Freescale Semiconductor, Inc.
Design Reference Manual

Document Number: DRM148
Rev. 1, 02/2016

Sensorless PMSM Field-Oriented Control

1. Introduction

Sinusoidal Permanent Magnet Synchronous Motors (PMSM) are getting more and more popular, replacing brushed DC, universal, and other motors in a wide application area. Here are some of the reasons:

- Better reliability when compared to brushed DC motors.
- Better efficiency when compared to AC Induction Motors (ACIM) because the rotor flux is generated by magnets.
- Lower acoustic noise.
- Smaller dimensions when compared to ACIM.

The disadvantage of the PMSM drives is the need for a more sophisticated electronic circuit. Nowadays, most applications need electronic speed or torque regulation and other features with the necessity of electronic control. When a system with electronic control is used, it only takes small system cost increase to implement more advanced drives, such as the sinusoidal PMSM with a digitally-controlled switching inverter and a DC-bus circuit. It is only necessary to have a cost-effective controller with a good calculation performance. PMSMs are used in domestic appliances (such as refrigerators, washing machines, dishwashers), pumps, compressors, fans, and other appliances that require high reliability and efficiency.

Contents

The control algorithms are divided into two general groups. The first group is called scalar control (SC). The constant Volt per Hertz control represents the scalar control. The other group is called vector control (VC), or field-oriented control (FOC). The FOC technique brings overall improvements in drive performance when compared to the scalar control (higher efficiency, full torque control from zero to nominal motor speed, decoupled control of flux and torque, improved dynamics, etc.). For the PMSM control, it is necessary to know the exact rotor position. This application uses special algorithms to estimate the speed and position instead of using a physical position sensor.

The software algorithms consist of software blocks (such as the state machine, motor-control drivers (MCDRV), and motor-control application tuning (MCAT) tool) supporting online application tuning. This design reference manual describes the basic motor theory and software design.

2. PMSM Control Theory

2.1. 3-phase PMSM

The PMSM is a rotating electric machine with a standard 3-phase stator, like that of an induction motor; the rotor has surface-mounted permanent magnets. See this figure:

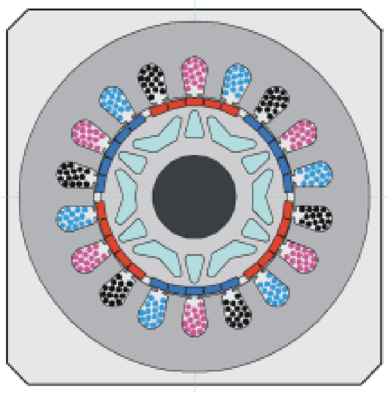

Figure 1. PMSM cross-section

In this aspect, the PMSM is equivalent to induction motors, where the air gap magnetic field is produced by permanent magnets, so the rotor magnetic field is constant. The PMSMs offer a number of advantages in designing modern motion-control systems. The use of a permanent magnet to generate a substantial air gap magnetic flux makes it possible to design highly efficient PMSMs.

2.2. Mathematical description of PMSM

There are several PMSM models. The model used for the vector control design is obtained using the space-vector theory. The 3-phase motor quantities (such as voltages, currents, magnetic flux, etc.) are expressed in terms of complex space vectors. Such model is valid for any instantaneous variation of voltage and current, and adequately describes the performance of the machine under both the steady-state and transient operations.

Complex space vectors are described using only two orthogonal axes. Using the 2-phase motor model reduces the number of equations and simplifies the control design.

2.2.1. Space vector definitions

Assume that i_{sa}, i_{sb}, and i_{sc} are the instantaneous balanced 3-phase stator currents:

$$\textit{Eq. 1} \qquad i_{sa} + i_{sb} + i_{sc} = 0$$

Then the stator-current space vector can be defined as:

$$\textit{Eq. 2} \qquad \bar{i_s} = k(i_{sa} + a i_{sb} + a^2 i_{sc})$$

where a and a^2 are the spatial operators $a = e^{j2\pi/3}$, $a^2 = e^{j4\pi/3}$, and k is the transformation constant $k=2/3$. Figure 2 shows the stator-current space vector projection. The space vector defined by Eq. 3 can be expressed using the 2-axis theory. The real part of the space vector is equal to the instantaneous value of the direct-axis stator current component $i_{s\alpha}$, and its imaginary part is equal to the quadrature-axis stator current component $i_{s\beta}$. Thus, the stator-current space vector in the stationary reference frame attached to the stator is expressed as:

$$\textit{Eq. 3} \qquad \bar{i_s} = i_{s\alpha} + j i_{s\beta}$$

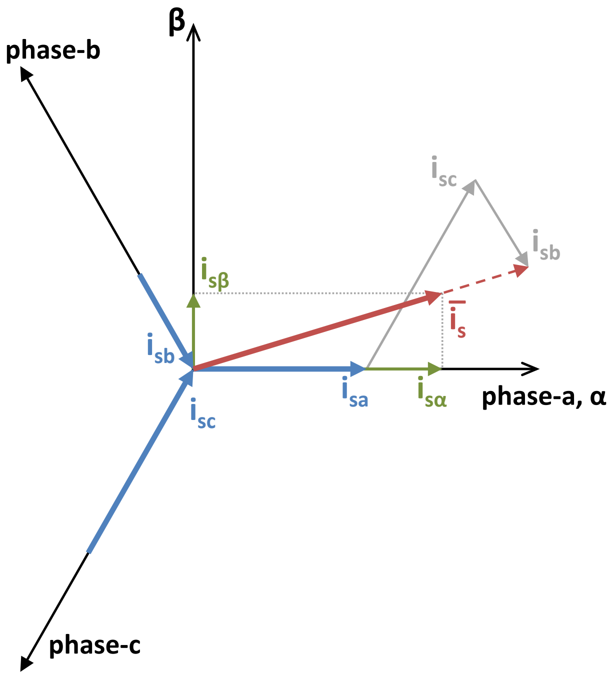

Figure 2. Stator current space vector and its projection

In symmetrical 3-phase machines, the direct and quadrature axis stator currents ($i_{s\alpha}$, $i_{s\beta}$) are fictitious quadrature-phase (2-phase) current components, which are related to the actual 3-phase stator currents as:

$$\textit{Eq. 4} \qquad i_{s\alpha} = k\left(i_{sa} - \tfrac{1}{2} i_{sb} - \tfrac{1}{2} i_{sc}\right)$$

Eq. 5 $$i_{s\beta} = k\frac{\sqrt{3}}{2}(i_{sb} - i_{sc})$$

where *k=2/3* is the transformation constant, so that the final equation is:

Eq. 6 $$i_{s\beta} = \frac{1}{\sqrt{3}}(i_{sb} - i_{sc})$$

The space vectors of other motor quantities (such as voltages, currents, magnetic fluxes) can be defined in the same way as the stator-current space vector.

2.2.2. PMSM model

The symmetrical 3-phase smooth-air-gap machine with sinusoidally-distributed windings is taken as the description of a PMSM. The voltage equations of the stator in the instantaneous form are expressed as:

Eq. 7 $$u_{SA} = R_S i_{SA} + \frac{d}{dt}\psi_{SA}$$

Eq. 8 $$u_{SB} = R_S i_{SB} + \frac{d}{dt}\psi_{SB}$$

Eq. 9 $$u_{SA} = R_S i_{SA} + \frac{d}{dt}\psi_{SA}$$

where u_{SA}, u_{SB}, and u_{SC} are the instantaneous values of stator voltages, i_{SA}, i_{SB}, and i_{SC} are the instantaneous values of stator currents, and ψ_{SA}, ψ_{SB}, ψ_{SC} are the instantaneous values of the stator flux linkages in phases *SA*, *SB*, and *SC*. Due to a large number of equations in the instantaneous form, (Eq. 7, Eq. 8, and Eq. 9), it is better to rewrite the instantaneous equations using the 2-axis theory (Clarke transformation). The PMSM is then expressed as:

Eq. 10 $$u_{S\alpha} = R_S i_{S\alpha} + \frac{d}{dt}\psi_{S\alpha}$$

Eq. 11 $$u_{S\beta} = R_S i_{S\beta} + \frac{d}{dt}\psi_{S\beta}$$

Eq. 12 $$\psi_{S\alpha} = L_S i_{S\alpha} + \psi_M \cos\Theta_r$$

Eq. 13 $$\psi_{S\beta} = L_S i_{S\beta} + \psi_M \sin\Theta_r$$

Eq. 14 $$\frac{d\omega}{dt} = \frac{p}{J}\left[\frac{2}{3}p\left(\psi_{S\alpha} i_{S\beta} - \psi_{S\beta} i_{S\alpha}\right) - T_L\right]$$

For a glossary of symbols, see Table 3.

The equations Eq. 10 to Eq. 14 represent the model of a PMSM in the stationary frame α, β, fixed to the stator.

Besides the stationary reference frame attached to the stator, the motor model voltage space vector equations can be formulated in a general reference frame that rotates at a general speed ω_g. Using a general reference frame with the direct and quadrature axes (*x* and *y*) rotating at a general instantaneous speed $\omega_g = d\theta_g/dt$ (as shown in Figure 3) where θ_g is the angle between the direct axis of the stationary reference frame (α) attached to the stator and the real axis (*x*) of the general reference frame, Eq. 15 defines the stator-current space vector in a general reference frame as:

Eq. 15 $$\overline{i_{sg}} = \overline{i_s} e^{-j\theta_g} = i_{sx} + j i_{sy}$$

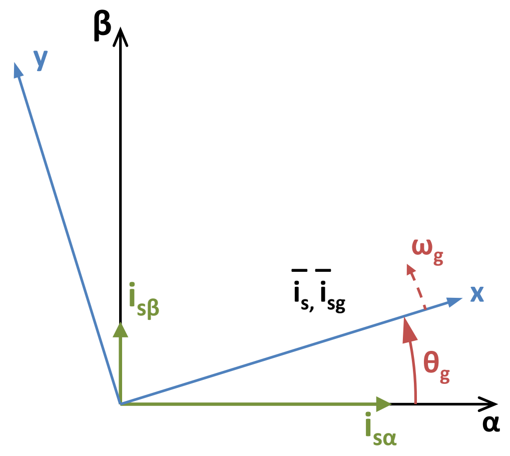

Figure 3. Application of general reference frame

The stator-voltage and flux-linkage space vectors can be obtained similarly in the general reference frame.

Similar considerations are held for the space vectors of the rotor voltages, currents, and flux linkages. The real axis (rα) of the reference frame attached to the rotor is displaced from the direct axis of the stator reference frame by the rotor angle θ_r. The angle between the real axis (x) of the general reference frame and the real axis of the reference frame rotating with the rotor (rα) is θ_g-θ_r. In the general reference frame, the space vector of the rotor currents is expressed as:

$$\overline{i_{rg}} = \overline{i_r} e^{-j(\theta_g - \theta_r)} = i_{rx} + j i_{ry} \quad \text{Eq. 16}$$

where $\overline{i_r}$ is the space vector of the rotor current in the rotor reference frame.

The space vectors of the rotor voltages and rotor flux linkages in the general reference frame can be expressed similarly.

The motor model voltage equations in the general reference frame can be expressed using the introduced transformations of the motor quantities from one reference frame to the general reference frame.
The PMSM model is often used in vector-control algorithms. The aim of the vector control is to implement the control schemes that produce high-dynamic performance and are similar to those used to control DC machines. To achieve this, the reference frames must be aligned with the stator flux-linkage space vector, the rotor flux-linkage space vector, or the magnetizing space vector. The most popular reference frame is the reference frame attached to the rotor flux-linkage space vector with the direct axis (d) and the quadrature axis (q).

Transformed into the d, q coordinates, the motor model is:

$$u_{Sd} = R_S i_{Sd} + \frac{d}{dt}\psi_{Sd} - \omega_F \psi_{Sq} \quad \text{Eq. 17}$$

$$u_{Sq} = R_S i_{Sq} + \frac{d}{dt}\psi_{Sq} - \omega_F \psi_{Sd} \quad \text{Eq. 18}$$

$$\psi_{Sd} = L_S i_{Sd} + \psi_M \quad \text{Eq. 19}$$

$$\psi_{Sq} = L_S i_{Sq} \quad \text{Eq. 20}$$

Eq. 21 $$\frac{d\omega}{dt} = \frac{p}{J}\left[\frac{2}{3}p(\psi_{Sd}i_{Sd} - \psi_{Sq}i_{Sq}) - T_L\right]$$

Considering that the below base speed is i_{sd}=0, Eq. 21 can be reduced as:

Eq. 22 $$\frac{d\omega}{dt} = \frac{p}{J}\left[\frac{2}{3}p(\psi_{M}i_{Sq}) - T_L\right]$$

In Eq. 22, the torque is dependent and can be directly controlled by the current i_{sq} only.

2.3. PMSM field-oriented control

2.3.1. Fundamental principle of field-oriented control (FOC)

High-performance motor control is characterized by smooth rotation over the entire speed range of the motor, full torque control at zero speed, and fast accelerations and decelerations. To achieve such control, use the FOC techniques for 3-phase AC motors. The basic idea of the FOC algorithm is to decompose the stator current into the magnetic field-generating part and the torque-generating part. Both components can be controlled separately after the decomposition. The structure of the motor controller is then as simple as that for separately excited DC motors.

Figure 4 shows the basic structure of the vector-control algorithm for the PMSM. To perform the vector control, follow these steps:

1. Measure the motor quantities (phase voltages and currents).
2. Transform them into the 2-phase system (α, β) using Clarke transformation.
3. Calculate the rotor-flux space vector magnitude and position angle.
4. Transform the stator currents into the d, q reference frame using Park transformation.
5. The stator current torque-producing (i_{sq}) and flux-producing (i_{sd}) components are controlled separately.
6. The stator-voltage space vector is transformed by the inverse Park transformation back from the d, q reference frame into the 2-phase system fixed with the stator.
7. The 3-phase output voltage is generated using the space vector modulation.

To decompose the currents into torque-producing and flux-producing components (i_{sd}, i_{sq}), you must know the rotor position. This requires an accurate sensing of the rotor position and velocity information. The incremental encoders or resolvers attached to the rotor are mostly used as position transducers for vector-control drives. In some applications, the use of speed/position sensors is not wanted. The aim is not to measure the speed/position directly, but to employ some indirect techniques to estimate the rotor position instead. The algorithms that do not employ speed sensors are called “sensorless control”.

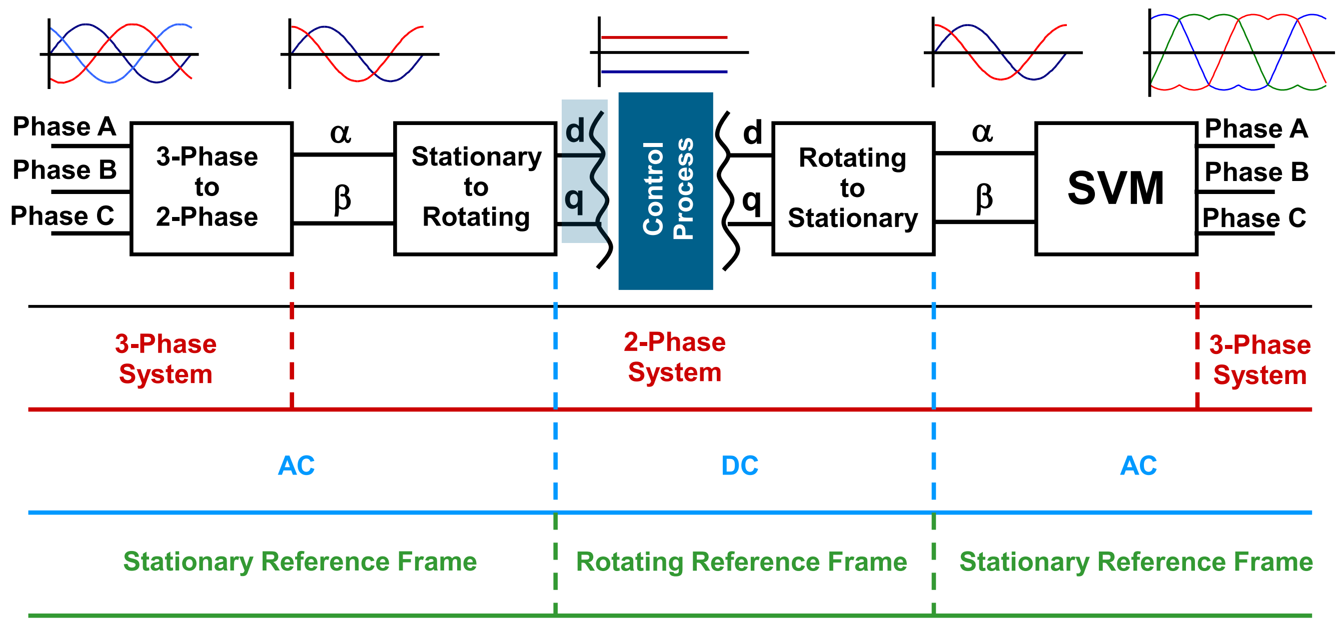

Figure 4. FOC transformations

2.3.2. FOC algorithm description

The overview block diagram of the implemented control algorithm is shown in Figure 5. As with the other FOC techniques, it is able to control the field and torque of the motor separately. The aim of the control is to regulate motor speed. The speed command value is set by the high-level control. The algorithm is executed in two control loops. The fast inner control loop is executed with a period of 100 µs. The slow outer control loop is executed with a period of 1 ms.

To achieve the goal of the PMSM control, the algorithm uses feedback signals. The essential feedback signals are the 3-phase stator current and DC-bus voltage. The regulator output is used for the stator voltage. To operate properly, the presented control structure requires either the position and speed sensors on the motor shaft or an advanced algorithm to estimate the position and speed.

The fast control loop executes these two independent current-control loops:

- The direct-axis current (i_{sd}) PI controllers—the direct-axis current (i_{sd}) is used to control the rotor magnetizing flux.
- The quadrature-axis current (i_{sq}) PI controllers—the quadrature-axis current corresponds to the motor torque.

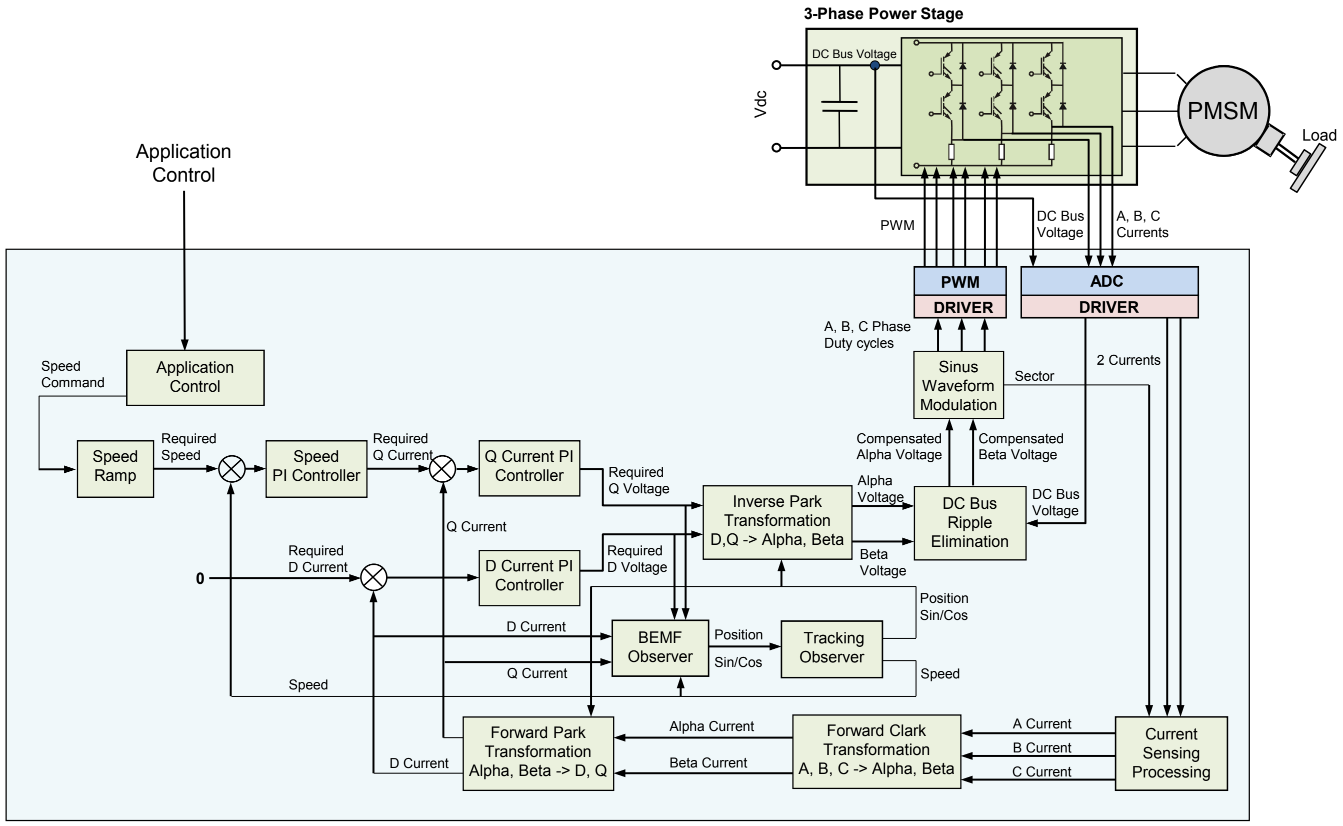

Figure 5. PMSM FOC algorithm overview

The fast control loop executes these tasks that are necessary to achieve an independent control of the stator-current components:

- 3-phase current reconstruction
- Forward Clark transformation
- Forward and backward Park transformations
- DC-bus voltage ripple elimination
- Space vector modulation (SVM)

The slow control loop executes the speed controller and lower-priority control tasks. The PI speed controller output sets a reference for the torque-producing quadrature-axis component of the stator current (i_{sq}).

2.3.3. Space vector modulation

The space vector modulation (SVM) directly transforms the stator-voltage vectors from the 2-phase α, β-coordinate system into pulse width modulation (PWM) signals (duty-cycle values).

The standard technique of output voltage generation uses the inverse Clarke transformation to obtain 3-phase values. The duty cycles needed to control the power stage switches are then calculated using the phase voltage values. Although this technique provides good results, space vector modulation is more straightforward (valid only for transformation from the α, β-coordinate system).

The basic principle of the standard space vector modulation technique can be explained using the power stage schematic diagram shown in Figure 6. Regarding the 3-phase power stage configuration (as shown in Figure 6) eight possible switching states (vectors) are feasible. They are given by the combinations of the corresponding power switches. The graphical representation of all combinations is the hexagon shown in Figure 7. There are six non-zero vectors (U_0, U_{60}, U_{120}, U_{180}, U_{240}, U_{300}) and two zero vectors (O_{000} and O_{111}) defined in the α, β coordinates.

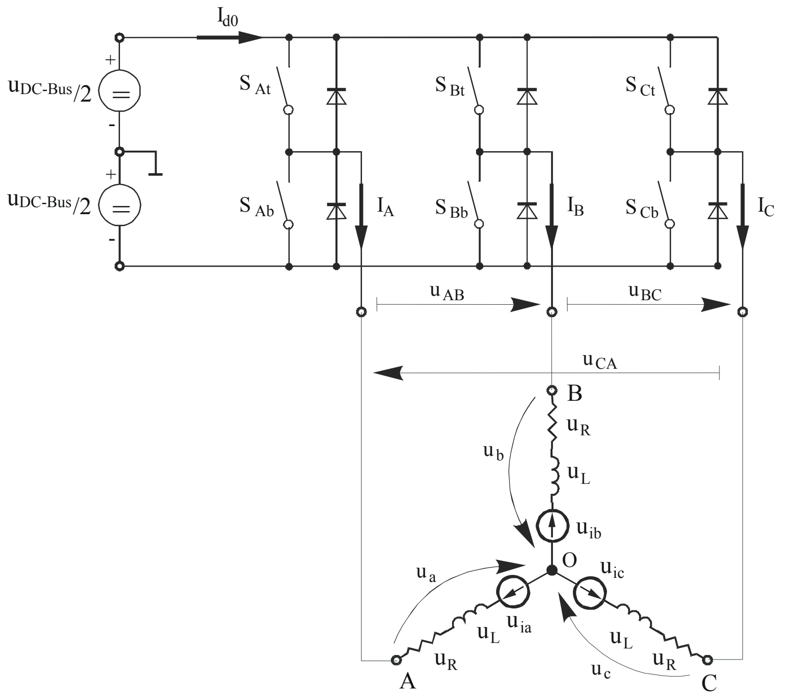

Figure 6. Power stage schematic diagram

The combination of ON/OFF states in the power stage switches for each voltage vector is coded in Figure 7 by the three-digit number in parentheses. Each digit represents one phase. For each phase, a value of 1 means that the upper switch is ON and the bottom switch is OFF. A value of 0 means that the upper switch is OFF and the bottom switch is ON. These states, together with the resulting instantaneous output line-to-line voltages, phase voltages, and voltage vectors, are listed in Table 1.

Table 1. Switching paterns and resulting instantaneous voltages

a	b	c	U_a	U_b	U_c	U_{AB}	U_{BC}	U_{CA}	Vector
0	0	0	0	0	0	0	0	0	O_{000}
1	0	0	$2U_{DC-Bus}/3$	$-U_{DC-Bus}/3$	$-U_{DC-Bus}/3$	U_{DC-Bus}	0	$-U_{DC-Bus}$	U_0
1	1	0	$U_{DC-Bus}/3$	$U_{DC-Bus}/3$	$-2U_{DC-Bus}/3$	0	U_{DC-Bus}	$-U_{DC-Bus}$	U_{60}
0	1	0	$-U_{DC-Bus}/3$	$2U_{DC-Bus}/3$	$-U_{DC-Bus}/3$	$-U_{DC-Bus}$	U_{DC-Bus}	0	U_{120}
0	1	1	$-2U_{DC-Bus}/3$	$U_{DC-Bus}/3$	$U_{DC-Bus}/3$	$-U_{DC-Bus}$	0	U_{DC-Bus}	U_{240}
0	0	1	$-U_{DC-Bus}/3$	$-U_{DC-Bus}/3$	$2U_{DC-Bus}/3$	0	$-U_{DC-Bus}$	U_{DC-Bus}	U_{300}
1	0	1	$U_{DC-Bus}/3$	$-2U_{DC-Bus}/3$	$U_{DC-Bus}/3$	U_{DC-Bus}	$-U_{DC-Bus}$	0	U_{360}
1	1	1	0	0	0	0	0	0	O_{111}

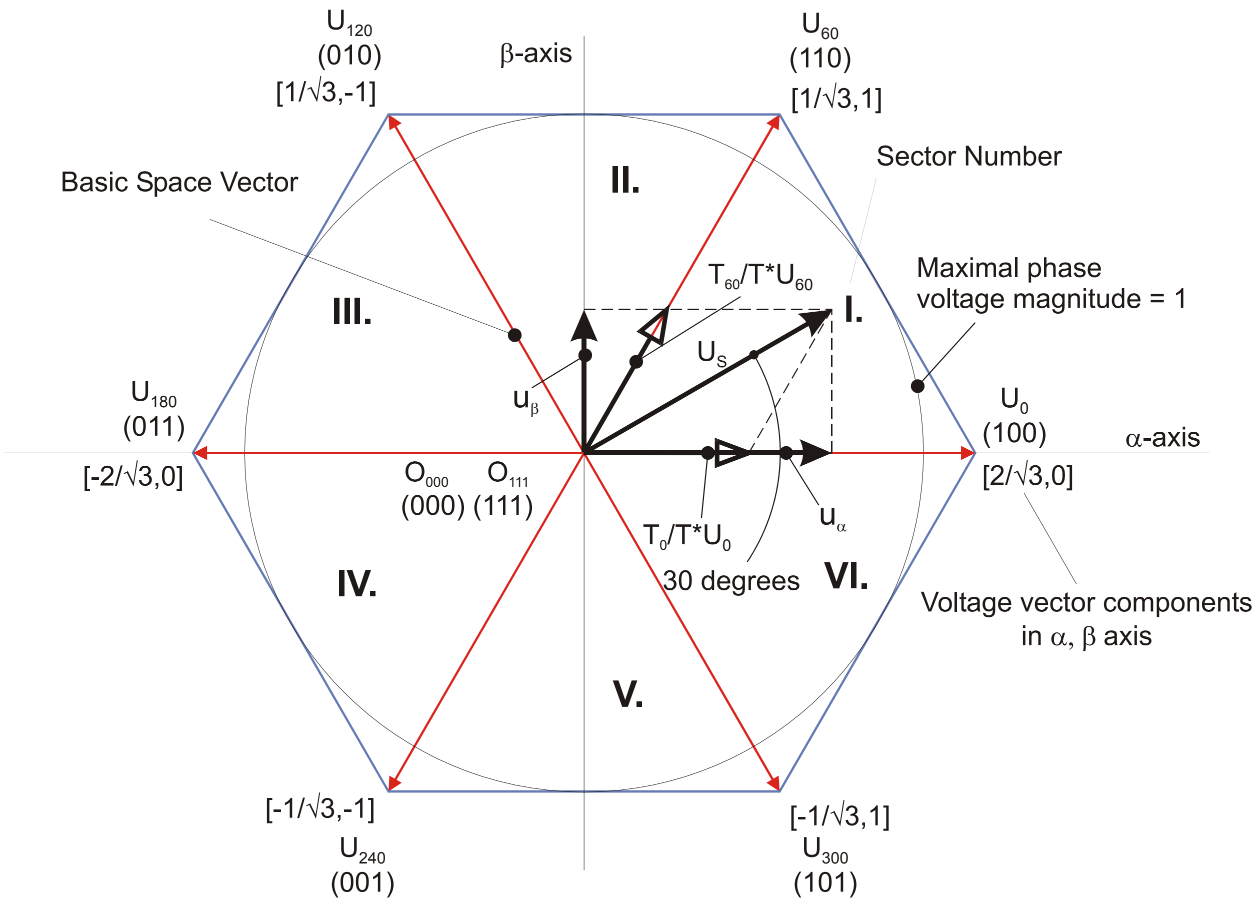

Figure 7. Basic space vectors and voltage vector projection

SVM is a technique used as a direct bridge between the vector control (voltage space vector) and PWM.

The SVM technique consists of these steps:

1. Sector identification
2. Space voltage vector decomposition into directions of sector base vectors U_x and $U_{x\pm60}$
3. PWM duty cycle calculation

The principle of SVM is the application of voltage vectors U_{xxx} and O_{xxx} for certain instances in such way that the “mean vector” of the PWM period T_{PWM} is equal to the desired voltage vector.

This method provides greatest variability in arranging of the zero and non-zero vectors during the PWM period. You can arrange these vectors to lower switching losses; another might want to reach a different result, such as center-aligned PWM, edge-aligned PWM, minimal switching, and so on.

For the SVM chosen, this rule is defined:

- The desired space voltage vector is created only by applying the sector base vectors: the non-zero vectors on the sector side, (U_x, $U_{x\pm60}$) and the zero vectors (O_{000} or O_{111}).

These expressions define the principle of SVM:

Eq. 23 $$T_{PWM} \cdot U_{S[\alpha,\beta]} = T_1 U_X + T_2 U_{X\pm60} + T_0(O_{000})$$

Eq. 24 $$T_{PWM} = T_1 + T_2 + T_0$$

To solve the time periods T_0, T_1, and T_2, decompose the space voltage vector $U_{S[\alpha,\beta]}$ into directions of the sector base vectors U_x and $U_{x\pm60}$. Eq. 23 splits into Eq. 25 and Eq. 26.

Eq. 25 $$T_{PWM} \cdot U_{SX} = T_1 \cdot U_X$$

Eq. 26 $$T_{PWM} \cdot U_{S(X\pm60)} = T_1 \cdot U_{X\pm60}$$

By solving this set of equations, you can obtain the necessary duration for the application of the sector base vectors U_x and $U_{x\pm60}$ during the PWM period T_{PWM} to produce the right stator voltages.

Eq. 27 $$T_1 = \frac{|U_{SX}|}{|U_X|} T_{PWM} \text{ for vector } U_X$$

Eq. 28 $$T_2 = \frac{|U_{SX}|}{|U_{X\pm60}|} T_{PWM} \text{ for vector } U_{X\pm60}$$

Eq. 29 $$T_0 = T_{PWM} - (T_1 + T_2) \text{ either for for vector } O_{000} \text{ or } O_{111}$$

2.3.4. Position sensor elimination

The first stage of the proposed overall control structure is the alignment algorithm of the rotor PM to set an accurate initial position. This enables applying full start-up torque to the motor. In the second stage, the field-oriented control is in the open-loop mode to push the motor up to a speed where the observer provides accurate speed and position estimations. When the observer provides the appropriate estimates, the rotor speed and position calculation is based on the estimation of the BEMF in the stationary reference frame using a Luenberger-type observer.

2.3.5. Motor position alignment

In this design, the quadrature encoder is used as a motor position and speed sensor. Because the quadrature encoder does not provide the absolute position, the exact rotor position must be known before starting the motor. One of the simplest methods is to align the rotor to a predefined position. The motor is powered by a selected static voltage pattern (usually the zero position in the sinewave table) and the rotor aligns to the predefined position. The alignment is done only once (during the first motor startup). The figure below shows the motor alignment. Before applying the constant current vector to the stator, the rotor position is not known. After the stabilization period, the rotor flux is aligned to the stator flux. This is true when the external load torque is low enough when compared to the torque produced by the alignment vector.

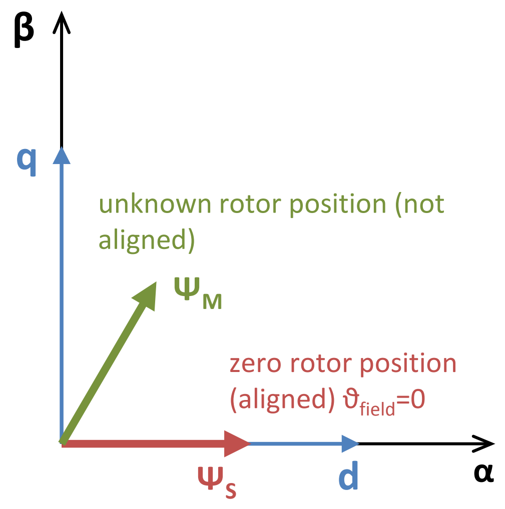
β
q
unknown rotor position (not aligned)
Ψ_M
zero rotor position (aligned) $\vartheta_{field}=0$
Ψ_S
d
α

Figure 8. Rotor alignment

2.3.6. Open-loop startup

After identifying the initial rotor position, the field-oriented control is used in the open-loop mode (only in case of the sensorless control). The current set-point is determined by the speed controller, which generates the torque reference current i_{Qref}, and the proportional integral controller of the speed control loop is initialized to the maximum current possible. The angular speed feedback ω_{FBCK} is kept at zero during the open-loop operation, and the vector transformations are fed a time-varying reference position signal derived by integrating the speed-ramp reference. This strategy moves the motor up to the speed where the observer provides accurate speed and position estimates.

The implementation of the open-loop startup is described in Section 3.8.4, “Motor open-loop startup”.

2.3.7. Back-EMF observer

When the PMSM reaches the minimum operating speed, the minimum-measurable level of the back-EMF is generated by the rotor permanent magnets. The back-EMF observer then gradually transitions into the closed-loop mode. The feedback loops are then controlled by the estimated angle and speed signals from the back-EMF observer.

This position and angular speed estimation method is based on the motor mathematical model with an extended electromotive force function. This extended back-EMF model includes the position information from the conventionally defined back-EMF and the stator inductance. This enables obtaining the rotor position and velocity information by estimating the extended back-EMF only.

Eq. 30
$$\begin{bmatrix} u_\gamma \\ u_\delta \end{bmatrix} = \begin{bmatrix} R_S + sL_D & -\omega_r L_Q \\ \omega_r L_Q & R_S + sL_D \end{bmatrix} \begin{bmatrix} i_\gamma \\ i_\delta \end{bmatrix} + \left[\Delta L \cdot \left(\omega_{el} i_D - i'_Q\right) + k_e \omega_e\right] \cdot \begin{bmatrix} -\sin\theta_{error} \\ \cos\theta_{error} \end{bmatrix}$$

The observer is applied to the PMSM motor with an estimator model excluding the extended back-EMF term. The extended back-EMF term is estimated using the observer (as shown in Figure 9), which utilizes a simple observer of the PMSM stator current. The back-EMF observer presented here is realized within the rotating reference frame (d,q). The estimator of the d,q axes consists of the stator current observer based on the RL motor circuit with estimated motor parameters. This current observer is fed the sum of the actual applied motor voltage (cross-coupled rotational term) which corresponds to the motor saliency (L_d-L_q), and the compensator corrective output. The observer provides the back-EMF signals as a disturbance, because the back-EMF is not included in the observer model.

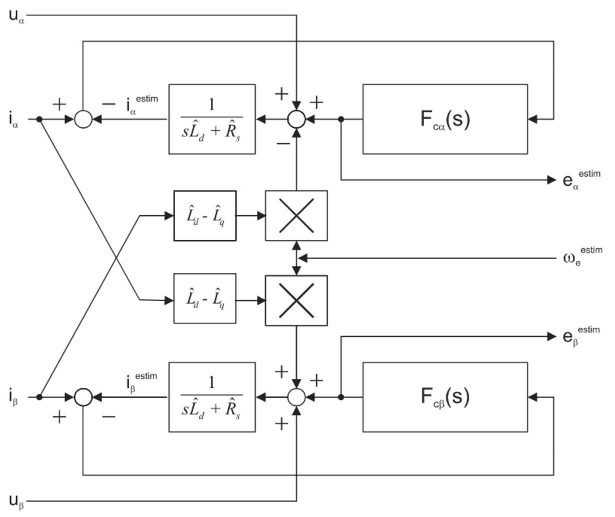

Figure 9. Luenberger-type stator current observer acting as state filter for back-EMF

2.3.8. Extracting speed and position

The tracking observer algorithm is used to obtain the speed and position information from the position error. This algorithm adopts the phase-locked loop mechanism and requires a single input argument as a phase error. The phase tracking observer with a standard PI controller used as the loop compensator is shown in Figure 10.

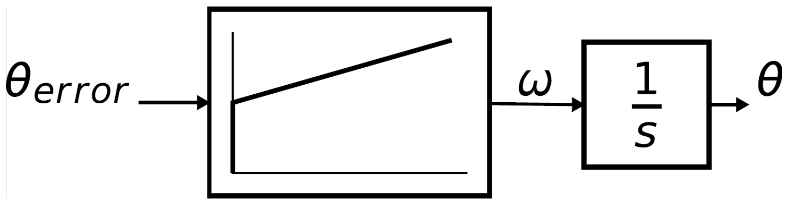

Figure 10. Block diagram of proposed PLL scheme for position estimation

3. Software Design

This application is written as a simple C project with the inclusion of Embedded Software Libraries (FSLESL). Download these libraries from www.nxp.com/fslesl. They contain the necessary algorithms, math functions, observers, filters, and other features used in the application.

This section describes the software design of the sensorless PMSM FOC application. It describes the numerical scaling in the fixed-point fractional arithmetic of the controller first. Then it describes specific issues, such as speed and current sensing. In the end, it describes the control software implementation.

3.1. Data types

This application uses several data types: (un)signed integer, fractional, and accumulator. The integer data types are useful for general-purpose computation; they are familiar to the MPU and MCU programmers. The fractional data types enable implementing numeric and digital signal processing algorithms. The accumulator data type is a combination of both; that means it has the integer and fractional portions.

This list shows the integer types defined in the libraries:

- Unsigned 16-bit integer—<0 ; 65535> with a minimum resolution of 1
- Signed 16-bit integer—<-32768 ; 32767> with a minimum resolution of 1
- Unsigned 32-bit integer—<0 ; 4294967295> with a minimum resolution of 1
- Signed 32-bit integer—<-2147483648 ; 2147483647> with a minimum resolution of 1

This list shows the fractional types defined in the libraries:

- Fixed-point 16-bit fractional—<-1 ; $1–2^{-15}$> with a minimum resolution of 2^{-15}
- Fixed-point 32-bit fractional—<-1 ; $1–2^{-31}$> with a minimum resolution of 2^{-31}

This list shows the accumulator types defined in the libraries:

- Fixed-point 16-bit accumulator—<-256.0 ; $256.0–2^{-7}$> with a minimum resolution of 2^{-7}
- Fixed-point 32-bit accumulator—<-65536.0 ; $65536.0–2^{-15}$> with a minimum resolution of 2^{-15}

3.2. Scaling of analog quantities

This equation shows the relationship between the real and fractional representations:

$$\textit{Eq. 31} \qquad \mathit{fractional\ value} = \frac{\mathit{Real\ value}}{\mathit{Real\ quantity\ range}}$$

where:

- Fractional Value—fractional representation of quantities [–]
- Real Value—real quantity in physical units [..]
- Real Quantity Range—maximum defined quantity value used for scaling in physical units [..]

Some examples of the quantities' scaling are provided in the following subsections.

3.2.1. Voltage scale

Voltage is usually measured on the voltage resistor divider by the ADC. Therefore, the maximum voltage scale is proportional to the maximum ADC input voltage range. In this example, the maximum voltage is 472 V. This example shows the usage of the fractional voltage variable:

Voltage scale: V_{max} = 472 V

Measured voltage: $V_{measured}$ = 352 V

$$\textit{Eq. 32} \qquad (\mathit{Frac16})\mathit{voltage}_{variable} = \frac{V_{measured}}{V_{max}} = \frac{352V}{472V} = 0.7458$$

This 16-bit fractional variable is stored internally as a 16-bit integer variable:

$$\textit{Eq. 33} \qquad (\mathit{Int16})\mathit{voltage}_{variable} = (\mathit{Frac16})\mathit{voltage}_{variable}.2^{15} = 0.7458.2^{15} = 24438$$

Figure 11 shows the previous equations of voltage scaling into a fractional number. The voltage value is read by the ADC as a 12-bit signed number with left justification for a 16-bit number.

In case a floating-point number representation of this value is needed, the fractional number is converted to a floating-point number as:

$$\textit{Eq. 34} \qquad (\mathit{float})\mathit{voltage}_{variable} = \frac{V_{max}}{2^{15}}(\mathit{Frac16})\mathit{voltage}_{variable}$$

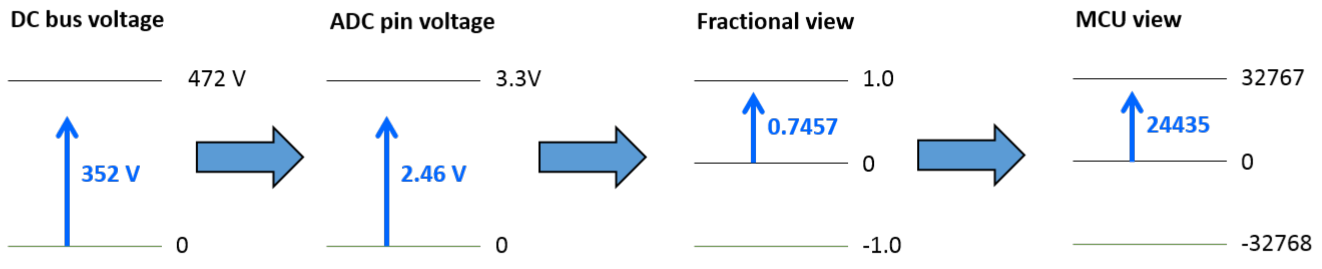

Figure 11. Voltage measurement

3.2.2. Current scale

Current is usually measured as a voltage drop on the shunt resistors, which is amplified by an operational amplifier. If both the positive and negative currents are needed, the amplified signal has an offset. The offset is generally half of the ADC range. The maximum current scale is then proportional to a half of the maximum ADC input voltage range (see Figure 12). Manipulation with the current variable is similar to manipulation with the voltage variable.

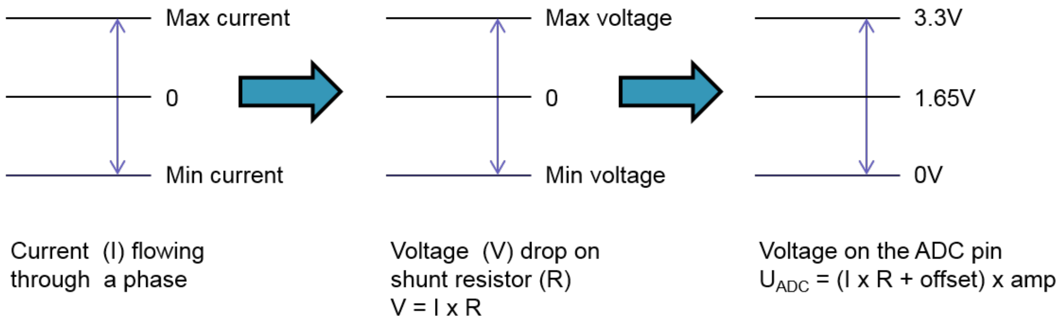
Max current
0
Min current
Max voltage
0
Min voltage
3.3V
1.65V
0V
Current (I) flowing through a phase
Voltage (V) drop on shunt resistor (R)
V = I x R
Voltage on the ADC pin
U_{ADC} = (I x R + offset) x amp

Figure 12. Current measurement

3.2.3. Angle scale

The angles (such as rotor position) are represented as fixed-point 32-bit accumulators, where the lower 16-bits are in the range <-1 , 1), which corresponds to an angle in the range <-π , π). In a 16-bit signed integer value, the angle is represented as:

$$-\pi = 0x8000 \quad \text{Eq. 35}$$

$$\pi = 0x7FFF \quad \text{Eq. 36}$$

3.3. Application overview

Figure 13 shows the FOC algorithm blocks of a PMSM. The state of the ON/OFF switch variable of the motor-control application part is scanned periodically. The speed command that is entered from the high-level application layer is processed by the means of a speed-ramp algorithm. The comparison between the actual speed command (obtained from the ramp algorithm output) and the measured speed generates a speed error. The speed error is input to the speed PI controller, generating a new desired level of reference for the torque-producing component of the stator current.

The DC-bus voltage and phase currents are sampled by the ADC. The ADC sampling is started by the PWM module trigger signals. A digital filter is applied to the sampled values of the DC-bus voltage. The phase currents are used unfiltered. The 3-phase motor current is reconstructed from two samples taken from the inverter's shunt resistors. The reconstructed 3-phase current is then transformed into the space vectors and used by the FOC algorithm.

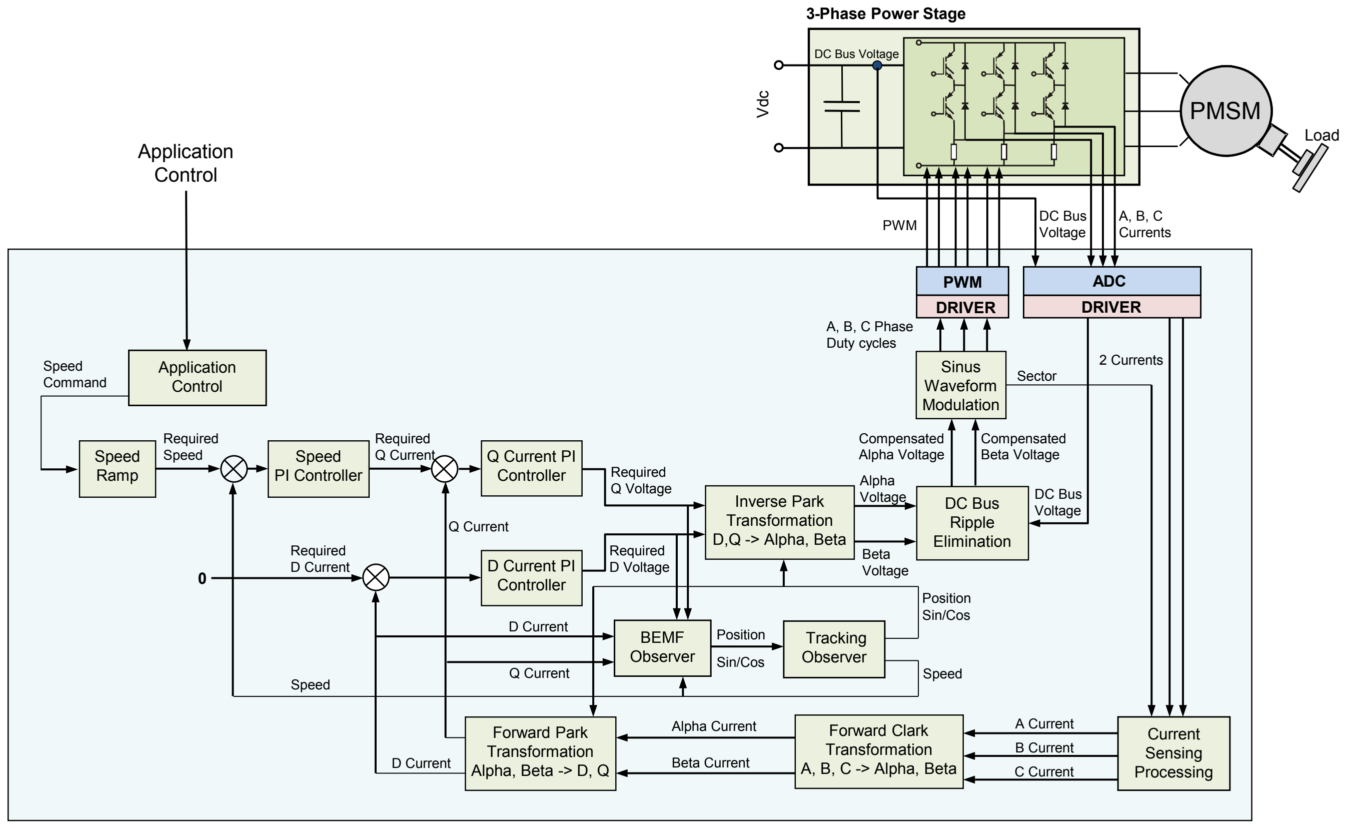

Figure 13. Sensorless PMSM field-oriented control algorithm blocks

The rotor position and speed are provided by the BEMF observer and the tracking observer.

The BEMF observer calculates the error of the angle using the d/q voltage, d/q current, and speed. The error angle of the BEMF is fed to the tracking observer, which returns the position and speed. Based on the measured feedback signals, the FOC algorithm performs a vector-control technique, as described in Section 2.3.2, "FOC algorithm".

Two independent current PI controllers are executed to achieve the desired behavior of the motor. The output from the FOC is a stator-voltage space vector, which is transformed into PWM signals using space vector modulation. The 3-phase stator voltage is generated using a 3-phase voltage source inverter and applied to the motor, which is connected to the power stage terminals.

3.4. Synchronizing motor-control algorithms

The motor-control algorithms are synchronized with the PWM module. The motor-control algorithms can be calculated on every nth PWM reload. This is useful when the CPU is running out of its computing power.

Figure 14 shows the synchronization of the PWM signals. You can observe the top and bottom MOSFET PWM motor signals, the points where the desired quantities are measured, and the points where the algorithm is calculated. The interrupt periods are dependent on hardware settings. For more information, see *Sensorless PMSM Field-Oriented Control on Kinetis KV* (document AN5237).

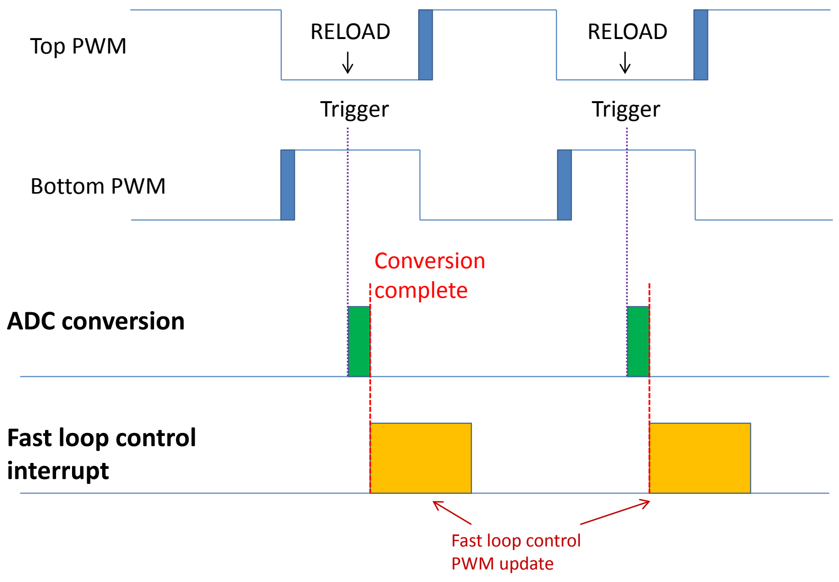

Figure 14. Application timing

3.5. Calling algorithms

This application is interrupt-driven, running in real time. There is one (or more) periodic-interrupt service routine executing the major application-control tasks.

The motor-control fast-loop algorithms are executed periodically in the ADC conversion complete interrupt. This interrupt has the highest priority level. When the interrupt is entered, the DC-bus voltage and motor phase currents are read and the PMSM FOC algorithm is calculated.

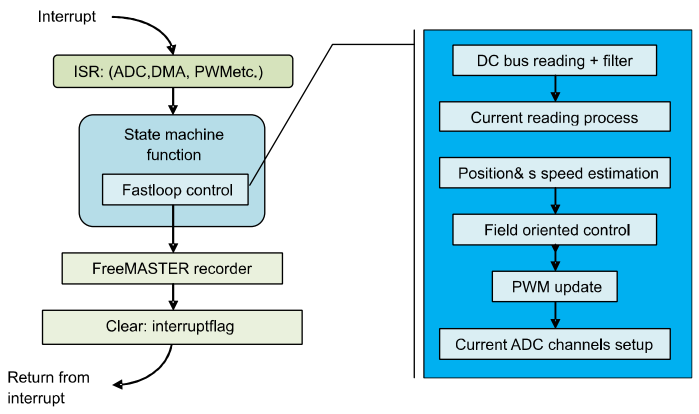

Figure 15. Fast-loop ISR

The slow-loop interrupt service routine is executed periodically when the independent FlexTimer reloads. It is basically a speed control.

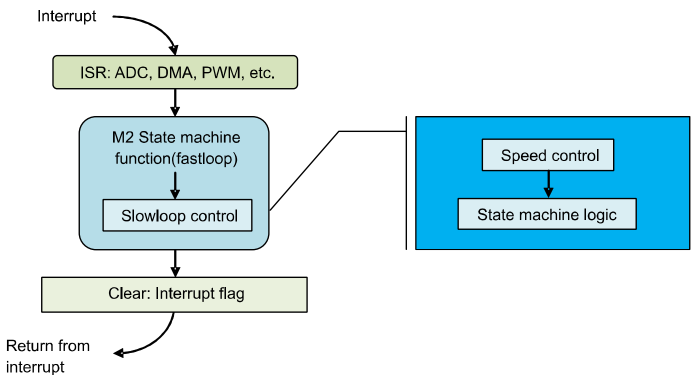

Figure 16. Slow-loop ISR

The background loop is executed in the main application. It handles non-critical timing tasks, such as the communication with FreeMASTER over UART.

The interrupt service routine calls the state machine function. The state machine function depends on the particular state. When in a dependent state, the particular state machine function is called, and it ensures that the correct state-machine logic and control algorithm is called. The state machine is described in the following sections.

3.6. Main state machine

The state-machine structure is unified for all application components and it consists of four main states (see Figure 17). The states are:

- Fault—the system detected a fault condition and waits for the condition to be cleared.
- Init—variables' initialization.
- Stop—the system is initialized and waiting for the Run command.
- Run—the system is running and can be stopped by the Stop command.

There are transition functions between these state functions:

- Init > Stop—initialization was done, the system is entering the Stop state.
- Stop > Run—the Run command was applied, the system is entering the Run state if the Run command was acknowledged.
- Run > Stop—the Stop command was applied, the system is entering the Stop state if the Stop command was acknowledged.
- Fault > Stop—fault flag was cleared, the system is entering the Stop state.
- Init, Stop, Run > Fault—a fault condition occurred, the system is entering the Fault state.

The state machine structure uses these flags to switch between the states:

- SM_CTRL_INIT_DONE—if this flag is set, the system goes from the Init state to the Stop state.
- SM_CTRL_FAULT—if this flag is set, the system goes from any state to the Fault state.
- SM_CTRL_FAULT_CLEAR—if this flag is set, the system goes from the Fault state to the Init state.

- SM_CTRL_START—this flag informs the system that there is a command to go from the Stop state to the Run state. The transition function is called; nevertheless, the action must be acknowledged. The reason is that sometimes it can take time before the system is ready to be switched on.
- SM_CTRL_RUN_ACK—this flag acknowledges that the system can proceed from the Stop state to the Run state.
- SM_CTRL_STOP—this flag informs the system that there is a command to go from the Run state to the Stop state. The transition function is called; nevertheless, the action must be acknowledged. The reason is that the system must be switched off properly, which takes time.
- SM_CTRL_STOP_ACK—this flag acknowledges that the system can proceed from the Run state to the Stop state.

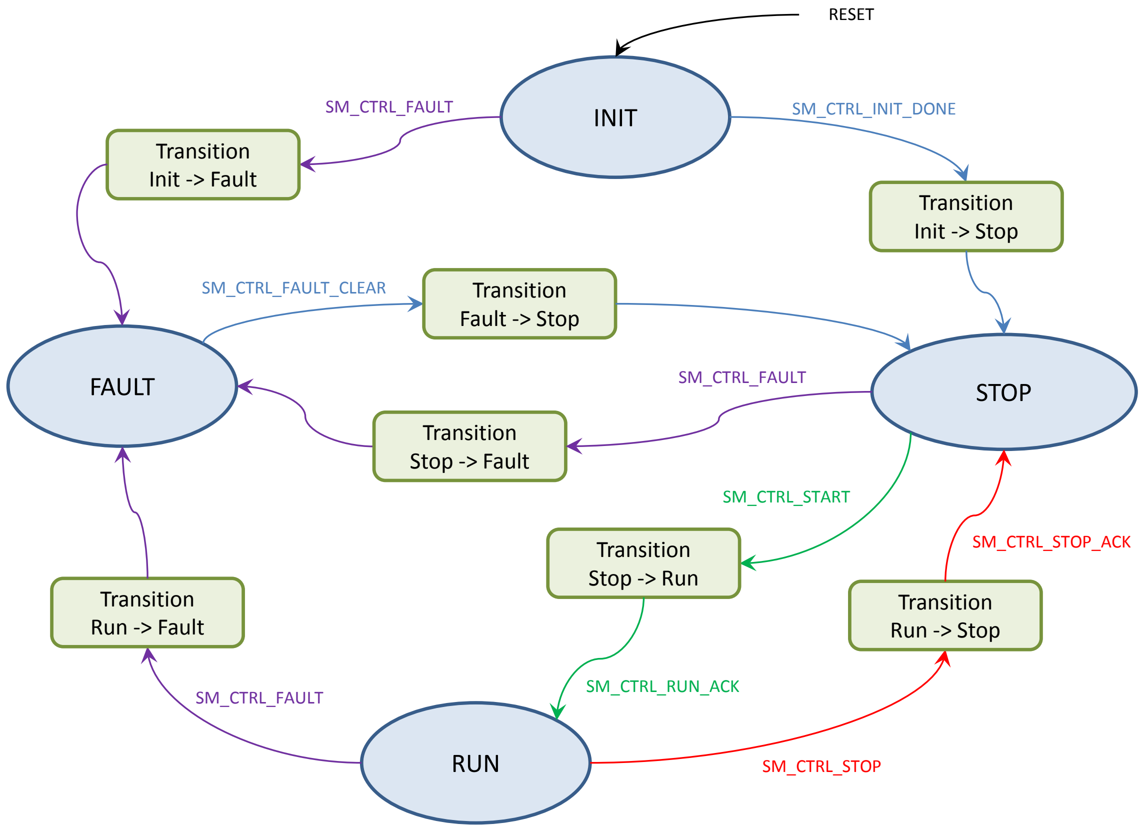

Figure 17. Main state machine diagram

The implementation of this state machine structure is done in the *state_machine.c* and *state_machine.h* files. The following code lines define the state machine structure:

```
/* State machine control structure */
typedef struct
{
    SM_APP_STATE_FCN_T const*     psState;  /* State functions */
    SM_APP_TRANS_FCN_T const*     psTrans;  /* Transition functions */
    SM_APP_CTRL                   uiCtrl;   /* Control flags */
    SM_APP_STATE_T                eState;   /* State */
} SM_APP_CTRL_T;
```

The four variables used in the code are:

- *psState*—pointer to the user state machine functions. The particular state machine function from this table is called when the state machine is in a specific state.
- *psTrans*—pointer to the user transient functions. The particular transient function is called when the system goes from one state to another.
- *uiCtrl*—controls the state machine behavior using the above-mentioned flags.
- *eState*—this variable determines the actual state of the state machine.

The user state machine functions are defined in this structure:

```
/* User state machine functions structure */
typedef struct
{
    PFCN_VOID_VOID       Fault;
    PFCN_VOID_VOID       Init;
    PFCN_VOID_VOID       Stop;
    PFCN_VOID_VOID       Run;
} SM_APP_STATE_FCN_T;
```

The user transient state machine functions are defined in this structure:

```
/* User state-transition functions structure*/
typedef struct
{
    PFCN_VOID_VOID       FaultInit;
    PFCN_VOID_VOID       InitFault;
    PFCN_VOID_VOID       InitStop;
    PFCN_VOID_VOID       StopFault;
    PFCN_VOID_VOID       StopRun;
    PFCN_VOID_VOID       RunFault;
    PFCN_VOID_VOID       RunStop;
} SM_APP_TRANS_FCN_T;
```

The control flag variable has these definitions:

```
typedef unsigned short SM_APP_CTRL;

/* State machine control command flags */
#define SM_CTRL_NONE            0x0
#define SM_CTRL_FAULT           0x1
#define SM_CTRL_FAULT_CLEAR     0x2
#define SM_CTRL_INIT_DONE       0x4
#define SM_CTRL_STOP            0x8
#define SM_CTRL_START           0x10
#define SM_CTRL_STOP_ACK        0x20
#define SM_CTRL_RUN_ACK         0x40
```

The state-identification variable has these definitions:

```
/* Application state identification enum */
typedef enum
{
    FAULT  = 0,
    INIT   = 1,
    STOP   = 2,
    RUN    = 3,
} SM_APP_STATE_T;
```

Call the state machine from the code periodically using the following inline function. This function's input is the pointer to the state machine structure described above. This structure is declared and initialized in the code where the state machine is called:

```
/* State machine function */
static inline void SM_StateMachineFast(SM_APP_CTRL_T *sAppCtrl)
{
        gSM_STATE_TABLE_FAST[sAppCtrl -> eState](sAppCtrl);
}
```

An example of initializing and using the state machine structure is described in the following section.

3.7. Motor state machine

The motor state machines are based on the main state machine structure. The Run state sub-states are added on top of the main structure to control the motors properly. This is the description of the main states' user functions:

- Fault—the system faces a fault condition and waits until the fault flags are cleared. The DC-bus voltage is measured and filtered.
- Init—variables' initialization.
- Stop—the system is initialized and waiting for the Run command. The PWM output is disabled. The DC-bus voltage is measured and filtered.
- Run—the system is running and can be stopped by the Stop command. The Run sub-state functions are called from here.

There are transition functions between these state functions:

- Init > Stop—nothing is processed in this function.
- Stop > Run:
  - The duty cycle is initialized to 50 %.
  - The PWM output is enabled.
  - The current ADC channels are initialized.
  - The Calib sub-state is set as the initial Run sub-state.
- Run > Stop:
  - The Stop command was applied and the system enters the Stop state if the Stop command was acknowledged.
- Fault > Stop
  - All state variables and accumulators are initialized.
- Init, Stop > Fault
  - The PWM output is disabled.
- Run > Fault:
  - Certain current and voltage variables are zeroed.
  - The PWM output is disabled.

The Run sub-states are called when the state machine is in the Run state. The Run sub-state functions are:

- Calib:
  - The current channels' ADC offset calibration.
  - The DC-bus voltage is measured and filtered.
  - The PWM is set to 50 % and its output is enabled.
  - The current channel offsets are measured and filtered.
  - After the calibration time expires, the system is switched to Ready.
- Ready:
  - The PWM is set to 50 % and its output is enabled.
  - The current is measured and the ADC channels are set up.
  - The DC-bus voltage is measured.
  - Certain variables are initialized.
- Align:
  - The current is measured and the ADC channels are set up.
  - The DC-bus voltage is measured and filtered.
  - The rotor-alignment algorithm is called and the PWM is updated.
  - After the alignment time expires, the system is switched to Startup.
- Startup:
  - The current is measured and the ADC channels are set up.
  - The BEMF observer algorithm is called to estimate the speed and position.
  - The FOC algorithm is called and the PWM is updated.
  - The DC-bus voltage is measured and filtered.
  - The open-loop start-up algorithm is called.
  - The estimated speed is filtered (in case the filter is necessary).
  - If the startup is successful, the system is switched to the Spin state; otherwise, it is switched to the Freewheel state.
- Spin:
  - The current is measured and the ADC channels are set up.
  - The BEMF observer algorithm is called to estimate the speed and position.
  - The FOC algorithm is called and the PWM is updated.
  - The motor starts spinning.
  - The DC-bus voltage is measured and filtered.
  - The estimated speed is filtered (in case the filter is necessary).
  - The speed ramp and the speed PI controller algorithms are called.

- Freewheel:
  - The PWM output is disabled and the module is set to 50 %.
  - The current is measured and the ADC channels are set up.
  - The DC-bus voltage is measured and filtered.
  - The system waits in this sub-state for a certain time, which is generated due to the rotor inertia. That means it waits until the rotor stops.
  - The system evaluates the conditions and enters either the Align state or the Ready state.

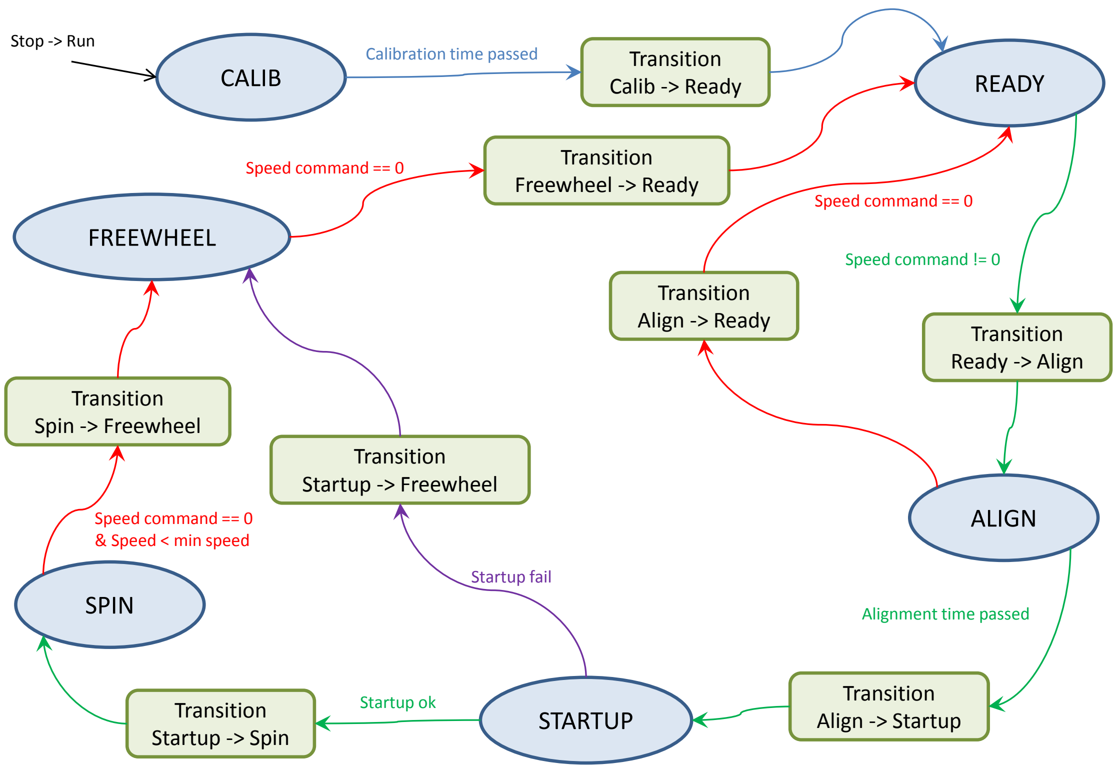
Stop -> Run
CALIB
Calibration time passed
Transition Calib -> Ready
READY
Speed command == 0
Transition Freewheel -> Ready
FREEWHEEL
Speed command == 0
Speed command != 0
Transition Align -> Ready
Transition Ready -> Align
Transition Spin -> Freewheel
Transition Startup -> Freewheel
ALIGN
Speed command == 0 & Speed < min speed
SPIN
Startup fail
Alignment time passed
Transition Startup -> Spin
Startup ok
STARTUP
Transition Align -> Startup

Figure 18. Motor Run sub-state diagram

The Run sub-states also have transition functions that are called in between the sub-states’ transitions. The sub-state transition functions are:

- Calib > Ready—calibration done; entering the Ready state.
- Ready > Align—non-zero speed command; entering the Align state.
  - Certain variables are initialized (voltage, speed, and position).
  - The startup counter is set to 1.
  - The alignment time is set up.
- Align > Ready—zero-speed command; entering the Ready state.
  - Certain voltage and current variables are zeroed.
  - The PWM is set to 50 %.

- Align > Startup—alignment is done; entering the Startup state.
  - The filters and control variables are initialized.
  - The PWM is set to 50 %.
- Startup > Spin—startup is successful; entering the Spin state.
- Startup > Freewheel.
  - Certain variables are initialized (voltage, speed, and position).
  - The startup counter is set to 1.
  - The Freewheel time is set up.
- Spin > Freewheel— zero speed command; entering the Freewheel state.
  - Certain variables are initialized (voltage, speed, and position).
  - The startup counter is set to 1.
  - The Freewheel time is set up.
- Freewheel > Ready—freewheel time elapsed; entering the Ready state.
  - The PWM output is enabled.

The implementation of this motor state-machine structure is done in the *M1_statemachine.c* and *M1_statemachine.h* files. Each state is doubled. States with the *Fast* suffix are called in the fast loop and states with the *Slow* suffix are called in the slow loop. However, the state transitions are not doubled and the state machine flow is controlled from the fast-loop state machine only. Here is the description of the main state-machine structure:

- The main states' user function prototypes are:

```
static void M1_StateFault(void);
static void M1_StateInit(void);
static void M1_StateStop(void);
static void M1_StateRun(void);
```

- The main states' user transient function prototypes are:

```
static void M1_TransFaultStop(void);
static void M1_TransInitFault(void);
static void M1_TransInitStop(void);
static void M1_TransStopFault(void);
static void M1_TransStopRun(void);
static void M1_TransRunFault(void);
static void M1_TransRunStop(void);
```

- The main states functions table initialization is:

```
/* State machine functions field */
static const SM_APP_STATE_FCN_T msSTATE = {M1_StateFault, M1_StateInit,
M1_StateStop, M1_StateRun};
```

- The main state transient functions table initialization is:

```
/* State-transition functions field */
static const SM_APP_TRANS_FCN_T msTRANS = {M1_TransFaultStop, M1_TransInitFault,
M1_TransInitStop, M1_TransStopFault, M1_TransStopRun, M1_TransRunFault,
M1_TransRunStop};
```

- Finally, the main state machine structure initialization is:

```
/* State machine structure declaration and initialization */
SM_APP_CTRL_T gsM1_Ctrl =
{
   /* gsM1_Ctrl.psState, User state functions */
   &msSTATE,

   /* gsM1_Ctrl.psTrans, User state-transition functions */
   &msTRANS,

   /* gsM1_Ctrl.uiCtrl, Default no control command */
   SM_CTRL_NONE,

   /* gsM1_Ctrl.eState, Default state after reset */
   INIT
};
```

- Similarly, the Run sub-state machine is declared. Thus, the Run sub-state identification variable has these definitions:

```
typedef enum {
    M1_CALIB                = 0,
    M1_READY                = 1,
    M1_ALIGN                = 2,
    M1_STARTUP              = 3,
    M1_SPIN                 = 4,
    M1_FREEWHEEL            = 5
} M1_RUN_SUBSTATE_T;          /* Run sub-states */
```

- For the Run sub-states, two sets of user functions are defined. The user function prototypes of the Run sub-states are:

```
static void M1_StateRunCalib(void);
static void M1_StateRunReady(void);
static void M1_StateRunAlign(void);
static void M1_StateRunStartup(void);
static void M1_StateRunSpin(void);
static void M1_StateRunFreewheel(void);
```

- The Run sub-states’ user transient function prototypes are:

```
static void M1_TransRunCalibReady(void);
static void M1_TransRunReadyAlign(void);
static void M1_TransRunAlignStartup(void);
static void M1_TransRunAlignReady(void);
static void M1_TransRunStartupSpin(void);
static void M1_TransRunStartupFreewheel(void);
static void M1_TransRunSpinFreewheel(void);
static void M1_TransRunFreewheelReady(void);
```

- The Run sub-states functions table initialization is:

```
/* Sub-state machine functions field */
static const PFCN_VOID_VOID mM1_STATE_RUN_TABLE[6] = {
M1_StateRunCalib,
M1_StateRunReady,
M1_StateRunAlign,
M1_StateRunStartup,
M1_StateRunSpin,
M1_StateRunFreewheel
};
```

- The state machine is called from the interrupt service routine, as mentioned in Section 3.5, "Calling algorithms". The code syntax used for calling the state machine is:

```
/* StateMachine call */
SM_StateMachine(&gsM1_Ctrl);
```

- Inside the user Run state function, the sub-state functions are called as:

```
/* Run sub-state function */
mM1_STATE_RUN_TABLE[meM1_StateRun]();
```

  where the variable identifies the Run sub-state.

3.8. Sensorless PMSM control

The application controls the PMSM in the sensorless mode. It is possible to control more PMSMs using one controller (if the MCU enables it). To avoid the multiplicity of routines in the memory, the application uses a set of routines where the inputs are the structures of the particular motors. This approach saves the necessary program flash memory in the application.

It is necessary to understand the fast-loop control loop implementation as a whole, and then break it down to pieces.

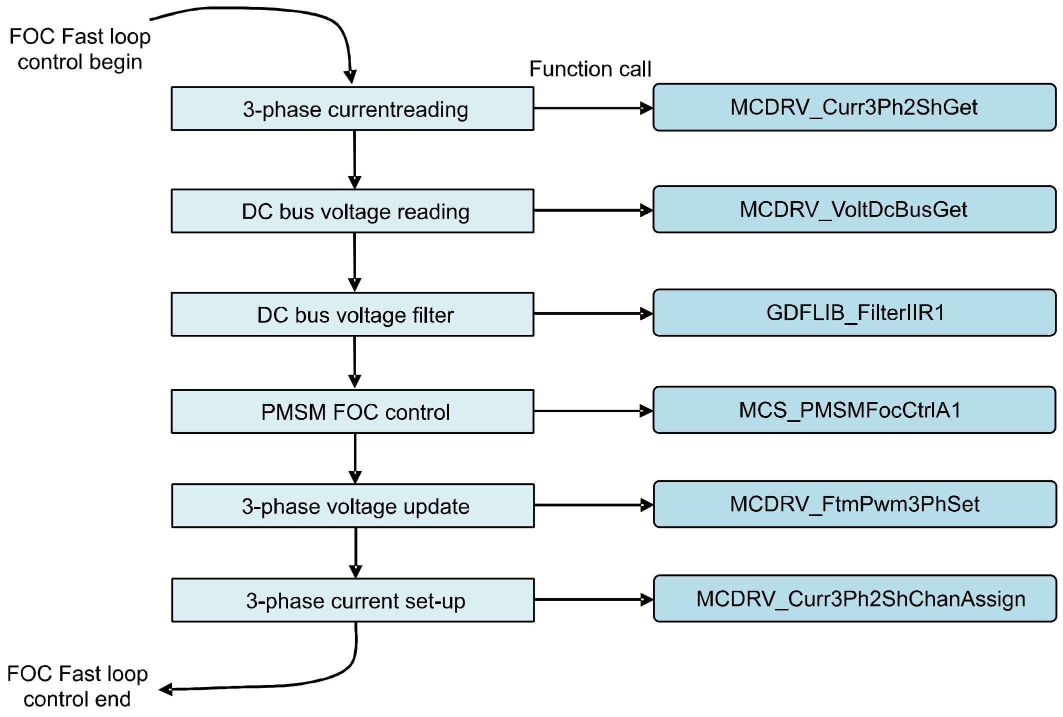

Figure 19. Fast-loop control flowchart

Figure 13 shows that the application requires sensors and estimators (current, DC-bus voltage, estimated speed, etc.) and actuators (PWM, etc.) to observe and control the whole system. The sensors and actuators are encapsulated into the dedicated modules within the unified interface called MCDRV. For more information on MCDRV, see *Sensorless PMSM Field-Oriented Control on Kinetis KV* (document AN5237).

3.8.1. MCS—field-oriented control

There is a single function executing the current FOC or the voltage FOC. The current FOC means that the inputs to this function are the required D and Q currents, and these currents are controlled to the required values by varying the D and Q voltages applied to the motor. The voltage control mode means that the input to this function are the required D and Q voltages that are applied to the motor. Choose between the current/voltage control using the `bCurrentLoopOn` flag. Example 1 belongs to the code implementation.

This FOC mode is optimized into one function which has one input/output pointer to a structure. The prototype of the function is:

```
void MCS_PMSMFocCtrlA1(MCS_PMSM_FOC_A1_T *psFocPMSM)
```

The structure referred to by the input/output structure pointer is defined in this example:

Example 1. Field-oriented current control code

```
typedef struct
{
    GFLIB_CTRL_PI_P_AW_T_A32    sIdPiParams;        /* Id PI controller parameters */
    GFLIB_CTRL_PI_P_AW_T_A32    sIqPiParams;        /* Iq PI controller parameters */
    GDFLIB_FILTER_IIR1_T_F32    sUDcBusFilter;      /* Dc bus voltage filter */
    GMCLIB_3COOR_T_F16          sIABC;              /* Measured 3-phase current */
    GMCLIB_2COOR_ALBE_T_F16     sIAlBe;             /* Alpha/Beta current */
    GMCLIB_2COOR_DQ_T_F16       sIDQ;               /* DQ current */
    GMCLIB_2COOR_DQ_T_F16       sIDQReq;            /* DQ required current */
    GMCLIB_2COOR_DQ_T_F16       sIDQError;          /* DQ current error */
    GMCLIB_3COOR_T_F16          sDutyABC;           /* Applied duty cycles ABC */
    GMCLIB_2COOR_ALBE_T_F16     sUAlBeReq;          /* Required Alpha/Beta voltage */
    GMCLIB_2COOR_ALBE_T_F16     sUAlBeComp;         /* Compensated to DC bus Alpha/Beta voltage
    GMCLIB_2COOR_DQ_T_F16       sUDQReq;            /* Required DQ voltage */
    GMCLIB_2COOR_DQ_T_F16       sUDQEst;            /* BEMF observer input DQ voltages */
    GMCLIB_2COOR_SINCOS_T_F16   sAnglePosEl;        /* Electrical position sin/cos (at the
       moment of PWM current reading) */
    AMCLIB_BEMF_OBSRV_DQ_T_A32 sBemfObsrv;          /* BEMF observer in DQ */
    AMCLIB_TRACK_OBSRV_T_F32    sTo;                /* Tracking observer */
    GDFLIB_FILTER_IIR1_T_F32    sSpeedElEstFilt;    /* Estimated speed filter */
    frac16_t                    f16SpeedElEst;      /* Rotor electrical speed estimated */
    uint16_t                    ui16SectorSVM;      /* SVM sector */
    frac16_t                    f16PosEl;           /* Electrical position */
    frac16_t                    f16PosElExt;        /* Electrical position set from external
       function - sensor, open loop */
    frac16_t                    f16PosElEst;        /* Rotor electrical position estimated*/
    frac16_t                    f16DutyCycleLimit;  /* Max. allowable duty cycle in frac */
    frac16_t                    f16UDcBus;          /* DC bus voltage */
    frac16_t                    f16UDcBusFilt;      /* Filtered DC bus voltage */
    bool_t                      bCurrentLoopOn;     /* Flag enabling calculation of current
       control loop */
    bool_t                      bPosExtOn;          /* Flag enabling use of electrical position
       passed from other functions */
    bool_t                      bOpenLoop;          /* Position estimation loop is open */
    bool_t                      bIdPiStopInteg;     /* Id PI controller manual stop integration
    bool_t                      bIqPiStopInteg;     /* Iq PI controller manual stop integration
} MCS_PMSM_FOC_A1_T;
```

This structure contains all the necessary variables or sub-structures for the field-oriented control algorithm implementation. The types used in this structure are defined in the Embedded Software Libraries (FSLESL). Here is a description of the items used in this application:

- D and Q current PI controllers—serve to control the D and Q currents.
- DC-bus voltage first-order IIR filter—serves to filter the DC-bus voltage.
- A, B, and C currents—measured 3-phase current; input to the algorithm.
- Alpha and beta currents—currents transformed into the alpha/beta frame.
- D and Q currents—currents transformed into the D/Q frame.
- Required D and Q currents—required currents in the D/Q frame; input to the algorithm.
- D and Q current error—error (difference) between the required and the measured D/Q currents.
- A, B, and C duty cycles—3-phase duty cycles; output from the algorithm.
- Required alpha and beta voltages—required voltages in the alpha/beta frame.
- Compensated required alpha and beta voltages—the previous item recalculated on the actual level of the DC-bus voltage.
- Required D and Q voltages—required voltages in the alpha/beta frame; outputs from the PI controllers.
- Sin/Cos angle—rotor estimated electrical angle (sine, cosine).
- Duty cycle limit—this variable limits the maximum value of the A, B, and C duty cycles.
- DC-bus voltage—measured DC-bus voltage.
- Filtered DC-bus voltage—filtered value of the previous item.
- SVM sector—sector information; output from the SVM algorithm.
- D current saturation flag—saturation flag for the D-current PI controller.
- Q current saturation flag—saturation flag for the Q-current PI controller.
- Speed startup flag—indicates that the motor is in startup.

This routine calculates the field-oriented control. The inputs to this routine include the 3-phase current, DC-bus voltage, the electrical position, and the required D and Q currents. The output from this routine are the 3-phase duty cycle, SVM sector, and the saturation flags of the PI controllers. The PI controllers and filters have structures that must be initialized before using this routine.

This routine is called in the fast loop state machine and its process flow chart is shown in the below figure. This function uses algorithms from the Embedded Software Libraries (FSLESL); the functions' names may differ from one platform to another.

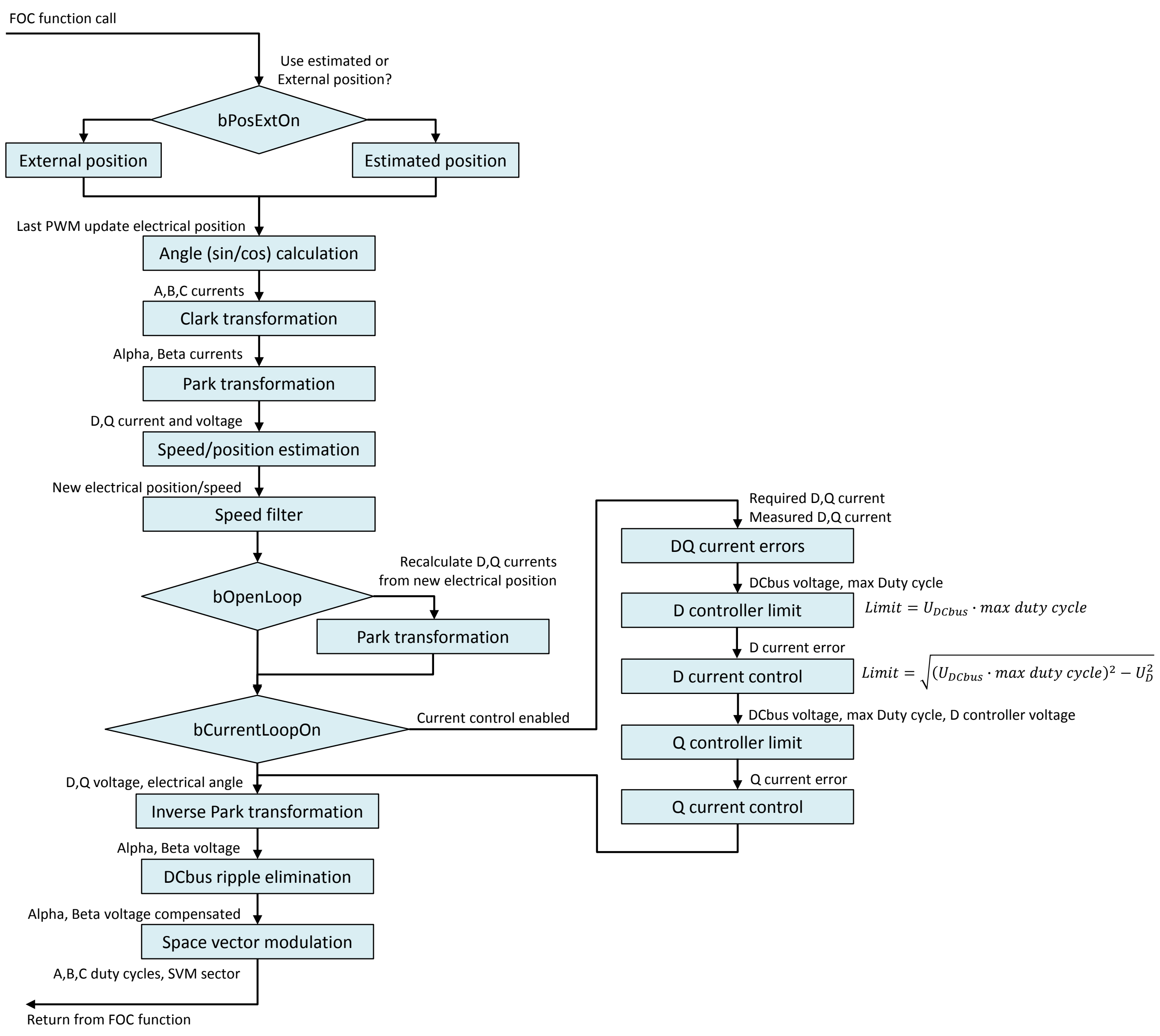
FOC function call
Use estimated or External position?
bPosExtOn
External position
Estimated position
Last PWM update electrical position
Angle (sin/cos) calculation
A,B,C currents
Clark transformation
Alpha, Beta currents
Park transformation
D,Q current and voltage
Speed/position estimation
New electrical position/speed
Speed filter
Recalculate D,Q currents from new electrical position
bOpenLoop
Park transformation
bCurrentLoopOn
Current control enabled
Required D,Q current
Measured D,Q current
DQ current errors
DCbus voltage, max Duty cycle
D controller limit
$Limit = U_{DCbus} \cdot max\ duty\ cycle$
D current error
D current control
$Limit = \sqrt{(U_{DCbus} \cdot max\ duty\ cycle)^2 - U_D^2}$
DCbus voltage, max Duty cycle, D controller voltage
Q controller limit
Q current error
Q current control
D,Q voltage, electrical angle
Inverse Park transformation
Alpha, Beta voltage
DCbus ripple elimination
Alpha, Beta voltage compensated
Space vector modulation
A,B,C duty cycles, SVM sector
Return from FOC function

Figure 20. FOC flowchart

3.8.2. MCS—scalar control

The scalar control (V/Hz) is another control function. It is recommended to run the application in this mode only while the application is mastered using MCAT tool, because this tool automatically calculates the voltage/frequency ratio based on the motor parameters. Similar to the previous algorithms, this control mode is optimized into one function which has two input/output pointers to a structure. The prototype of this function is:

```
void MCS_PMSMScalarCtrlA1(MCS_PMSM_FOC_A1_T *psFocPMSM, MCS_SCALAR_A1_T *psScalarPMSM)
```

The first structure referred to by the input/output structure pointer is the same as for the current control mode described in Section 3.8.1, “MCS—field-oriented control”. The second structure has the proper scalar control variables defined in Example 2.

Example 2. Scalar control code

```
typedef struct
{
    GFLIB_RAMP_T_F32        sFreqRampParams;   /* Parameters of frequency ramp */
    GMCLIB_2COOR_DQ_T_F16   sUDQReq;           /* Required voltage vector in d,q coordinates */
    GFLIB_INTEGRATOR_T_A32  sFreqIntegrator;   /* integrator parameters (integrates the omega
                                                  to get the position */
    frac16_t                f16FreqCmd;        /* required electrical frequency from master
                                                  system */
    frac16_t                f16FreqRamp;       /* Required frequency limited by ramp - the ramp
                                                  output */
    frac16_t                f16PosElScalar;    /* Electrical angle of the rotor */
    frac16_t                f16VHzGain;        /* VHz_factor constant gain for scalar control
    int16_t                 f16VHzGainShift;   /* VHz_factor constant shift for scalar control
} MCS_PMSM_SCALAR_CTRL_A1_T;
```

This structure contains the additional necessary variables or sub-structures for the scalar control algorithm implementation. The types used in this structure are defined in the Embedded Software Libraries (FSLESL). Here is a description of the items used in this application:

- Frequency ramp—the parameters for the frequency ramp generation.
- Required D and Q voltages—the required voltages in the alpha/beta frame; outputs from the V/Hz algorithm.
- Frequency integrator—the parameters for the frequency integrator, where its output is the position.
- Frequency command—the required frequency commanded by the user application.
- Ramped frequency—the output from the frequency ramp algorithm.
- Electrical position—the electrical position of the rotor.
- V/Hz gain—the factor to calculate the required Q voltage based on the frequency.
- V/Hz gain shift—the shift of the factor to calculate the required Q voltage based on frequency.

This routine calculates the Q voltage based on the actual electrical rotor frequency. The inputs to this routine include the required speed and the V/rpm factor. The outputs of this routine are the 3-phase duty cycle and the SVM sector. The filters have structures that must be initialized before using this routine.

This routine is called in the fast-loop state machine and its process flow chart is shown in Figure 21. This function reuses the *MCS_PMSMFocCtrlA1* described in Section 3.8.1, “MCS—field-oriented control”. This function uses algorithms from the Embedded Software Libraries (FSLESL); the functions’ names may differ from one platform to another.

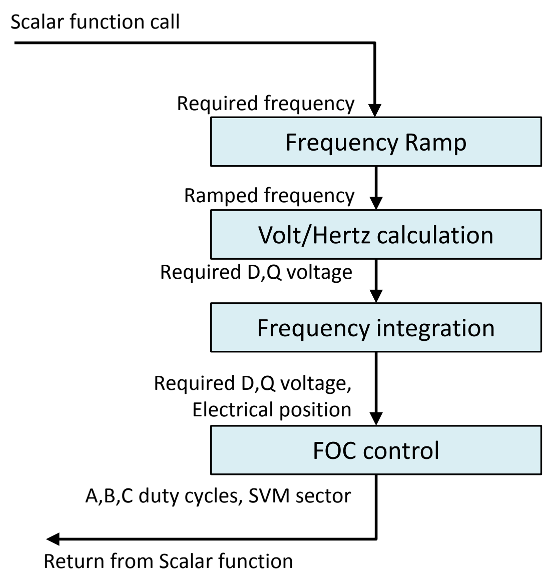

Figure 21. Scalar control flowchart

3.8.3. Rotor alignment

This application performs the rotor alignment before starting the motor. That means the rotor is forced to a known position.

Similarly to the previous algorithms, the alignment is optimized into one function which has one input/output pointer to a structure. The prototype of the function is:

```
void MCS_PMSMAlignmentA1(MCS_PMSM_FOC_A1_T *psFocPMSM, MCS_ALIGNMENT_A1_T *psAlignment)
```

This function uses the FOC structure described in Section 3.8.1, “MCS—field-oriented control”. The alignment variables are contained in the alignment structure referred to by the second pointer. The alignment structure definition is:

Example 3. Alignment code

```
typedef struct
{
    frac16_t                    f16UdReq;            /* Required D voltage at alignment */
    uint16_t                    ui16Time;            /* Alignment time duration */
    uint16_t                    ui16TimeHalf;        /* Alignment half time duration */
}MCS_ALIGNMENT_A1_T;           /* PMSM simple two-step Ud voltage alignment */
```

The structure contains the variables necessary to perform the rotor alignment. The structure is described as follows:

- D voltage—the required voltage applied in the D axis during the alignment.
- Alignment duration—the duration of the alignment.
- Alignment halftime—the time for which the 120-degree position is required; when this time passes, the 0-degree position is applied.

The routine applies the D voltage defined in the alignment structure. This voltage command is applied during the alignment. The alignment duration is split into two segments:

1. The angle of 120 degrees is forced (from the beginning of the alignment until the halftime).
2. The angle of 0 degrees is forced (from the alignment halftime to the end).

This function calls the *MCS_PMSMFocCtrlA1* function. The alignment algorithm flowchart is shown in this figure:

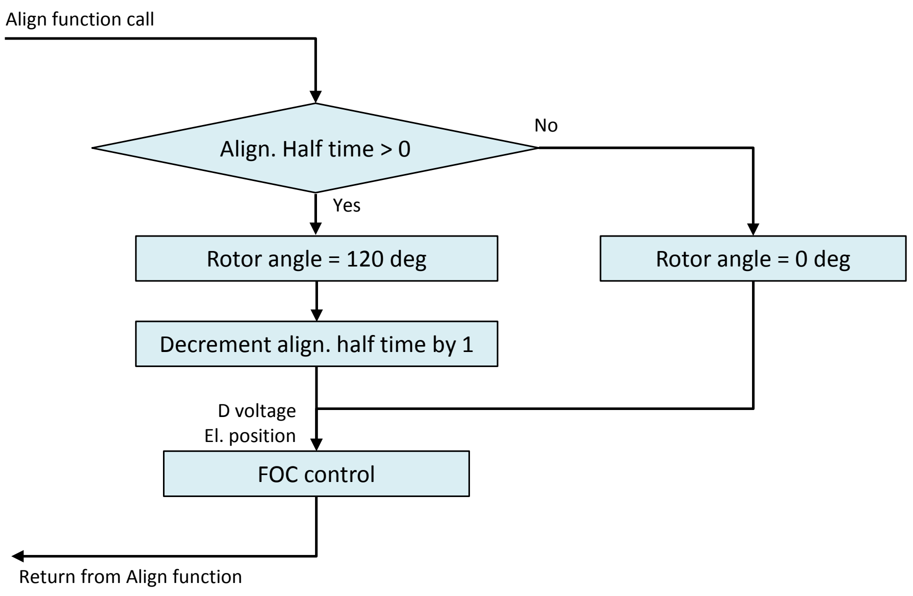

Figure 22. Rotor alignment flowchart

3.8.4. Motor open-loop startup

Because the BEMF observer does not provide reliable feedback at very low speeds, the motor must be pushed to a certain speed in the open-loop mode. The way of the startup expects similar start-up conditions for each startup. This method consists of a generated rotating field with the Q-current profile that spins the rotor according to the generated speed.

Similar to the previous algorithms, the open-loop startup is optimized into one function which has one input/output pointer to a structure. The prototype of the function is:

```
void MCS_PMSMOpenLoopStartUpA1(MCS_STARTUP_A1_T *psStartUp)
```

The function uses its own start-up structure that is referred to by the input/output structure pointers. Its definition is:

Example 4. Scalar control code

```
typedef struct
{
    GFLIB_INTEGRATOR_T_A32 sSpeedIntegrator;          /* Speed integrator structure*/
    GFLIB_RAMP_T_F32       sSpeedRampOpenLoopParams;  /* Parameters of startup speed ramp */
    frac16_t               f16SpeedReq;               /* Required speed */
    frac16_t               f16PosEst;                 /* Fractional electrical position */
    frac16_t               f16SpeedRampOpenLoop;      /* Open loop startup speed ramp */
    frac16_t                f16CoeffMerging;            /* increment of merging weight for
       position merging */
    frac16_t                f16RatioMerging;            /* merging weight coefficient */
    frac16_t                f16PosGen;                  /* generated open loop position from
       the speed ramp integration */
    frac16_t                f16PosMerged;               /* merged position */
    frac16_t                f16SpeedCatchUp;            /* merging speed threshold */
    frac16_t                f16CurrentStartup;          /* required Iq current during open
       loop start up */
    uint16_t                ui16TimeStartUpFreeWheel;   /* Free-wheel duration if start-up
       aborted by user input) */
    bool_t                  bOpenLoop;                  /* Position estimation loop is open */
} MCS_PMSM_STARTUP_A1_T;                  /* simple open-loop start-up with merged position and
constant start-up current */
```

The structure contains the variables necessary to perform the open-loop startup. The start-up procedure is shown in Figure 23. The structure is described as follows:

- Speed integration structure—integrates the speed resulting the position in the correct position scale.
- Speed ramp structure—generates the speed ramp during the open-loop startup.
- Required speed—the required speed set by the user.
- Open-loop mode speed ramp output—the speed generated from the ramp during the open-loop mode.
- Estimated electrical position—the electrical position estimated by the BEMF observer.
- Merging coefficient—defines the duration increment of the position-merging process.
- Merging ratio— the ratio that determines how much of the generated and the estimated position information is used as the merging output.
- Open-loop mode position—the position generated as the integration of the speed ramp output during the open-loop mode.
- Merged position—the generated and estimated position output of the merging process.
- Catch-up speed—determines where to start the merging of the generated open-loop and BEMF observer estimated position.
- Start-up current—the Q current applied to start the motor.
- Start-up freewheel time—the duration applied due to the rotor inertia after a start-up process is terminated by the user (entering zero-speed command during the initiated start-up process).
- Open-loop mode—the switch that determines whether the observer feedback is in the open-loop mode or the closed-loop mode.

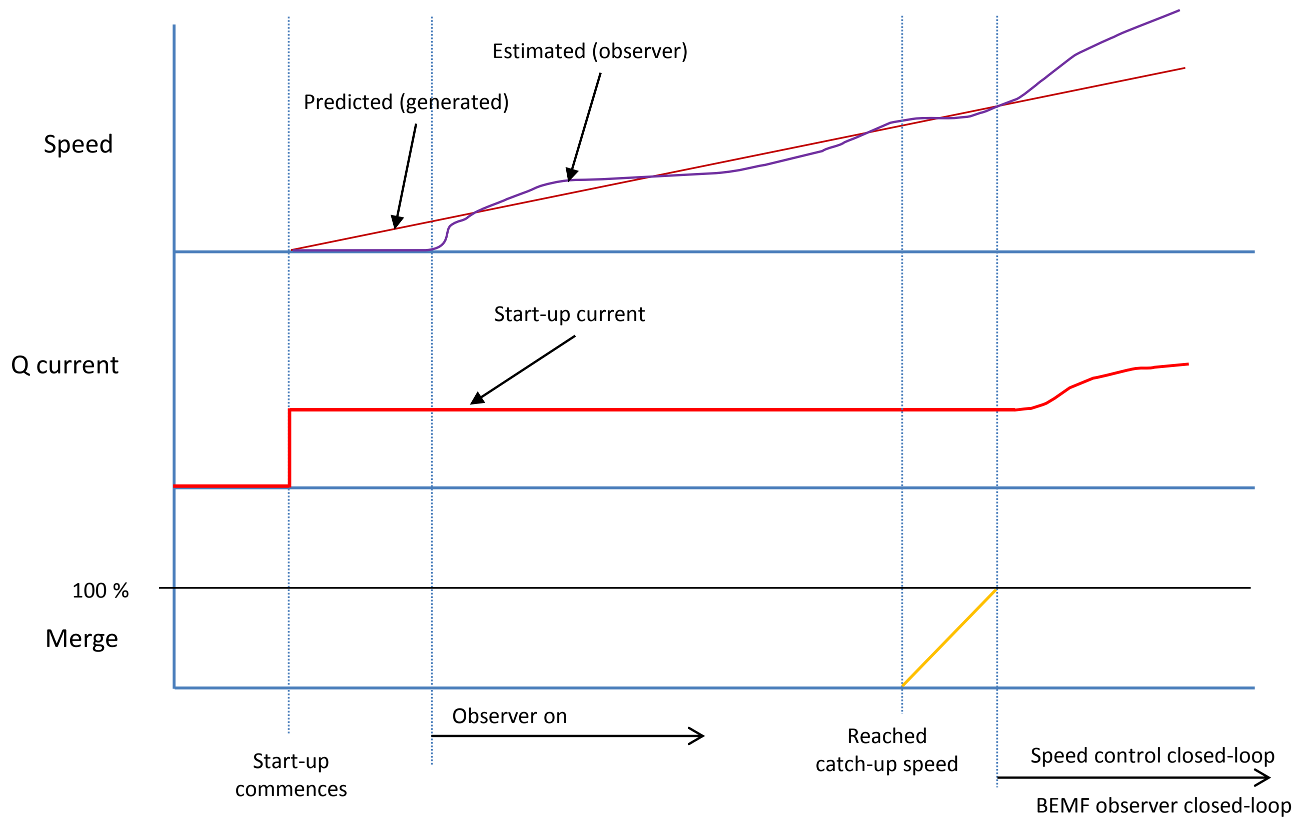

Figure 23. Motor open-loop startup

The start-up procedure is a complex process which is based on applying the torque and expecting the speed to respond accordingly. Therefore, the torque is proportional to the Q current which is kept constant.

The speed is ramped with the constant acceleration while the Q current is kept at the defined level. When the catch-up speed is reached, the merging ratio is incremented with the merging coefficient. This ratio determines how much of the estimated position information is merged with the generated (predicted) position information. At the beginning, it is 0 % for the estimated position information and 100 % for the predicted position information. Their sum is always 1.

After the system finishes merging the BEMF position, the loop is closed, and the motor is successfully started with the speed control turned on.

3.8.5. Speed loop

The speed loop is calculated in the Run > Spin sub-state (in the *M1_StateRunSpin* function). The slow-loop frequency is divided to reach a lower frequency of control.

In this application, the slow-loop control is very simple. See the following figure.

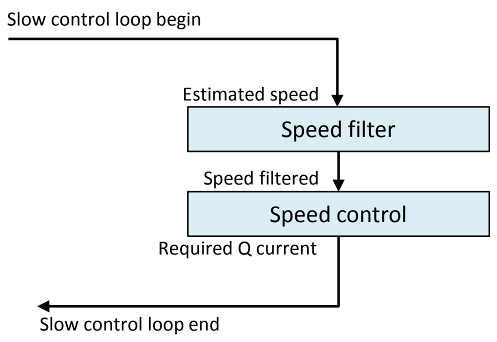

Figure 24. Slow-loop control flowchart

The structure used in the speed control loop is:

Example 5. Speed control code

```
typedef struct
{
    GDFLIB_FILTER_IIR1_T_F32 sSpeedFilter;     /* Speed filter */
    GFLIB_CTRL_PI_P_AW_T_A32 sSpeedPiParams;   /* Speed PI controller parameters */
    GFLIB_RAMP_T_F32         sSpeedRampParams; /* Speed ramp parameters */
    frac16_t                 f16Speed;         /* Speed */
    frac16_t                 f16SpeedFilt;     /* Speed filtered */
    frac16_t                 f16SpeedError;    /* Speed error */
    frac16_t                 f16SpeedRamp;     /* Required speed (ramp output) */
    frac16_t                 f16SpeedCmd;      /* Speed command (required speed)*/
    bool_t                   bSpeedPiSatFlag;  /* Speed PI saturation flag */
} MCS_SPEED_A1_T;
```

The speed loop contains the structure of the variables necessary to perform the speed control. The structure is described as follows:

- Speed first-order IIR filter—filters the speed.
- Speed PI controller structure—controls the speed.
- Speed ramp structure—generates the speed ramp.
- Speed—the speed of the motor.
- Filtered speed—the filtered speed of the motor.
- Speed error—the error between the required speed and the measured speed.
- Ramped speed—the speed ramp algorithm output.
- Required speed—the speed input to the ramp algorithm.
- Speed command—the speed requested by the user or a higher layer of the application.
- Speed PI controller saturation flag—determines whether the PI controller is in saturation.

The measured speed can use the filter that filters out the noise. In that case, the *MCS_PMSMFocCtrlSpeedA1* speed-control function is called.

This function performs these tasks (see Figure 25):

- Generation of the saturation flag based on the Q current and speed PI controller saturation flag, but only if the speed request is higher than the actual speed.
- Speed ramp generation. This ramp has an input of the previously mentioned saturation flag. That means the ramp is generated towards the requested speed; when the saturation flag is set, the ramp returns slowly (with a different increment) towards the actual speed to get the system out of saturation.
- Speed PI controller. This algorithm controls the speed. The input is the error (the difference) between the desired speed and the actual speed. Its output is the desired Q current.

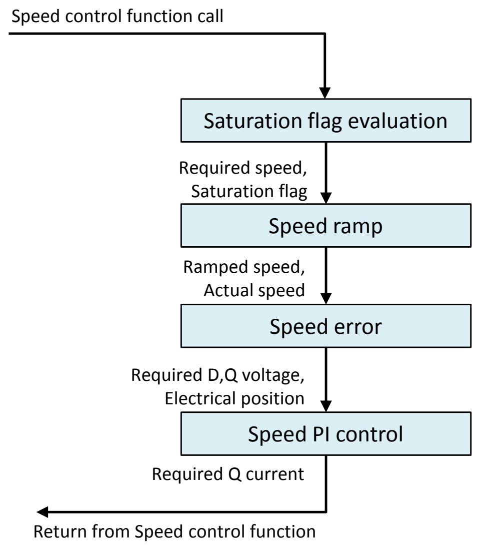

Figure 25. Speed control flowchart

4. Acronyms and Abbreviations

This table lists the acronyms and abbreviations used in this document:

Table 2. Acronyms and abbreviations

Term	Meaning
AC	Alternating current
ACIM	AC Induction Motor
ADC	Analogue-to-digital converter
FSLESL	Freescale's Embedded Software Libraries
BEMF	Back-electromotive force
DC	Direct current
DRM	Design reference manual
FTM	Flex Timer Module
ISR	Interrupt Service Routine
MCAT	Motor Control Application Tuning
MCDRV	Motor Control Drivers
MCU	Microcontroller Unit
PMSM	Permanent Magnet Synchronous Motor
PWM	Pulse-width modulation
SC	Scalar Control
SVM	Space Vector Modulation
VC	Vector Control
XBAR	Crossbar unit

5. List of Symbols

This table lists the symbols used in this document:

Table 3. List of symbols

Symbol	Definition
d,q	Rotational orthogonal coordinate system
u_γ, u_δ	Alpha/Beta BEMF observer error
i_{sa}, i_{sb}, i_{sc}	Stator currents of the a, b, and c phases
$i_{sd,q}$, $i_s(d,q)$	Stator currents in the d, q coordinate system
$i_{\gamma,\delta}$	Stator currents in estimated d, q coordinate system
$i_{s\alpha,\beta}$, $i_s(\alpha,\beta)$	Stator currents in α, β coordinate system
$\hat{i}_{sg}$	Stator current space vector in general reference frame
i_{sx}, i_{sy}	Stator current space vector components in general reference frame
$\hat{i}_{rg}$	Rotor current space vector in general reference frame
i_{rx}, i_{ry}	Rotor current space vector components in general reference frame
J	Mechanical inertia
K_M	Motor constant
k_e	BEMF constant
L_s	Stator-phase inductance
L_{sd}, L_D	Stator-phase inductance d axis
L_{sq}, L_Q	Stator-phase inductance q axis
p	Number of poles per phase
R_s	Stator-phase resistance
s	Derivative operator
T_L	Load torque
$u_{S\alpha,\beta}$,$u_{S(\alpha,\beta)}$	Stator voltages in α, β coordinate system
$u_{Sd,q}$, $u_{S(d,q)}$	Stator voltages in d, q coordinate system
$\Psi_{S\alpha,\beta}$	Stator magnetic fluxes in α, β coordinate system
$\Psi_{Sd,q}$	Stator magnetic fluxes in d, q coordinate system
Ψ_M	Rotor magnetic flux
θ_r	Rotor position angle in α, β coordinate system
ω, ω_r, ω_e	Electrical rotor angular speed / fields angular speed

6. Revision History

This table summarizes the changes made to this document since the initial release.

Table 4. Revision history

Revision number	Date	Substantive changes
0	04/2014	Initial release.
1	02/2016	Updated the document to correspond to the new software structure. Updated the state machine description. Added Section 4, “Acronyms and Abbreviations” and Section 5, “List of Symbols”.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: DRM148
Rev. 1
02/2016