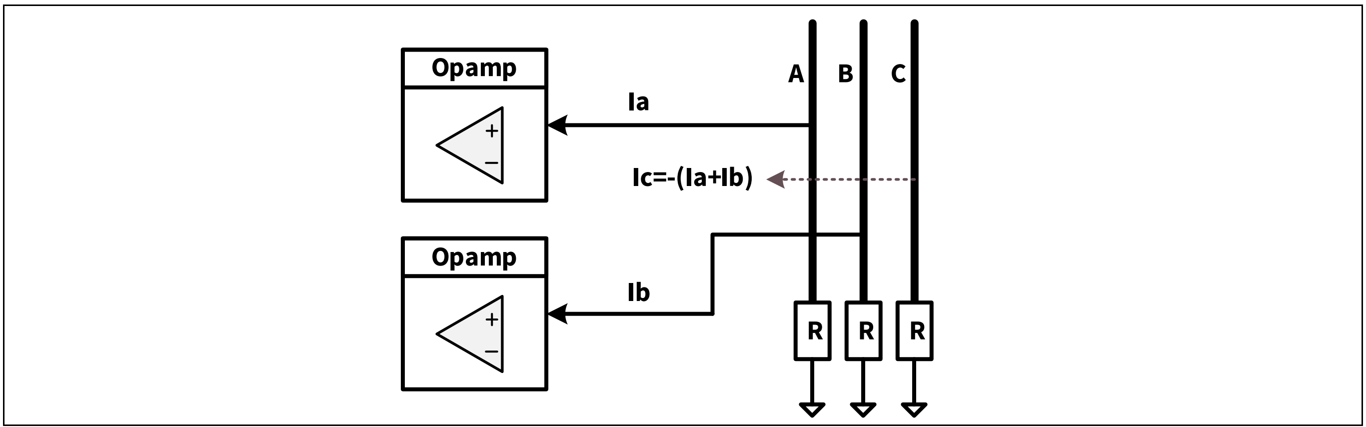


Figure 22 Dual-shunt current sampling

Figure 23 shows the schematic design for the CY8CKIT-037 kit. The kit board has 30-mΩ sensing resistors (not shown) and a 2.1-A rated current. Bias resistors (R40, R41) are included to handle positive and negative currents.

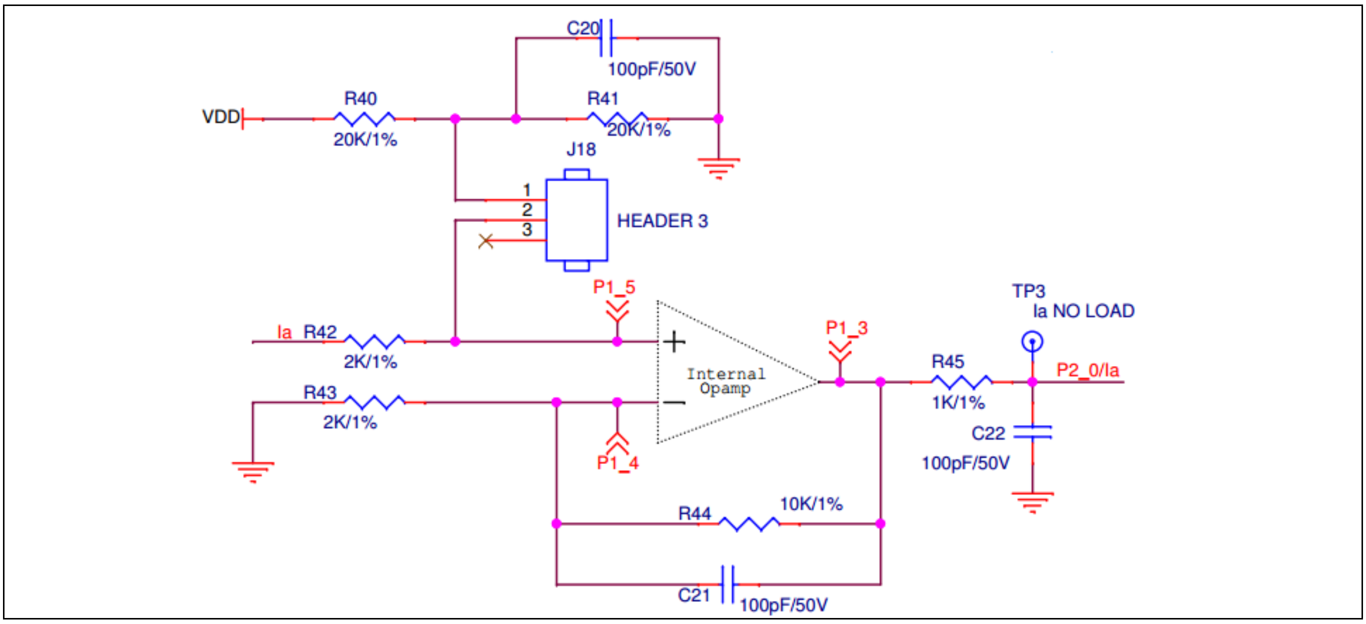


Figure 23 CY8CKIT-037 schematic: Signal conditioning for phase-A current

Due to the reuse of PSoC™ 6 MCU, only one internal opamp of this chip is used; the other opamp is an external opamp. **Figure 24** shows the configuration of the internal opamp.

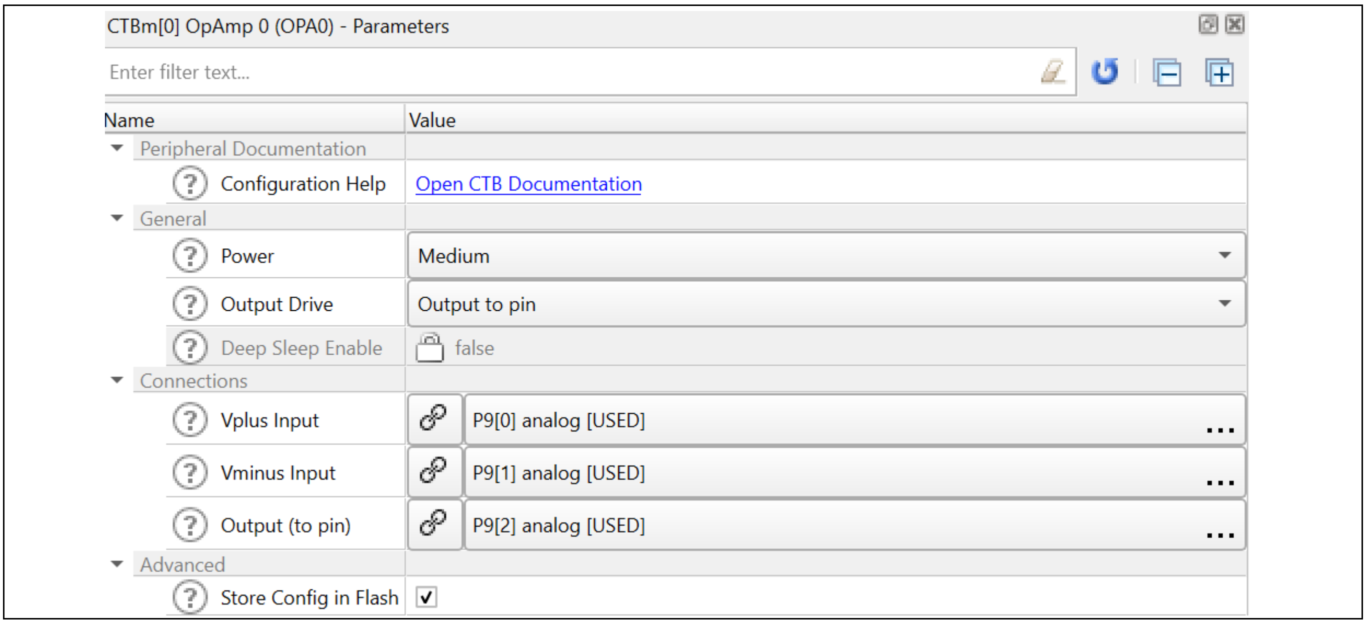


Figure 24 PSoC™ 6 internal opamp configuration

4.2 Transformations

Four functions are defined to do the transformations. The structures and function prototypes are declared in the motor control library file (*coordinate_transform.h*):

Code Listing 1 Clarke and Park transformation structures and function prototypes

```
/* coordinate_transform.h*/
/* struct definition for coordinate transformation*/
```

Code Listing 1 Clarke and Park transformation structures and function prototypes

```
typedef struct
{
    int32_t i32Q8_Xu;   /*Phase U variable*/
    int32_t i32Q8_Xv;   /*Phase V variable*/
    int32_t i32Q8_Xw;   /*Phase W variable*/
}stc_uvw_t;
typedef struct
{
    int32_t i32Q8_Xa;   /*Alpha axis variable*/
    int32_t i32Q8_Xb;   /*Beta axis variable*/
}stc_ab_t;
typedef struct
{
    int32_t i32Q8_Xd;   /*D-axis variable*/
    int32_t i32Q8_Xq;   /*Q-axis variable*/
    int32_t i32Q12_Cos; /*Angle sin variable*/
    int32_t i32Q12_Sin; /*Angle cos variable*/
}stc_dq_t;
extern void Clark(stc_uvw_t *pstc_uvw, stc_ab_t *pstc_ab);
extern void InvClark(stc_ab_t *pstc_ab, stc_uvw_t *pstc_uvw);
extern void Park(stc_ab_t *pstc_ab, stc_dq_t *pstc_dq);
extern void InvPark(stc_dq_t *pstc_dq, stc_ab_t *pstc_ab);
```

Code listing 2 shows how to use these functions:

Code Listing 2 Using Clarke and Park transformation functions

```
/* motor_ctrl.c */

MotorCtrl_Process
{
    /* Clarke Transformation uvw -> αβ */
    Clark(&MotorCtrl_stcIuvwSensed, &MotorCtrl_stcIabSensed);

    /* Park Transformation αβ -> dq */
    Park(&MotorCtrl_stcIabSensed, &MotorCtrl_stcIdqSensed);

    /* InvPark Transformation dq-> αβ */
    InvPark(&MotorCtrl_stcVdqRef, &MotorCtrl_stcVabRef);

    /* InvClark Transformation αβ -> uvw */
    InvClark(&_2sC_Ref,&pstcPar->_3sC_Ref);
}
```

4.3 Slide mode observer (SMO)

See **Slide mode observer (SMO)** for an introduction to the SMO theory. The structure and function prototypes for the SMO calculation (**Code Listing 3**) are defined in *smo_calculate.h*.

Code Listing 3 Clarke and Park transformation structures and function prototypes

```
/*smo_calculate.h*/
typedef struct stc_SMO_Estimator
{
```

Code Listing 3 Clarke and Park transformation structures and function prototypes

```
    int32_t i32Q8_Res;     /*the phase resistance*/
    int32_t i32Q8_Lddt;    /*q axis inductance digital factor*/
    int32_t i32Q12_LdLq;   /*dq Axis Mutual Inductance*/
    int32_t i32Q8_IalphaPre; /*stationary alpha-axis stator current*/

    int32_t i32Q8_IbetaPre;  /*stationary beta-axis stator current*/
    int32_t i32Q8_ValphaPre; /*stationary alpha-axis stator voltage*/

    int32_t i32Q8_VbetaPre;  /*stationary beta-axis stator voltage  */
    int32_t i32Q8_ValphaBemf;    /*eistimated alpha Back EMF*/
    int32_t i32Q8_VbetaBemf;     /*eistimated beta Back EMF*/
    int32_t i32Q8_ValphaBemfLpf; /*filtered alpha Back EMF for angle
calculate*/
    int32_t i32Q8_VbetaBemfLpf;  /*filtered beta Back EMF for angle
calculate*/
    stc_one_order_lpf_t ValphaBemLpfK; /*LPF calculate factor*/
    stc_one_order_lpf_t VbetaBemLpfK;  /*LPF calculate factor*/
    int32_t i32Q22_EstimWmHz;   /*estimated rotor speed Q22 format*/
    int32_t i32Q8_EstimWmHz;    /*estimated rotor speed Q8 format*/
    int32_t i32Q8_EstimWmHzf;   /*filtered estimated rotor speed Q8
format*/
    stc_one_order_lpf_t stcWmLpf; /*LPF calculate factor*/
    int32_t i32Q12_Cos;
    int32_t i32Q12_Sin;
    int32_t i32Q12_CosPre;
    int32_t i32Q12_SinPre;
    int32_t i32Q22_Theta;     /*estimated rotor angle*/
    int32_t     i32Q22_ThetaOld; /*estimated rotor angle old*/
    int32_t i32Q22_Dtheta;    /*delta theta of rotor angle for speed
calculate*/
    uint16_t u16_1msCount;    /*counter used to calculate motor speed*/
    int32_t i32Q12_MaxLPFK;   /*BackEMF voltage's max filter
parameter*/
    int32_t i32Q12_MinLPFK;   /*BackEMF voltage's min filter
parameter*/
    int32_t i32Q15_LPFKTS;    /*BackEMF filter's calculation factor*/
    uint16_t u161msTimer; /*1ms timer count*/
    int32_t i32SpdCalKts; /*speed calculate factor*/
    uint8_t u8closeLoopFlg;   /*closed loop flag*/
}stc_SMO_Estimator_t;
extern void Smo_Estimate(stc_SMO_Estimator_t *pstcEstimPar,stc_ab_t
*pstc2sVol, stc_ab_t *pstc2sCurrent);
extern void Smo_Init(stc_SMO_Estimator_t *SMO_Eistimator_t);
```

4.4 PI controllers

The PI regulator keeps the output to follow the expected output by comparing the error between the expected output and the real output. The *P*-value is to make a fast output response to the comparing error, and the *I*-value is to decrease stable output errors. The transfer function can be expressed as shown in **Figure 25**.

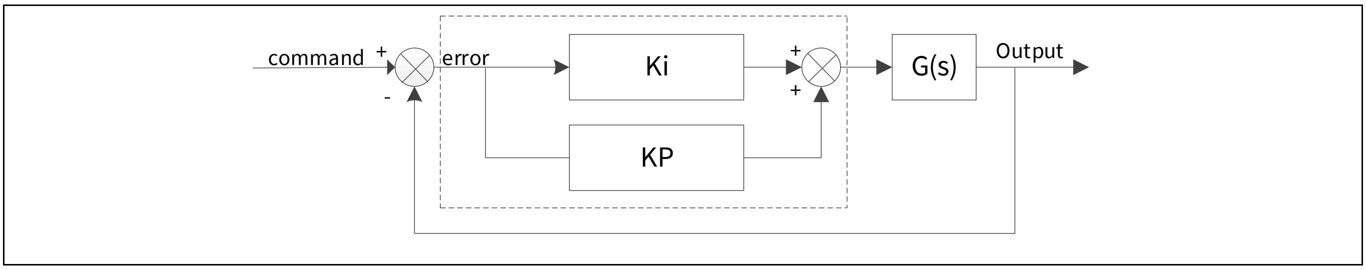


Figure 25 PI-regulator controller

The PI regulator causes a fluctuating output. The fluctuating amplitude decreases, and after the regulating period, the output follows the expected output with a very small fluctuation around the expected output value.

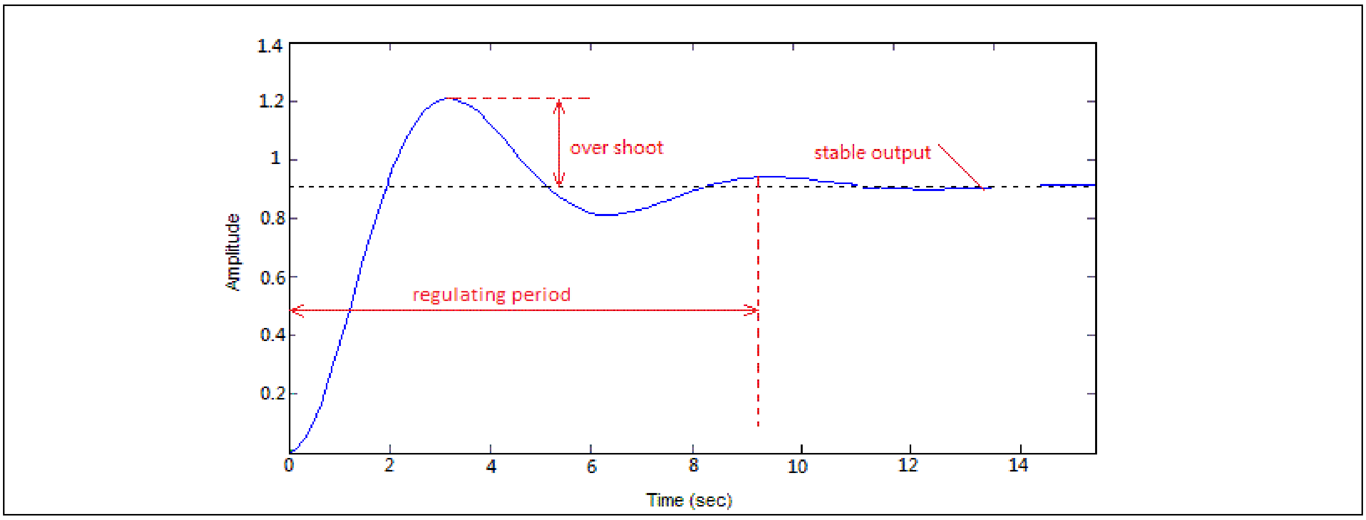


Figure 26 PI regulator output

PI regulator formula:

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau)\, d\tau \qquad \text{Equation 1}$$

Incremental algorithm:

$$\Delta u(k) = k_p[e(k) - e(k-1)] + k_i e(k) \qquad \text{Equation 2}$$

$$u(k) = u(k-1) + \Delta u(k) \qquad \text{Equation 3}$$

Where,

k_p: Proportional factor

k_i: Integration factor

$e(k)$: error between actual and reference

$e(k-1)$: last error

$u(k)$: output value of the PI regulator

$u(k-1)$: last output value of the PI regulator

$\Delta u(k)$: differential value between two output values

PI output limitation:
This is to limit the PI output to a regular range:

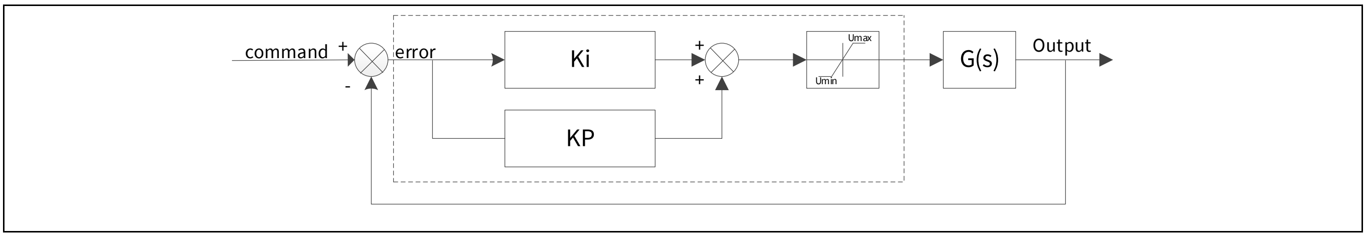


Figure 27 PI regulator with limitation

Three parameters – motor speed, $\boldsymbol{i_q}$, and $\boldsymbol{i_d}$ – are controlled by separate PI controllers. The speed PI controller uses the error between the calculated rotation speed and a given speed reference to calculate the control output, which in turn is the reference for the $\boldsymbol{i_q}$ PI controller. The $\boldsymbol{i_q}$ and $\boldsymbol{i_d}$ PI controllers control $\boldsymbol{u_q}$ and $\boldsymbol{u_d}$, respectively, using the errors for $\boldsymbol{i_q}$ and $\boldsymbol{i_d}$. See **Figure 4**.

4.5 Generating the SVPWM

The SVPWM subsystem produces sinusoidal currents on the motor phases by changing the output duty cycles of the three PWMs (for details, see **SVPWM theory**). The PWM outputs – two complementary outputs for each motor phase – turn the MOSFETs ON or OFF (see **Figure 3**).

Figure 28 shows the SVPWM implementation in PSoC™ 6. A common 75-MHz clock synchronizes the PWM outputs.

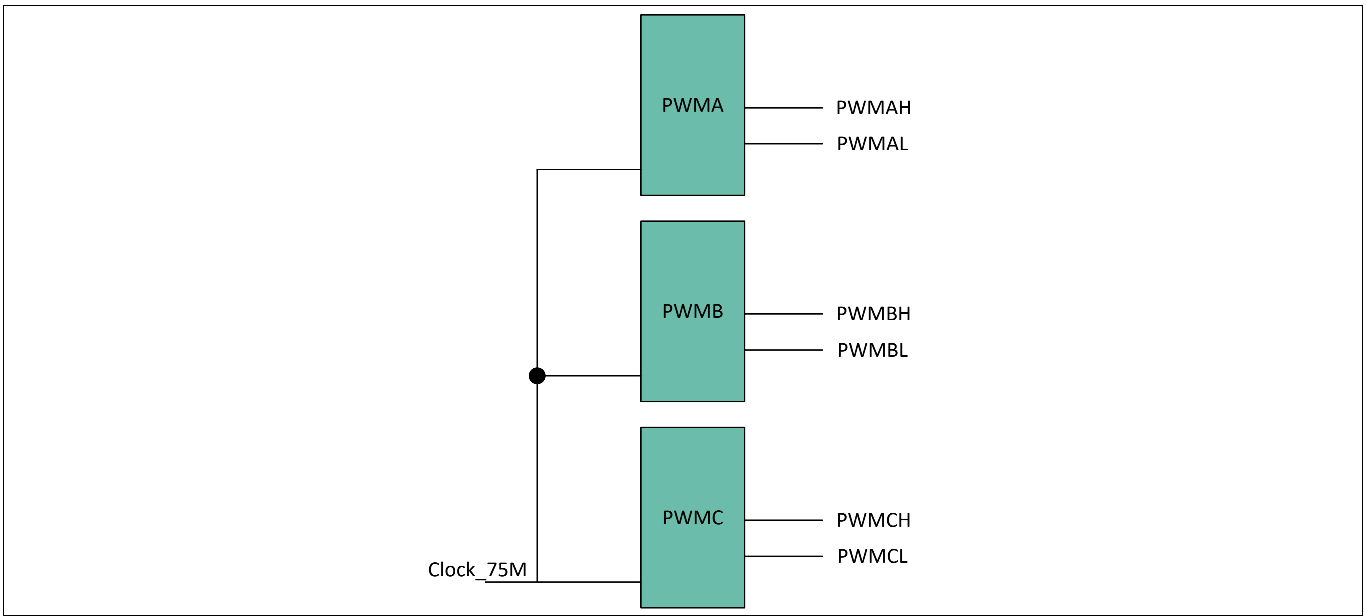


Figure 28 PI regulator with limitation

Figure 29 shows the timing for all three PWMs as well as the details of PWM_A. In addition to the PWM signals, PWM_D generates the trigger signals for the PWM interrupt and the ADC trigger signal as well.

The PWM interrupt is triggered on the terminal count of PWMD. The result is that the ISR controls the PWM duty cycle on every cycle by updating the PWM compare buffer register (**Figure 30**). The register must be updated before the next underflow event occurs, or the duty cycle will be incorrect, which in turn causes an increased motor noise.

Note that each PWM has a different duty cycle.

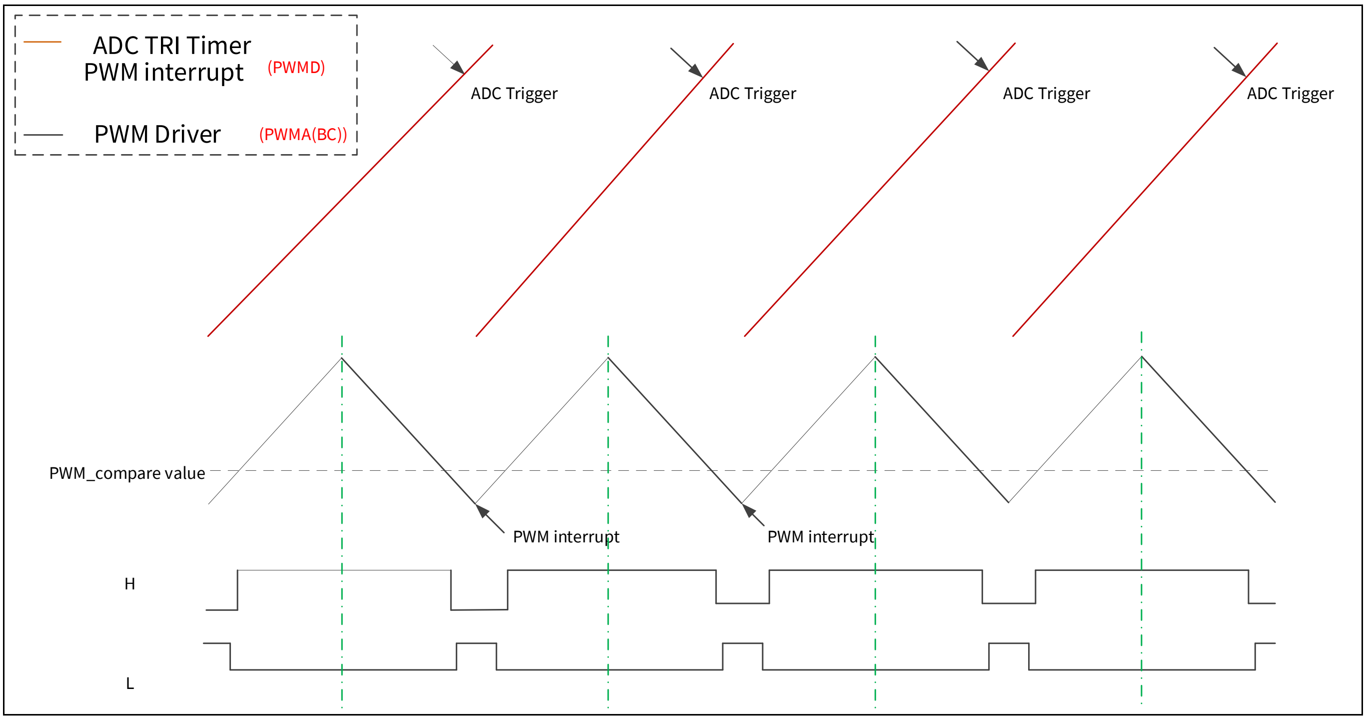


Figure 29 PWM timing

Figure 30 shows the configuration for PMW_A (phase A); it applies to all three PWM components:

- The alignment mode is “Center align”. This produces the complementary PWM outputs ‘line’ and ‘line_n’. The outputs turn the MOSFETs of one of the motor phases ON and OFF (such as Q1 and Q2 in **Figure 3**).
- A deadband time is inserted to avoid turning ON both MOSFETs at the same time, which can damage the MOSFETs. In this code example, 41 cycles of a 75-MHz clock results in a dead time of 0.55 µs. Deadtime can also been changed in motor control firmware; set it to 1.0 µs.
- The period value is the clock frequency divided by twice the desired PWM frequency. Here, the desired PWM frequency is doubled because the count mode is up-down (see **Figure 30**). For a 75-MHz clock and a desired PWM frequency of 10 kHz, the period is (75,000,000 / (2 * 10,000)), or 3750.

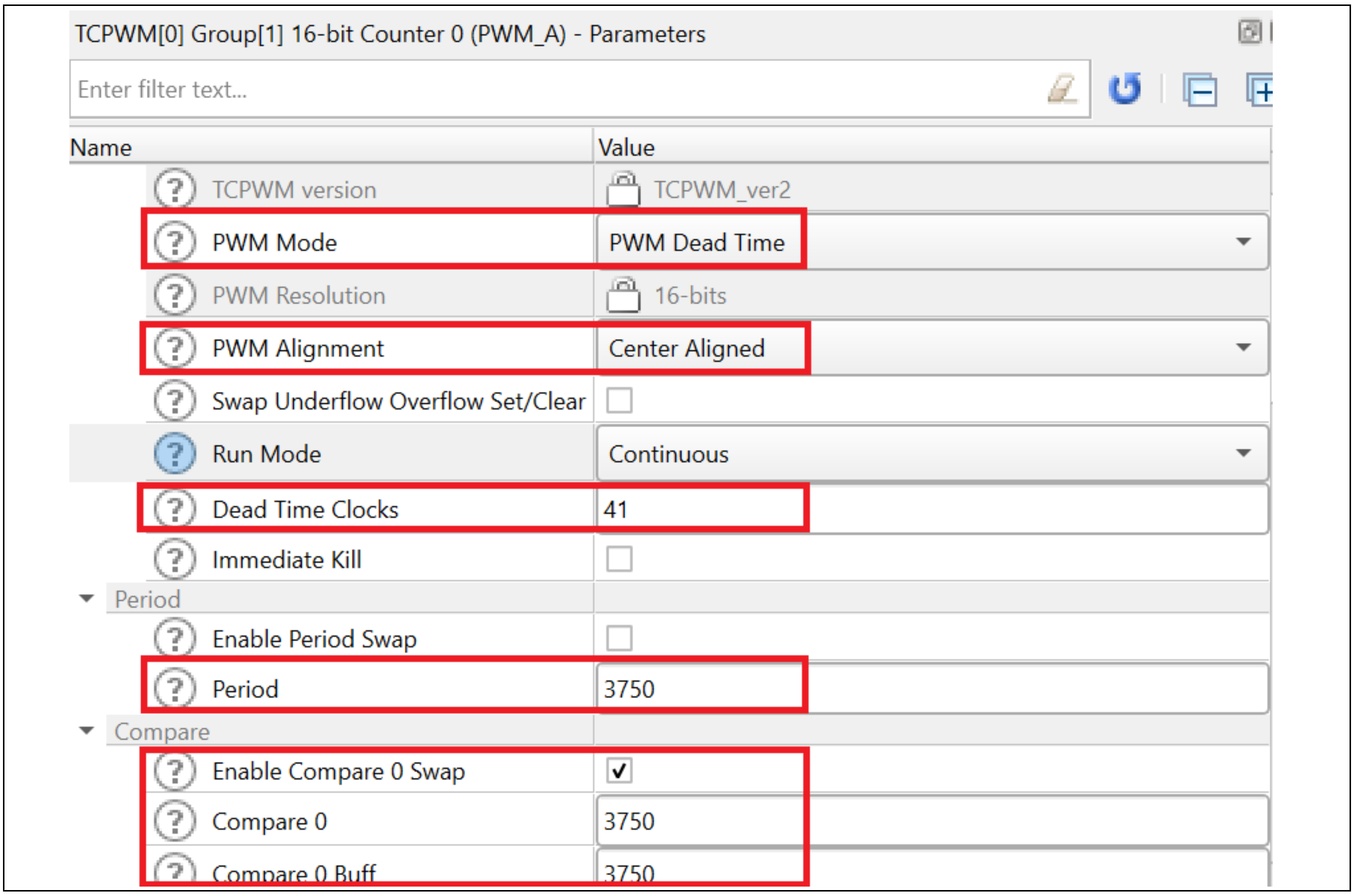


Figure 30 PWMA configuration

5 Appendix A: PMSM model

This section presents the mathematical model of a permanent magnet synchronous motor (PMSM). To simplify the model, some assumptions are made:

- The PMSM motor winding connection is the "star" type. "Delta" type connections must be converted to the "star" type.
- Magnetic saturation is neglected.
- Eddy currents and hysteresis losses are negligible.

Figure 31 illustrates the PMSM motor model in a 3-phase stator reference frame, also called the *(a, b, c)* frame. In this frame, the a, b, and c axes are aligned with the currents i_a, i_b, i_c in the three phases of the PMSM stator, and are 120° apart from each other. Ψ_f is the flux linkage vector of the rotor magnet. The rotor rotates with an angular speed ω_r, and θ_r is the angle between Ψ_f and phase a.

The a, b, and c phases are each called "line". The connection point of a, b, and c is called the neutral point.

The voltages on the stator windings are represented as:

$$\begin{cases} u_a = R_a \times i_a + \dfrac{d\Psi_a}{dt} \\ u_b = R_b \times i_b + \dfrac{d\Psi_b}{dt} \\ u_c = R_c \times i_c + \dfrac{d\Psi_c}{dt} \end{cases}$$

Where:

u_a, u_b, u_c	Stator voltage vector
R_a, R_b, R_c	Stator resistance
i_a, i_b, i_c	Stator current vector
Ψ_a, Ψ_b, Ψ_c	Stator flux linkages

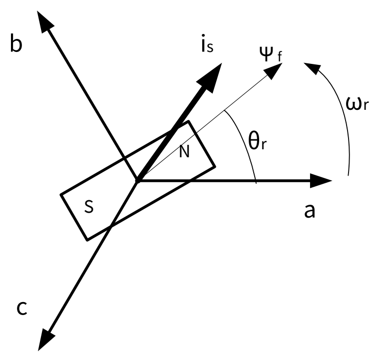


Figure 31 3-phase stator reference frame

The stator winding flux linkage is the sum of the flux linkages from their own excitation, mutual flux linkages from other winding currents, and flux linkages from the rotor magnet. Because the current phases on the stator windings are 120° apart, the stator flux linkages are written as:

$$\begin{cases} \Psi_a = L_{aa}(\theta_r) \times i_a + M_{ab}(\theta_r) \times i_b + M_{ac}(\theta_r) \times i_c + \Psi_f \times \cos\theta_r \\ \Psi_b = M_{ba}(\theta_r) \times i_a + L_{bb}(\theta_r) \times i_b + M_{bc}(\theta_r) \times i_c + \Psi_f \times \cos(\theta_r - 120°) \\ \Psi_c = M_{ca}(\theta_r) \times i_a + M_{cb}(\theta_r) \times i_b + L_{cc}(\theta_r) \times i_c + \Psi_f \times \cos(\theta_r + 120°) \end{cases}$$

Where:

L_{aa}, L_{bb}, L_{cc} Equivalent inductances of stator phases

M_{ab}, M_{ac}, M_{ba}, M_{bc}, M_{ca}, M_{cb} Mutual equivalent inductances of stator phases

Ψ_f Amplitude of rotor flux linkage

θ_r Angle between Ψ_f and phase a

This model is of a high order, is strongly coupled, and has nonlinearity; analyzing it and controlling the torque and flux based on it is difficult. Therefore, the (d, q) frame is used to simplify the 3-phase model. The (d, q) frame defines a rotating 2-phase reference frame where the d axis is aligned with the rotor flux direction.

There are two transformations to convert the (a, b, c) frame to the (d, q) frame. The first one is a Clarke transformation – it converts the (a, b, c) frame to a 2-phase stationary reference frame (α, β) (**Figure 32**).

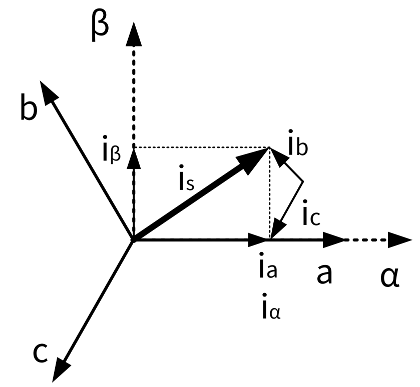


Figure 32 Clarke transformation

The current vectors in the (α, β) frame are:

$$\begin{cases} i_\alpha = \frac{2}{3} \times i_a - \frac{1}{3} \times i_b - \frac{1}{3} \times i_c \\ i_\beta = \frac{\sqrt{3}}{3} \times i_b - \frac{\sqrt{3}}{3} \times i_c \end{cases}$$

For “star” type winding connections, the sum of the currents in the three phases is zero:

$$i_a + i_b + i_c = 0$$

Therefore, the current vectors in the (a, b, c) frame are transformed to the (α, β) frame as:

$$\begin{cases} i_\alpha = i_a \\ i_\beta = \frac{\sqrt{3}}{3} \times i_a + \frac{2\sqrt{3}}{3} \times i_b \end{cases}$$

The Park transformation then converts the (α, β) frame to the (d, q) frame. The (d, q) frame has two axes – direct and quadrature – that rotate with the same angle speed ω_r as the current vector. The direct axis is aligned with the rotor flux Ψ_f (**Figure 33**). The angle between the d axis and the α axis is θ_r.

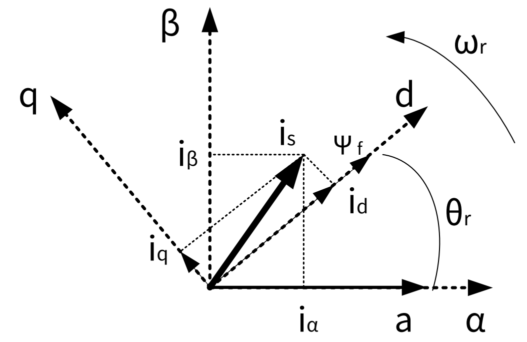


Figure 33 Park transformation

The current vectors in the (d, q) frame are calculated as:

$$\begin{cases} i_d = i_\beta \times \sin\theta_r + i_\alpha \times \cos\theta_r \\ i_q = i_\beta \times \cos\theta_r - i_\alpha \times \sin\theta_r \end{cases}$$

The voltages in the (d, q) frame are calculated from i_d and i_q, as:

$$\begin{cases} u_d = R \times i_d + \dfrac{d\Psi_d}{dt} - \omega_r \times \Psi_q \\ u_q = R \times i_q + \dfrac{d\Psi_q}{dt} + \omega_r \times \Psi_d \end{cases}$$

and:

$$\begin{cases} \Psi_d = L_d \times i_d + \Psi_f \\ \Psi_q = L_q \times i_q \end{cases}$$

The torque equation is expressed as:

$$T_e = \frac{3}{2} P_n [\Psi_f i_q - (L_q - L_d) i_d i_q] - T_L$$

Where:

L_d , L_q Inductances of direct and quadrature axes

P_n Number of pole pairs in rotor

Note that for a SPMSM (**Figure 1**), L_q and L_d are independent of θ_r, and L_q is equal to L_d. Thus, the torque equation is simplified for SPMSM as:

$$T_e = \frac{3}{2} P_n \Psi_f i_q - T_L$$

$\mathbf{P_n}$ and $\boldsymbol{\Psi_f}$ are motor characteristics that are not affected by the motor rotation. Compared to the 3-phase model, the torque is proportional only to the q-axis current $\boldsymbol{i_q}$, which is easier to control.

5.1 Slide mode observer (SMO)

Obtaining the position of a rotating rotor is critical for FOC. The Park transformation requires the rotor position angle $\boldsymbol{\theta_r}$ between the rotor flux linkage $\boldsymbol{\Psi_f}$ and the α axis. Originally, this information came from physical sensors, such as Hall-effect sensors and optical encoders. These sensors not only increase the system cost, but they also require maintenance. Later, the sensorless technique was developed to remove the need for sensors. Some high-precision applications such as robotics still require encoders.

The idea of the sensorless technique is to estimate the angle $\boldsymbol{\theta_r}$ based on the BEMF value in the (α, β) frame. The typical algorithm to do this is called a slide mode observer (SMO). In this algorithm, the 2-phase voltages in the (α, β) frame is expressed as:

$$\begin{cases} u_\alpha = R_s \times i_\alpha + L_s \times \dfrac{di_\alpha}{dt} + e_\alpha \\ u_\beta = R_s \times i_\beta + L_s \times \dfrac{di_\beta}{dt} + e_\beta \end{cases}$$

Where:

- $\boldsymbol{R_s}$ Line-to-neutral resistance
- $\boldsymbol{L_s}$ Line-to-neutral inductance
- $\boldsymbol{e_\alpha}$, $\boldsymbol{e_\beta}$ BEMF on (α, β) axes

In the digital domain, the $\boldsymbol{u_\alpha}$ equation is changed to:

$$\frac{i_\alpha(n+1) - i_\alpha(n)}{T_s} = \left(-\frac{R_s}{L_s}\right) i_\alpha(n) + \frac{1}{L_s}[u_\alpha(n) - e_\alpha(n)]$$

Where:

- $\boldsymbol{T_s}$ Period of PWM on inverter

Solving for $\boldsymbol{i_\alpha}$:

$$i_\alpha(n+1) = (1 - T_s \tfrac{R_s}{L_s}) i_\alpha(n) + \tfrac{T_s}{L_s}[u_\alpha(n) - e_\alpha(n)]$$

You can now define two new parameters that are related to motor parameters:

$$F = 1 - T_s \frac{R_s}{L_s}$$

$$G = \frac{T_s}{L_s}$$

Note that $\boldsymbol{R_s}$ and $\boldsymbol{L_s}$ are motor characteristics that can be measured. $\boldsymbol{T_s}$ is a constant system parameter, $\boldsymbol{i_\alpha(n)}$ is the sampled result from the last control cycle, and $\boldsymbol{u_\alpha(n)}$ is the calculation result of the last control cycle. Therefore, if given an estimated $\boldsymbol{e_\alpha^*(n)}$, an estimated current value $\boldsymbol{i_\alpha^*(n+1)}$ can be calculated ("*" indicates an estimated value).

Comparing $\boldsymbol{i_\alpha^*(n+1)}$ with the actual current value $\boldsymbol{i_\alpha(n+1)}$ sampled by the ADC, the error between these two values is used to adjust $\boldsymbol{e_\alpha^*(n)}$ for a better estimation. Repeat this process until the error between

$i_\alpha^*(n+1)$ and $i_\alpha(n+1)$ is small enough to meet the design requirements. Then, the estimated $e_\alpha^*(n)$ can represent the actual BEMF $e_\alpha(n)$. The $e_\beta(n)$ is obtained in the same manner.

Because $e_\alpha(n)$ and $e_\beta(n)$ are expressed as:

$$\begin{cases} e_\alpha(n) = -\Psi_f \times \omega \times \sin\theta \\ e_\beta(n) = \Psi_f \times \omega \times \cos\theta \end{cases}$$

The angle θ is calculated as:

$$\theta(n) = arc\,tan\frac{-e_\alpha(n)}{e_\beta(n)}$$

The angular speed ω_r is calculated by accumulating θ over m samples and multiplied by the speed constant K:

$$\omega_r = \sum_{n=1}^{m}[\theta(n) - \theta(n-1)] * K$$

Thus, the position and speed information are calculated from the estimated BEMF.

5.2 SVPWM theory

In **Figure 3**, Q1, Q3, and Q5 are the upper MOSFETs of the inverter. If you consider the MOSFET ON state as "1" and the OFF state as "0", there are eight combinations of ON/OFF states, which lead to eight inverter outputs.

Table 4 lists the ON/OFF state combinations and their corresponding inverter outputs. u_a, u_b, and u_c are the phase (line-to-neutral) voltages, while u_{ab}, u_{bc}, and u_{ac} are the line-to-line voltages. The values in each cell indicate the voltage as a percentage of the bus voltage, V_{bus}. For example, 2/3 means 2/3 of V_{bus}.

Table 4 Output combination in 3-phase frame

Q1 (A)	Q3 (B)	Q5 (C)	u_a	u_b	u_c	u_{ab}	u_{bc}	u_{ca}	
1	0	0	2/3	-1/3	-1/3	1	0	-1	U_0
1	1	0	1/3	1/3	-2/3	0	1	-1	U_{60}
0	1	0	-1/3	2/3	-1/3	-1	1	0	U_{120}
0	1	1	-2/3	1/3	1/3	-1	0	1	U_{180}
0	0	1	-1/3	-1/3	2/3	0	-1	1	U_{240}
1	0	1	1/3	-2/3	1/3	1	-1	0	U_{300}
0	0	0	0	0	0	0	0	0	0_{000}
1	1	1	0	0	0	0	0	0	0_{111}

The eight combinations can be considered as six non-zero vectors and two zero vectors (000 and 111). As **Figure 34** shows, the non-zero vectors are the axes of a hexagon; the angle between any two adjacent axes is 60 degrees. This divides the hexagon into six sectors (Roman numerals I to VI). The zero vectors are at the origin, and they generate zero voltage on the three phases. These eight vectors, called "basic space vectors," are called U_0, U_{60}, U_{120}, U_{180}, U_{240}, U_{300}, 0_{000}, and 0_{111}. A voltage vector is synthesized by one or two of the six non-zero basic space vectors.

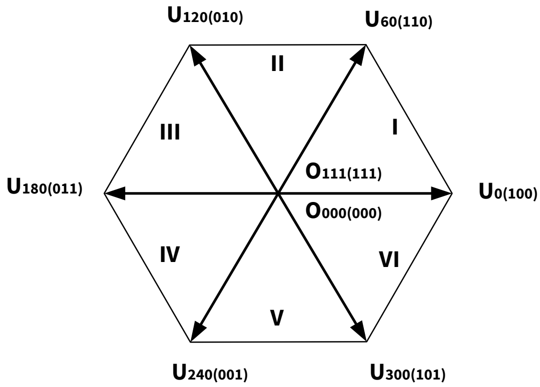


Figure 34 Basic space vectors

For example, as **Figure 35** shows, the voltage vector $\overrightarrow{U_s}$ is in sector I, and the period of the PWM is T. T_1 is the duration of U_0; T_2 is the duration of U_{60}; and T_0 is the duration of the two zero vectors. The vectors $\overrightarrow{u_\alpha}$ and $\overrightarrow{u_\beta}$ compose a voltage vector, $\overrightarrow{U_s}$, that can also be composed by basic space vectors U_0 and U_{60}.

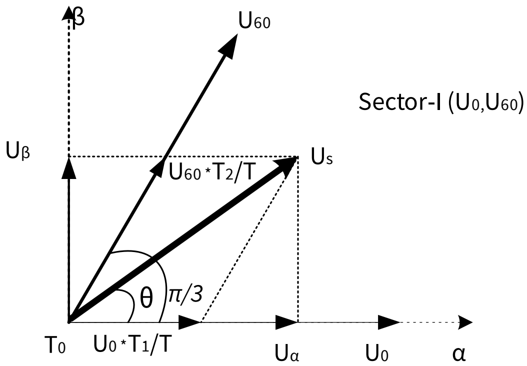


Figure 35 Voltage vector in Sector I

$\overrightarrow{U_s}$ can be expressed as:

$$T = T_1 + T_2 + T_0$$

$$\overrightarrow{U_s} = \overrightarrow{U_{60}} \times \frac{T_2}{T} + \overrightarrow{U_0} \times \frac{T_1}{T}$$

Therefore:

$$|U_s|\cos\theta = |U_{60}| \times \frac{T_2}{T} \times \cos\frac{\pi}{3} + |U_0| \times \frac{T_1}{T}$$

$$|U_s|\sin\theta = |U_{60}| \times \frac{T_2}{T} \times \sin\frac{\pi}{3}$$

Then:

$$T_1 = mT\sin\left(\frac{\pi}{3} - \theta\right)$$

$$T_2 = mT\sin\theta$$

$$\boldsymbol{T_0} = \boldsymbol{T} - \boldsymbol{T_1} - \boldsymbol{T_2}\ (\boldsymbol{T_0} \geq \mathbf{0})$$

Where:

$$\boldsymbol{m} = \sqrt{\mathbf{3}} \times \frac{|\boldsymbol{U_{out}}|}{\boldsymbol{U_{dc}}}$$

$$|\boldsymbol{U_{out}}| = \sqrt{|\boldsymbol{u_\alpha}|^2 + |\boldsymbol{u_\beta}|^2}$$

Note that all basic space vectors are generated with a specific ON/OFF state of upper MOSFETs; the duration is actually the time of the PWM being high, or the duty cycle. Thus, generating a $\overrightarrow{\boldsymbol{U_s}}$ is related to a change in duty cycle of the PWMs applied to the inverter. In this example, both $\boldsymbol{U_0}$ and $\boldsymbol{U_{60}}$ require phase A to be turned ON, and $\boldsymbol{U_{60}}$ requires phase B to be turned ON. Therefore:

$$\boldsymbol{Duty_A} = \frac{\boldsymbol{T_1} + \boldsymbol{T_2}}{\boldsymbol{T}}, \qquad \boldsymbol{T_1} + \boldsymbol{T_2} \leq \boldsymbol{T}$$

$$\boldsymbol{Duty_B} = \frac{\boldsymbol{T_2}}{\boldsymbol{T}}, \qquad \boldsymbol{T_2} \leq \boldsymbol{T}$$

$$\boldsymbol{Duty_0} = \frac{\boldsymbol{T} - \boldsymbol{T_1} - \boldsymbol{T_2}}{\boldsymbol{T}}$$

Depending on how you use zero vectors, the SVPWM has two output patterns: a five-phase pattern and a seven-phase pattern. The five-phase pattern uses only $\mathbf{0_{000}}$ or $\mathbf{0_{111}}$.The seven-phase pattern uses both $\mathbf{0_{000}}$ and $\mathbf{0_{111}}$, and their durations are equal. **Figure 37** illustrates these two patterns. Note that in 5-phase SVPWM, phase A is always on or always off.

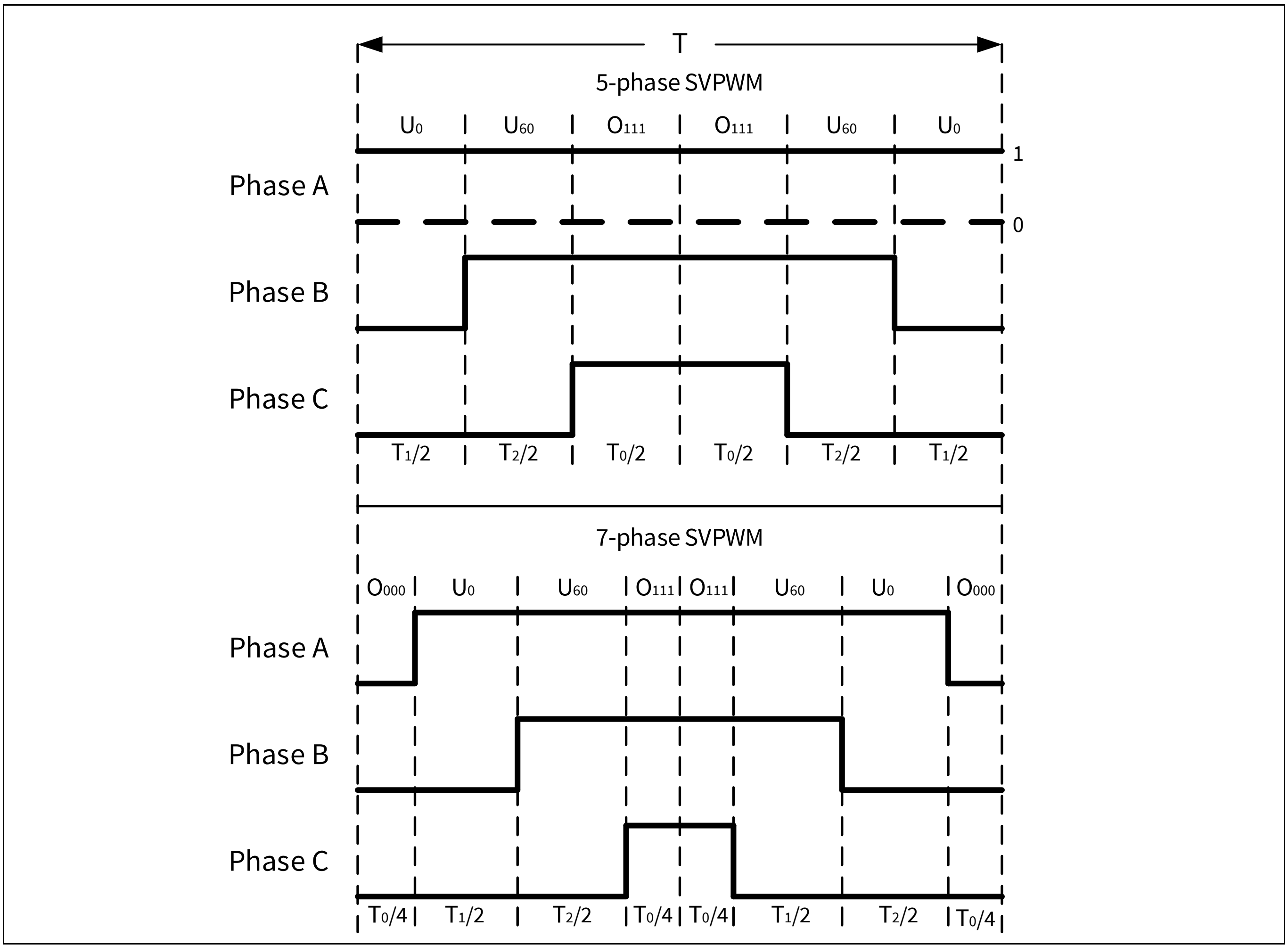


Figure 36 5- and 7-phase SVPWM in Sector I

There is no difference in the synthesized voltage vector generated by these two methods. However, the 5-phase pattern reduces the number of MOSFETs that are switching. This can reduce the switching losses in the power components, but it creates more harmonics than the seven-phase pattern.

6 Appendix B: Adapting the design to other motors

This appendix helps you to drive other motors with the code example provided with this application note. You should follow the operation guide step by step. A **bold** font indicates a **mandatory** action or **critical** information that requires more attention.

Hardware: CY8CKIT-037 or your own motor driver board

Firmware: Sensorless FOC project from the latest version of this application note

Equipment: Oscilloscope, multimeter, PC, USB cable for CY8CKIT-062S4 or J-Link for programming your own board.

Operation guide:

1. **Check the power range and motor type.**

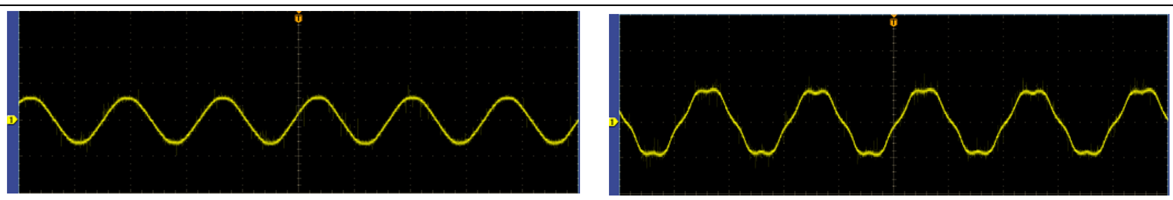
   a. **Power range**

   CY8CKIT-037 supports a 12-V to 48-VDC supply voltage with up to 2 A input DC current. You should use the kit only in this power range; using the kit out of this power range may damage it.

   b. **Motor type**

   A motor with **sinusoidal back electromotive force (BEMF)** is recommended. A motor with trapezoidal BEMF may not rotate or achieve the desired performance with the sensorless FOC project. **Figure 37** illustrates these two BEMF types. To measure BEMF, connect the ground of the oscilloscope probe to one motor phase and the probe to another motor phase. Leave the other motor phases floating. Rotate the motor either by hand or by using another motor. You should see the BEMF waveform on the oscilloscope.

   The sinusoidal BEMF contains the complete angle information, which can be calculated with the SMO algorithm. The trapezoidal BEMF is almost flat at the wave crest and trough and therefore is missing sufficient angle information. As a result, the SMO algorithm cannot reliably retrieve the angle from this waveform, which may halt the motor rotation.



(a) Sinusoidal BEMF (b) Trapezoidal BEMF

Figure 37 Sinusoidal BEMF versus trapezoidal BEMF

2. **Change the parameters in the example project.**

   a. **These parameters are defined as global variables in *h03_user\customer_interface.c*. You should change them based on your motor specifications.**

```
int32_t   i32_motor_pole_pairs = 4;      // the pole pairs of rotor
float32_t f32_motor_ld         = 0.6;  // the d axis reductance,unit:mh
float32_t f32_motor_lq         = 0.6;  // the q axis reductance,unit:mh
float32_t f32_motor_res    = 0.8;    // the resistance between two phases
```

   b. **Change the macro definitions for the system parameters.**

These macro definitions are related to system parameters, such as the sampling resistor and so on. You should change them (*h03_user\hardware_config.h*) if the default values are different from your system.

```
#define SYS_VDC_FACTOR   20.1 //DC voltage sample resistor factor
#define MOTOR_SHUNT_NUM   2 // The number of shunt used to sense current
#define MOTOR_IUVW_SAMPLE_RESISTOR  0.03 // Iuvw sample resistor (ohm)
#define MOTOR_IUVW_AMPLIFIER_FACTOR   4.16 // Iuvw calculation factor
#define ADC_VOLT_REF           5.0f     // Reference voltage for ADC
#define ADC_VALUE_MAX          4096.0f  // 12-bits ADC max value
```

c. **Change the parameters for the PI controllers.**

You may need to change the PI coefficient parameters in the PI controller if the PI controller does not work well with your motor. You can change the parameters in *h03_user\customer_interface.c.* For more details, see **Tunable parameters**.

3. **Set up the hardware.**

If you are using the CY8CKIT-037 kit, you can use the adapter provided with the kit for any motor whose maximum power is 24 V DC / 2.1 A. If a different voltage (such as 48 V) or current (such as 3 A) is required, connect the DC voltage source to the J8 connector (yellow marker in **Figure 38**) instead of the supplied power adapter.

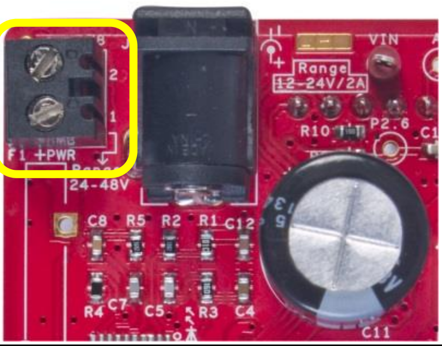


Figure 38 Setting up the board for a motor with a higher voltage or current

4. **Program the CY8CKIT-062S4 kit and observe the performance.**
5. **Tune the parameters if the motor does not rotate**

a. The motor starts up in open-loop control and then switches to closed-loop speed control later. If switching to the closed loop control fails (motor halts very soon after the rotation starts), you may need to tune the following parameters. Try the following methods:

- Confirm that the motor parameters are set correctly in **Step 2**.
- Change the parameters switch from open-loop to closed-loop in *h03_user\customer_interface.c*.

```
uint16_t  u16_motor_open_loop_spd_init_hz    = 5; //open loop start speed
uint16_t  u16_motor_open_loop_spd_end_hz     = 10;//open loop end speed
uint16_t  u16_motor_open_loop_spd_inc_hz     = 10;//acceleration speed of open loop
uint16_t  u16_motor_close_loop_target_spdhz  =  10;  //target  speed  when switching to close loop
```

- Debug with the Back-EMF low-pass filter factors in SMO structure *Motor_stcSMO*.
- If the error occurs when the motor is running, the motor will stop immediately. Check the variable *MotorCtrl_stcRunPar.u32ErroType* to find the error. If the error is over/under voltage, confirm the parameters set in **2.b**. You can clear the error by rotating the potentiometer to the smallest value. If the error occurs more than 10 times, it cannot be cleared, and you should reset the board.
- When the motor is running, LED2 will blink according to the motor's speed. If motor's speed goes high, the LED2 will blink more frequently.

b. If the motor rotates with a vibration, try tuning the Kp and Ki parameters in the PI controller. The larger the Kp value, the faster the system closes in on the reference value; however, it may make the system unstable. The Ki value can reduce the static error and make the system stable; however, a larger Ki may make the integration value saturate.

6.1 Tunable parameters

6.1.1 Hardware parameter setting

The hardware parameters should be set according to the kit. If you have your own inverter board, change the parameters mentioned in **Table 5** in the *h03_user\hardware_config.h* file.

Table 5 Hardware parameter setting

Macro	Description	Value
`SYS_VDC_FACTOR`	DC voltage sample resistor factor	20.1
`MOTOR_SHUNT_NUM`	Number of shunts used to sense current	2
`ADC_VOLT_REF`	AD reference voltage	3.3 V
`ADC_VALUE_MAX`	AD accuracy set, 12-bit AD is set to '0xFFF'	4096
`COMP_ADC_CH_IU`	ADC channel for U phase current	0
`COMP_ADC_CH_IW`	ADC channel for W phase current	1
`SYS_ADC_CH_VDC`	ADC channel for VBUS	2
`MOTOR_SPEED_VR`	ADC channel for potentiometer voltage	3
`MOTOR_IUVW_SAMPLE_RESISTOR`	Iuvw sample resistor	0.03 Ω
`MOTOR_IUVW_AMPLIFIER_FACTOR`	Iuvw calculation factor	4.16

Especially, in the **Table 5**,

- `SYS_VDC_FACTOR`: The factor for calculating Vbus, which is determined by the input protection circuit in the following diagram. Here, `SYS_VDC_FACTOR` = (R9 + R10) / R10.

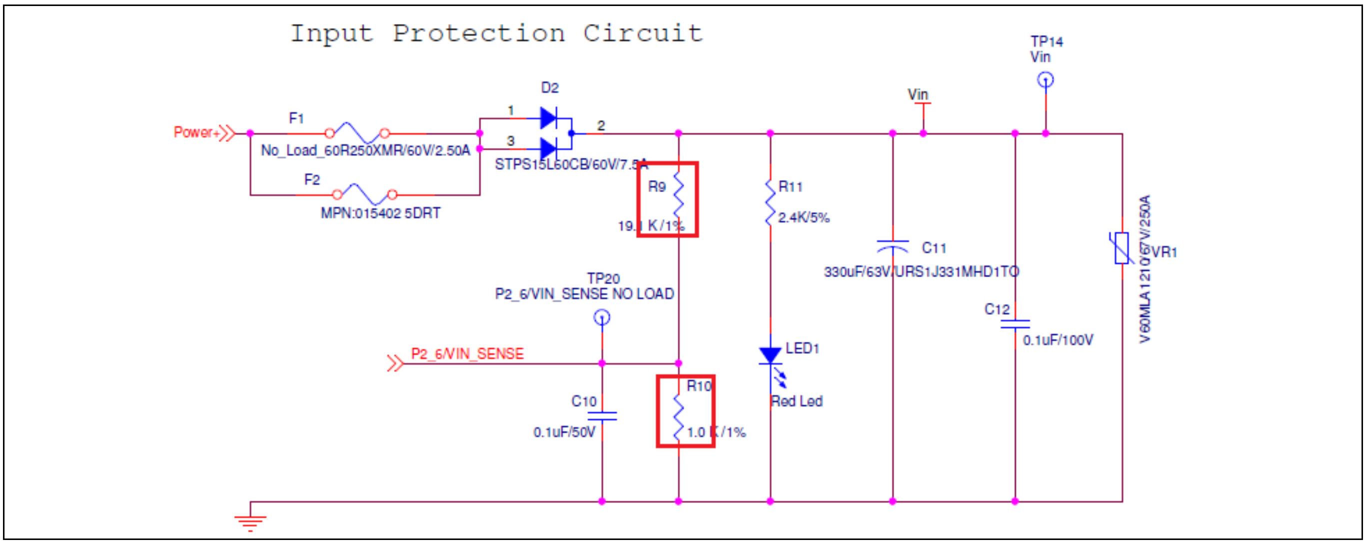


Figure 39 Input protection circuit

- `MOTOR_SHUNT_NUM`: Number of shunts used to sense current, which is dependent on your circuit of motor current detection.
- `ADC_VOLT_REF`: ADC sampling reference voltage of the system.
- `ADC_VALUE_MAX`: Depends on the accuracy of the ADC; the accuracy of the internal ADC is 12-bits, thus the maximum ADC value is 4096. You need to change the value according to your own schematic.
- `COMP_ADC_CH_IU`: ADC channel number for motor U phase current sense, that is channel 0.
- `COMP_ADC_CH_IW`: ADC channel number for motor W phase current sense, that is channel 1.
- `SYS_ADC_CH_VDC`: ADC channel number for bus voltage sense, that is channel 2.
- `MOTOR_SPEED_VR`: ADC channel number for potentiometer input sense, that is channel 3.

  The four parameters (`COMP_ADC_CH_IU`, `COMP_ADC_CH_IW`, `SYS_ADC_CH_VDC`, `MOTOR_SPEED_VR`) are set by the *design.modus* file, and the motor phase current sense depends on the circuit for current detection. In CY8CKIT-037, the circuit detects the current of U and W phase. If the order of ADC channels in **Figure 40** is changed, for example, if OP_Ia_Vout_Filt and VR-In are interchanged, the COMP_ADC_CH_IU parameter should set to 3, and MOTOR_SPEED_VR should set to 0.

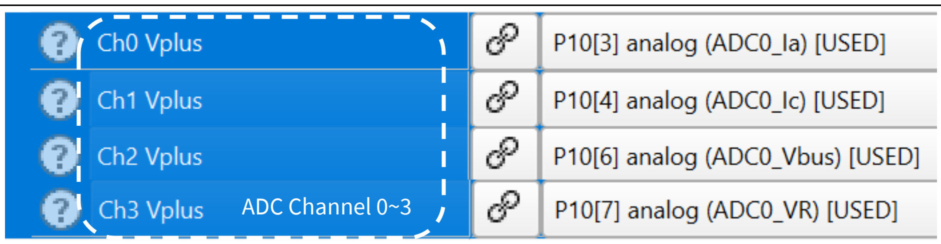


Figure 40 ADC channel number set

- `MOTOR_IUVW_SAMPLE_RESISTOR`: Value of the sample resistor in the current detection circuit.
- `MOTOR_IUVW_AMPLIFIER_FACTOR`: Amplification factor in the amplification circuit.

6.1.2 Firmware parameter setting

The firmware parameters are defined for motor running. The firmware parameters include motor parameters, motor carry frequency, PI parameters, and motor start-up parameters, which are in the *s03_user\customer_interface.c* file.

6.1.2.1 Motor parameters

Motor parameters include the motor pole pairs, phase current, and phase inductance. **Table 6** lists the details of these parameters.

Table 6 Motor parameters

Variable	Description
`i32_motor_pole_pairs`	Motor's pole pairs
`f32_motor_ld`	Phase inductance of d axis. Unit: mH.
`f32_motor_lq`	Phase inductance of q axis. Unit: mH.
`f32_motor_res`	Resistance between two phases. Unit: Ω.

Motor parameters are dependent on the motor that you choose.

The motor pole pair is usually labeled in the motor nameplate. The phase inductance of d/q axis and the phase resistor can be detected by the RLC measuring instrument.

6.1.2.2 ADC sampling parameters

These parameters are defined for ADC sampling. The value of the sample resistor is related to the circuit. **Table 7** lists the details of these parameters.

Table 7 ADC sampling parameters

Variable	Description
`i32_motor_iuvw_offset_normal`	Middle value of 12-bits ADC: 4096/2=2048
`i32_motor_iuvw_offset_range`	ADC offset range of Iuvw sampling. If the error of the ADC checked value is out of this range, the system will raise the `AD_MIDDLE_ERROR` fault.
`i32_motor_iuvw_offset_check_times`	Iuvw ADC sample offset check times
`f32_motor_dead_time_micro_sec`	Dead time (µs) of the PWM
`u16_motor_carrier_freq`	Motor carry frequency (Hz)

- `i32_motor_iuvw_offset_normal`: The middle value of 12-bits ADC. For example: if your system has 3.3 VDDA, the maximum ADC input is 3.3 V and the normal offset value is 1.65 V. Thus, the ADC normal offset output is 2048.
- `i32_motor_iuvw_offset_range`: The range of current offset check. If the offset check result is out of this range, the system will raise the `AD_MIDDLE_ERROR` fault. Do not set a higher value for this parameter because the motor current will fluctuate a lot if there is something wrong with the current detection circuit. This parameter can be set to a value of 150~200.
- `i32_motor_iuvw_offset_check_times`: Iuvw ADC sample offset check times. The offset check result is an average value of the sum of those check values. You can set this value based on your requirement. However, the value should not exceed 256.

- `f32_motor_dead_time_micro_sec`: The dead time of the PWM expressed in µs. This parameter is set according to the inverter circuit or the IPM blocks that you use.
- `u16_motor_carrier_freq`: This parameter should be set based on the MCU and the FOC execute time. You should set it according to your own MCU and load (motor). However, if this parameter is set to a higher value, the inverter service life will be reduced.

6.1.2.3 PI regulator parameters

Table 8 PI regulator parameters

Variable	Description
f32_motor_dki	d axis current PI regulator integral constant
f32_motor_dkp	d axis current PI regulator proportion constant
f32_motor_qki	q axis current PI regulator integral constant
f32_motor_qkp	q axis current PI regulator proportion constant
f32_motor_low_speed_ki	Speed PI regulator integral constant at low speed
f32_motor_low_speed_kp	Speed PI regulator proportion constant at low speed
f32_motor_ski	Speed PI regulator integral constant at high speed
f32_motor_skp	Speed PI regulator proportion constant at high speed
u16_motor_change_pi_spdhz	PI parameters change at this speed

These parameters are set for the current and speed PI loop. You should change the values according to your own motor and prior experience.

6.1.2.4 Startup parameters

Table 9 Motor startup parameters

Variable	Description
u8_motor_run_level	Motor run stage: 1 → orientation, 2 → open-loop running, 3 → closed-loop running, 4 → change speed enable
i16q8_motor_orient_end_iqref	Orientation current when motor in orient stage. Unit: A.
i16q8_motor_orient_init_iqref	Orientation start current. Unit: A.
f32q8_motor_orient_iqref_inc_aps	Reference vary step in orient stage
f32q8_motor_orient_time	Orientation time. Unit: s.
u16_motor_open_loop_spd_init_hz	Open-loop start speed. Unit: Hz.
u16_motor_open_loop_spd_end_hz	Open-loop end speed; this value should be the same as the speed when the motor changes to closed-loop. Unit: Hz.
u16_motor_open_loop_spd_inc_hz	Open-loop acceleration. Unit: Hz.
i16q8_motor_open_loop_init_iqref	q axis current reference in open loop. Unit: A.
i16q8_motor_open_loop_end_iqref	q axis current reference in open loop. Unit: A.
f32_motor_open_loop_iqref_inc_aps	q axis current reference vary step in open loop

- `u8_motor_run_level`: This parameter is used to set the motor running stage. Set this parameter to 4 if you need to change the speed while the motor is running.
- `i16q8_motor_orient_end_iqref`, `i16q8_motor_orient_init_iqref`, `f32q8_motor_orient_iqref_inc_aps`, and `f32q8_motor_orient_time`: These parameters are explained in **Figure 42**.

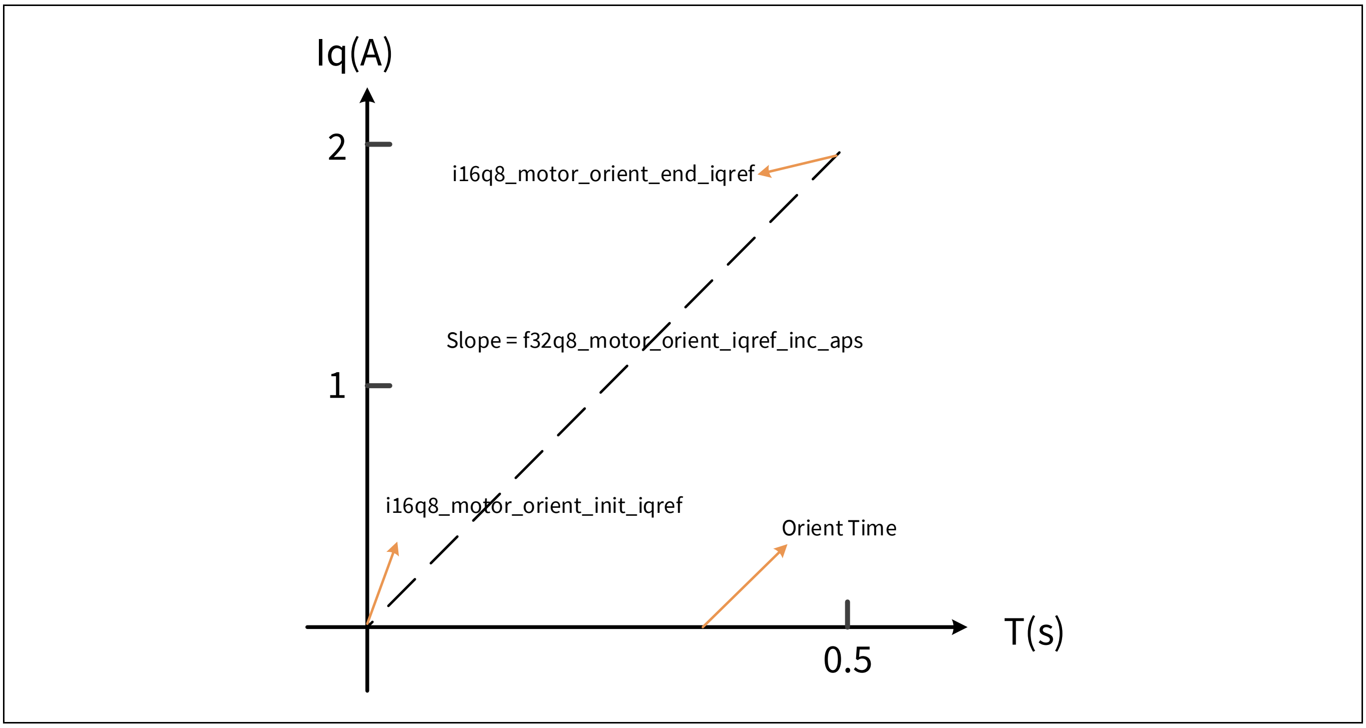


Figure 41 **q-axis current set in orient stage**

- Parameters 6 to 11 are similar to the parameters shown in **Figure 41**.
- The values of parameters `i16q8_motor_open_loop_init_iqref` and `i16q8_motor_open_loop_end_iqref` should be the same as parameter 2.

6.1.2.5 Closed-loop running parameters

The parameters when the motor enters the closed-loop stage are defined in **Table 10**; these mainly include the target speed when the motor switches to closed-loop from open-loop, max and min speed, and acceleration when motor is running.

Table 10 **Closed-loop running parameters**

Variable	Description
`u16_motor_close_loop_target_spdhz`	Target speed when switching to closed-loop. Unit: Hz.
`u8_motor_running_direction`	Motor run direction 0: CW, 1: CCW
`i16q8_motor_close_loop_is_max`	Maximum torque current when motor running. Unit: A.
`i16q8_motor_close_loop_iqref_max`	Maximum value of q axis current reference in closed-loop. Unit: A.
`u16_motor_spdmax`	Motor run maximum speed. Unit: rpm.
`u16_motor_spdmin`	Motor run minimum speed. Unit: rpm.
`f32_motor_spd_acceleration_hz`	Acceleration. Unit: Hz.

f32_motor_spd_deceleration_hz	Deceleration. Unit: Hz.

- u16_motor_close_loop_target_spdhz: Motor switches to closed-loop stage when the motor reaches this speed.
- u8_motor_running_direction: Determines the rotate direction of the motor when you first start the motor. If the direction of rotation does not suit the situation, change it to counter-clockwise.
- i16q8_motor_close_loop_is_max and i16q8_motor_close_loop_iqref_max: Limit the maximum current when the motor is running.
- u16_motor_spdmax and u16_motor_spdmin: Limit the motor speed. The values are set according to the motor's rated speed.
- f32_motor_spd_acceleration_hz, f32_motor_spd_deceleration_hz: Set for the acceleration/deceleration speed when the motor speed is changed. This value should not set too large. You can set it according to your needs.

6.1.2.6 Protection parameters

Table 11 Protection parameters

Variable	Description
i16q8_motor_current_max	Motor phase current peak. Unit: A.
u16_motor_vbus_max	Maximum DC voltage. Unit: V.
u16_motor_vbus_min	Minimum DC voltage. Unit: V.

- i16q8_motor_current_max: Specifies the peak of motor phase current when the motor is running. If the motor current exceeds this value, the system will enter software overcurrent protection process, and MotorCtrl_stcRunPar.u32ErroType will be set to SW_OVER_CURRENT fault.
- u16_motor_vbus_max / u16_motor_vbus_min: Specifies the maximum/minimum value of the bus voltage. If the bus voltage that the ADC sampled is out of this range, the system will enter voltage protection process, and the MotorCtrl_stcRunPar.u32ErroType will be set to OVER_VOLTAGE /UNDER_VOLTAGE fault.

6.1.2.7 Other global parameters

Table 12 Other global parameters

Variables in project	Structure member	Comments
Name: motor_contrl_iq_pid_reg **Type**: stc_pid_t **Location**: *motor_ctrl.h* **Comments**: PID controller for Iq	int32_t **i32q15_kp**	p coefficient for PID calculation
	int32_t **i32q15_ki**	i coefficient for PID calculation
	int32_t **i32q15_kd**	d coefficient for PID calculation
	int32 **i_cnt**	Counter for PI regulator Out calculation
	int32_t **i_timer**	Cycle for PI regulator Out calculation
	int32 **i32_p_out**	Output: Item P
	int32 **i32_i_out**	Output: Item I
	int32 **i32_d_out**	Output: Item D

Variables in project	Structure member	Comments
	int32 **i32_out**	Output: PID regulator
	int32 **i32_outPre**	Last output: PID regulator
	int32 **i32_qn_Iout**	Output: Item I QN format
	int32 **i32_out_max**	Output upper limitation
	int32 **i32_out_min**	Output lower limitation
	int32 **i32_error_0**	Input error
	int32 **i32_errot_1**	Last input error
	int32 **i32_error0_max**	Input error max limit
	int32 **i32_errot0_Min**	Input error min limit
Name: motor_control_id_pid_reg **Type**: stc_pid_t **Location**: *motor_ctrl.h* **Comments**: PID controller for Id	Same as PID_Iq	Same as PID_Iq
Name: motor_control_spd_pid_reg **Type**: stc_pid_t **Location**: *motor_ctrl.h* **Comments**: PID controller for speed	Same as PID_Iq	Same as PID_Iq
Name: motor_control_run_par **Type**: stc_motor_run_t **Location**: *motor_ctrl.h* **Comments**: Structure for motor control	int32_t **i32_target_speed_rpm**	Motor target speed
	int32_t **i32_motor_speed_lpf**	Motor max target speed
	int32_t **i32_target_speed_rpm_max**	Controller output
	int32_t **i32_target_speed_rpm_min**	Motor min target speed
	int32_t **i32q8_estmi_wm_hz**	Motor speed Hz
	int32_t **i32q8_estmi_wm_hzf**	Motor speed Hz Lpf
	uint8_t **u8status**	Motor running status
	uint32_t **u32_error_type**	Motor running error type
	int32_t **i32q8_vbus**	Sampled bus voltage
	int32_t **i32q8_vr**	Sampled VR value
	int32_t **i32q22_delta_theta_ts**	Delta theta
	int32_t **i32q22_delta_theta_kts**	Delta theta calculation factor
	int32_t **i32q8_target_speed_wm_hz**	Motor target speed Hz format
	int32_t **i32Q22_TargetSpeedWmHz**	Motor target speed Hz format

Variables in project	Structure member	Comments
	`int32_t` **`i32Q22_TargetWmIncTs`**	Motor target speed acceleration
	`int32_t` **`i32q22_target_speed_wm_hz`**	Motor target speed deceleration
	`int32_t` **`i32q22_elec_angle`**	Motor target electric angle
	`uint8_t` **`u8_speed_pi_enable`**	Speed PI enable or disable flag
	`uint8_t` **`u8_startup_complete_flag`**	Startup complete flag
	`uint8_t` **`u8_running_stage`**	Motor running stage
	`uint8_t` **`u8_running_level`**	Motor running level
	`uint8_t` **`u8_close_loop_flag`**	Enter closed-loop or not flag
	`uint8_t` **`u8_change_speed_enable`**	Speed change flag
Name: `motor_stc_iuvw_sensed` **Type**: `stc_uvw_t` **Location**: *motor_ctrl.h* **Comments**: Structure for motor current sampling results	`int32_t` **`i32q8_xu`**	Phase-a variable
	`int32_t` **`i32q8_xv`**	Phase-b variable
	`int32_t` **`i32q8_xw`**	Phase-c variable
Name: `motor_stc_iab_sensed` **Type**: `stc_ab_t` **Location**: *motor_ctrl.h* **Comments**: Structure for alpha-beta axis current	`int32_t` **`i32q8_xa`**	Alpha variable of fixed 2-phase
	`int32_t` **`i32q8_xb`**	Beta variable of fixed 2-phase
Name: `motor_stc_idq_sensed` **Type**: `stc_dq_t` **Location**: *motor_ctrl.h* **Comments**: Structure for d-q axis current	`int32_t` **`i32q8_xd`**	d-axis variable
	`int32_t` **`i32q8_xq`**	q-axis variable
	`int32_t` **`i32q12_cos`**	Cosine value with angle
	`int32_t` **`i32q12_sin`**	Sine value with angle
Name: `motor_control_idq_ref` **Type**: `stc_dq_t` **Location**: *motor_ctrl.h* **Comments:** Structure for d-q axis reference current	`int32_t` **`i32q8_xd`**	d-axis variable
	`int32_t` **`i32q8_xq`**	q-axis variable
	`int32_t` **`i32q12_cos`**	Cosine value with angle
	`int32_t` **`i32q12_sin`**	Sine value with angle
Name: `motor_contrl_vdq_ref` **Type**: `stc_dq_t` **Location**: *motor_ctrl.h* **Comments:**	`int32_t` **`i32q8_xd`**	d-axis variable
	`int32_t` **`i32q8_xq`**	q-axis variable
	`int32_t` **`i32q12_cos`**	Cosine value with angle
	`int32_t` **`i32q12_sin`**	Sine value with angle

Variables in project	Structure member	Comments
Structure for d-q axis reference current		

7 Appendix C: Q number format (fixed-point number)

The Q number format is a well-known method to store and process floating-point numbers. It enables faster floating-point operations done by the CPU, so that a separate floating-point unit is not needed. However, some accuracy may be lost by using floating-point.

The example project provided in this application note uses the Q number format. Although understanding the Q number format is not mandatory, gaining a fundamental knowledge of it will help you master the example code faster.

An introduction to the Q number format can be found in Wikipedia. This appendix contains a copy of the "Q (number format)" page from the Wikipedia site, if you are not able to connect to the Internet but need to know about the Q number format when reading this application note.

The following content is from Wikipedia. Infineon does *not* maintain this content for accuracy, nor guarantee that it is up to date. If you have access to the Internet, go to the Wikipedia website to read the latest version by clicking the following link or entering it in your browser.

From Wikipedia, the free encyclopedia:

Q (number format) on Wikipedia: http://en.wikipedia.org/wiki/Q_%28number_format%29

Q is a **fixed point** number format where the number of **fractional bits** (and optionally the number of **integer** bits) is specified. For example, a Q15 number has 15 fractional bits; a Q1.14 number has 1 integer bit and 14 fractional bits. Q format is often used in hardware that does not have a floating-point unit and in applications that require **constant resolution**.

7.1 Characteristics

Q format numbers are (*notionally*) fixed point numbers (but not actually a number itself); that is, they are stored and operated upon as regular binary numbers (i.e. signed integers), thus allowing standard integer hardware/**ALU** to perform **rational number** calculations. The number of integer bits, fractional bits and the underlying word size are to be chosen by the programmer on an application-specific basis—the programmer's choices of the foregoing will depend on the range and resolution needed for the numbers.

Some DSP architectures offer native support for common formats, such as Q1.15. In this case, the processor can support arithmetic in one step, offering saturation (for addition and subtraction) and renormalization (for multiplication) in a single instruction. Most standard CPUs do not. If the architecture does not directly support the particular fixed point format chosen, the programmer will need to handle saturation and renormalization explicitly with bounds checking and bit shifting.

There are 2 conflicting notations for fixed point. Both notations are written as Q*m.n*, where:

- Q designates that the number is in the Q format notation—the "Q" being reminiscent of the standard symbol for the set of **rational numbers**.
- *m.* (optional, assumed to be zero or one) is the number of bits set aside to designate the two's complement integer portion of the number, exclusive or inclusive of the sign bit (therefore if m is not specified it is taken as zero or one).

- *n* is the number of bits used to designate the fractional portion of the number, i.e. the number of bits to the right of the binary point. (If n = 0, the Q numbers are integers—the degenerate case).

One convention includes the sign bit in the value of *m*, and the other convention does not. The choice of convention can be determined by summing *m*+*n*. If the value is equal to the register size, then the sign bit is included in the value of *m*. If it is one less than the register size, the sign bit is not included in the value of *m*.

In addition, the letter U can be prefixed to the Q to indicate an unsigned value, such as UQ1.15, indicating values from 0.0 to +1.99997.

Signed Q values are stored in 2's complement format, just like signed integer values on most processors. In 2's complement, the sign bit is extended to the register size.

For a given Qm.n format, using an m+n+1 bit signed integer container with n fractional bits:

- its range is $[-(2^m), 2^m - 2^{-n}]$
- its resolution is 2^{-n}

For a given UQ*m*.*n* format, using an *m*+*n* bit unsigned integer container with *n* fractional bits:

- its range is $[0, 2^m - 2^{-n}]$
- its resolution is 2^{-n}

For example, a Q14.1 format number:

- requires 14+1+1 = 16 bits
- its range is $[-2^{14}, 2^{14} - 2^{-1}]$ = [-16384.0, +16383.5] = [0x8000, 0x8001 … 0xFFFF, 0x0000, 0x0001 … 0x7FFE, 0x7FFF]
- its resolution is 2^{-1} = 0.5
- Unlike **floating point** numbers, the resolution of Q numbers will remain constant over the entire range.

7.2 Conversion

Float to Q

To convert a number from **floating point** to Q*m*.*n* format:

1. Multiply the floating point number by 2n
2. Round to the nearest integer

Q to float

To convert a number from Q*m*.*n* format to floating point:

1. Convert the number to floating point as if it were an integer
2. Multiply by 2−n

7.3 Math operations

Q numbers are a ratio of two integers: the numerator is kept in storage, the denominator is equal to 2n.

Consider the following example:

The Q8 denominator equals 2^8 = 256

1.5 equals 384/256

384 is stored, 256 is inferred because it is a Q8 number.

If the Q number's base is to be maintained (n remains constant) the Q number math operations must keep the denominator constant. The following formulas shows math operations on the general Q numbers N_1 and N_2.

$$\frac{N_1}{d} + \frac{N_2}{d} = \frac{N_1 + N_2}{d}$$
$$\frac{N_1}{d} - \frac{N_2}{d} = \frac{N_1 - N_2}{d}$$
$$\left(\frac{N_1}{d} \times \frac{N_2}{d}\right) \times d = \frac{N_1 \times N_2}{d}$$
$$\left(\frac{N_1}{d} / \frac{N_2}{d}\right) / d = \frac{N_1 / N_2}{d}$$

Because the denominator is a power of two the multiplication can be implemented as an arithmetic shift to the left and the division as an arithmetic shift to the right; on many processors shifts are faster than multiplication and division.

To maintain accuracy the intermediate multiplication and division results must be double precision and care must be taken in rounding the intermediate result before converting back to the desired Q number.

Using C the operations are (note that here, Q refers to the fractional part's number of bits):

7.3.1 Addition

```
signed int a, b, result;
result = a + b;
```

With saturation

```
signed int a, b, result;
signed long int tmp;
tmp = a + b;
if (tmp > 0x7FFFFFF) tmp = 0x7FFFFFFF;
if (tmp < -1 * 0x7FFFFFFF) tmp = -1 * 0x7FFFFFFF;
result = (signed int) tmp;
```

7.3.2 Subtraction

```
signed int a, b,result;
result = a - b;
```

7.3.3 Multiplication

```
// precomputed value:
#define K   (1 << (Q-1))
signed int        a, b, result;
signed long int  temp;
temp = (long int)a * (long int)b; // result type is operand's type
// Rounding; mid values are rounded up
temp += K;
// Correct by dividing by base
result = temp >> Q;
```

7.3.4 Division

```
signed int a, b, result;
signed long int temp;
// pre-multiply by the base (Upscale to Q16 so that the result will be in Q8
format)
temp = (long int)a << Q;
// So the result will be rounded ; mid values are rounded up.
temp += b/2;
result = temp/b;
```

References

[1] **ModusToolbox™ home page**

[2] **AN228571 - Getting started with PSoC™ 6 MCU on ModusToolbox™ software**

[3] **CY8CKIT-037 motor control evaluation kit**

[4] **CY8CKIT-062S4**

Revision history

Document version	Date of release	Description of changes
**	2022-07-14	Initial release

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2022-07-14

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to www.infineon.com/support

Document reference

002-35096 Rev. **