NXP Semiconductors
Application Notes

Document Number:AN12235
Rev. 1, 06/2020

3-Phase Sensorless PMSM Motor Control Kit with S32K144

Featuring Motor Control Application Tuning (MCAT) Tool

by: NXP Semiconductors

Contents

1. Introduction

This application note describes the design of a 3-phase Permanent Magnet Synchronous Motor (PMSM) vector control (Field Oriented Control - FOC) drive with 2-shunt current sensing with and without position sensor.

This design serves as an example of motor control design using S32K1 family of automotive motor control MCUs based on a 32-bit ARM® Cortex™-M4F optimized for a full range of automotive applications.

Following are the supported features:

- 3-phase PMSM speed Field Oriented Control.
- Current sensing with two shunt resistors.
- Shaft position and speed estimated by sensorless algorithm and encoder position sensor
- Application control user interface using FreeMASTER debugging tool.
- Motor Control Application Tuning (MCAT) tool.

2. System concept

The system is designed to drive a 3-phase PM synchronous motor. The application meets the following performance specifications:

- Targeted at the S32K144EVB Evaluation Board (refer to dedicated user manual for S32K144EVB available at www.nxp.com). See section References for more information.
- S32 Software Development Kit (SDK) and Processor Expert (PEx) used as S32K144 device configuration and control tool being a part of the S32 Design Studio (see section References)
- MC34GD3000 MOSFETs pre-driver with extensive set of functions and condition monitoring (see section References)
- Control technique incorporating:
  - Field Oriented Control of 3-phase PM synchronous motor without position sensor
  - Closed-loop speed control with action period 1ms
  - Closed-loop current control with action period 100µs
  - Bi-directional rotation
  - Flux and torque independent control
  - Field weakening control extending speed range of the PMSM beyond the base speed
  - Position and speed is estimated by Extended BEMF observer or obtained by Encoder sensor
  - Open-loop start up with alignment
  - Reconstruction of three-phase motor currents from two shunt resistors
  - FOC state variables sampled with 100 µs period
- Automotive Math and Motor Control Library (AMMCLIB) - FOC algorithm built on blocks of precompiled SW library (see section References)
- FreeMASTER software control interface (motor start/stop, speed setup)
- FreeMASTER software monitor
- FreeMASTER embedded Motor Control Application Tuning (MCAT) tool (motor parameters, current loop, sensorless parameters, speed loop) (see section References)
- FreeMASTER software MCAT graphical control page (required speed, actual motor speed, start/stop status, DC-Bus voltage level, motor current, system status)
- FreeMASTER software speed scope (observes actual and desired speeds, DC-Bus voltage and motor current)
- FreeMASTER software high-speed recorder (reconstructed motor currents, vector control algorithm quantities)
- DC-Bus over-voltage and under-voltage, over-current, overload and start-up fail protection

3. PMSM field oriented control

3.1. Fundamental principle of PMSM 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 acceleration/deceleration. To achieve such control, Field Oriented Control is used for PM synchronous motors.

The FOC concept is based on an efficient torque control requirement, which is essential for achieving a high control dynamic. Analogous to standard DC machines, AC machines develop maximal torque when the armature current vector is perpendicular to the flux linkage vector. Thus, if only the fundamental harmonic of stator magnetomotive force is considered, the torque T_e developed by an AC machine, in vector notation, is given by the following equation:

$$T_e = \frac{3}{2} \cdot pp \cdot \overline{\psi_s} \times \bar{\imath}_s$$

Equation 1

where pp is the number of motor pole-pairs, i_s is stator current vector and ψ_s represents vector of the stator flux. Constant 3/2 indicates a non-power invariant transformation form.

In instances of DC machines, the requirement to have the rotor flux vector perpendicular to the stator current vector is satisfied by the mechanical commutator. Because there is no such mechanical commutator in AC Permanent Magnet Synchronous Machines (PMSM), the functionality of the commutator has to be substituted electrically by enhanced current control. This reveal that stator current vector should be oriented in such a way that component necessary for magnetizing of the machine (flux component) shall be isolated from the torque producing component.

This can be accomplished by decomposing the current vector into two components projected in the reference frame, often called the *dq* frame that rotates synchronously with the rotor. It has become a standard to position the *dq* reference frame such that the d-axis is aligned with the position of the rotor flux vector, so that the current in the d-axis will alter the amplitude of the rotor flux linkage vector. The reference frame position must be updated so that the d-axis should be always aligned with the rotor flux axis.

Because the rotor flux axis is locked to the rotor position, when using PMSM machines, a mechanical position transducer or position observer can be utilized to measure the rotor position and the position of the rotor flux axis. When the reference frame phase is set such that the d-axis is aligned with the rotor flux axis, the current in the q-axis represents solely the torque producing current component.

What further resulted from setting the reference frame speed to be synchronous with the rotor flux axis speed is that both d and q axis current components are DC values. This implies utilization of simple current controllers to control the demanded torque and magnetizing flux of the machine, thus simplifying the control structure design.

Figure 1 shows the basic structure of the vector control algorithm for the PM synchronous motor. To perform vector control, it is necessary to perform the following four steps:

- Measure the motor quantities (DC link voltage and currents, rotor position/speed).
- Transform measured currents into the two-phase orthogonal system (α, β) using a Clarke transformation. After that transform the currents in α, β coordinates into the d, q reference frame using a Park transformation.
- The stator current torque (i_{sq}) and flux (i_{sd}) producing components are separately controlled in d, q rotating frame.
- The output of the control is stator voltage space vector and it is transformed by an inverse Park transformation back from the d, q reference frame into the two-phase orthogonal system fixed with the stator. The output three-phase voltage is generated using a space vector modulation.

Clarke/Park transformations discussed above are part of the Automotive Math and Motor Control Library set (see section *References*).

To be able to decompose currents into torque and flux producing components (i_{sd}, i_{sq}), position of the motor-magnetizing flux has to be known. This requires knowledge of accurate rotor position as being strictly fixed with magnetic flux. This application note deals with the sensorless FOC control where the position and velocity is obtained by either a position/velocity estimator or incremental Encoder sensor.

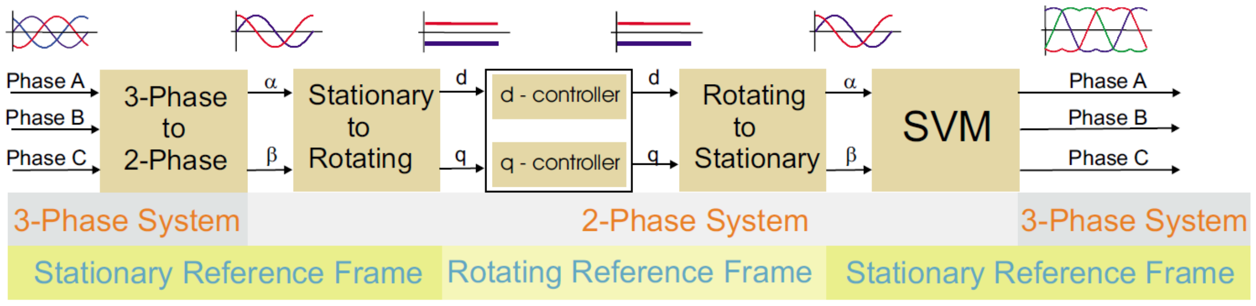


Figure 1. **Field oriented control transformations**

3.2. PMSM model in quadrature phase synchronous reference frame

Quadrature phase model in synchronous reference frame is very popular for field oriented control structures, because both controllable quantities, current and voltage, are DC values. This allows to employ only simple controllers to force the machine currents into the defined states. Furthermore, full decoupling of the machine flux and torque can be achieved, which allows dynamic torque, speed and position control.

The equations describing voltages in the three phase windings of a permanent magnet synchronous machine can be written in matrix form as follows:

$$\begin{bmatrix} u_a \\ u_b \\ u_c \end{bmatrix} = R_s \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} + \frac{d}{dt} \begin{bmatrix} \psi_a \\ \psi_b \\ \psi_c \end{bmatrix}$$

Equation 2

where the total linkage flux in each phase is given as:

$$\begin{bmatrix}\psi_a\\ \psi_b\\ \psi_c\end{bmatrix} = \begin{bmatrix}L_{aa} & L_{ab} & L_{ac}\\ L_{ba} & L_{bb} & L_{bc}\\ L_{ca} & L_{cb} & L_{cc}\end{bmatrix}\begin{bmatrix}i_a\\ i_b\\ i_c\end{bmatrix} + \Psi_{PM}\begin{bmatrix}\cos(\theta_e)\\ \cos\left(\theta_e - \frac{2\pi}{3}\right)\\ \cos\left(\theta_e + \frac{2\pi}{3}\right)\end{bmatrix}$$

Equation 3

where L_{aa}, L_{bb}, L_{cc}, are stator phase self-inductances and $L_{ab}=L_{ba}$, $L_{bc}=L_{cb}$, $L_{ca}=L_{ac}$ are mutual inductances between respective stator phases. The term Ψ_{PM} represents the magnetic flux generated by the rotor permanent magnets, and θ_e is electrical rotor angle.

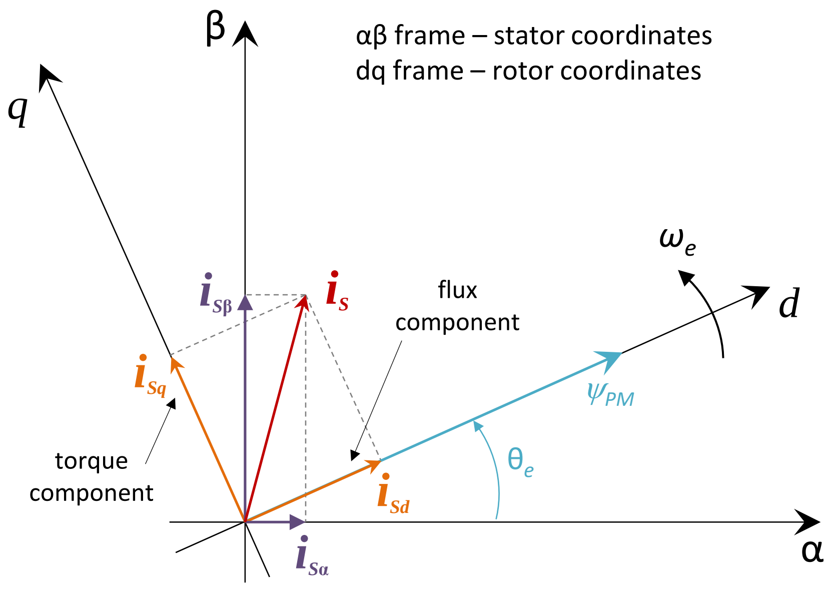


Figure 2. **Orientation of stator (stationary) and rotor (rotational) reference frames, with current components transformed into both frames**

The voltage equation of the quadrature phase synchronous reference frame model can be obtained by transforming the three phase voltage equations (*Equation 2*) and flux equations (*Equation 3*) into a two phase rotational frame which is aligned and rotates synchronously with the rotor as shown in *Figure 2*. Such transformation, after some mathematical corrections, yields the following set of equations:

$$\begin{bmatrix}u_d\\ u_q\end{bmatrix} = R_s\begin{bmatrix}i_d\\ i_q\end{bmatrix} + \begin{bmatrix}L_d & 0\\ 0 & L_q\end{bmatrix}\frac{d}{dt}\begin{bmatrix}i_d\\ i_q\end{bmatrix} + \omega_e\begin{bmatrix}0 & -L_q\\ L_d & 0\end{bmatrix}\begin{bmatrix}i_d\\ i_q\end{bmatrix} + \omega_e\Psi_{PM}\begin{bmatrix}0\\ 1\end{bmatrix}$$

Equation 4

where ω_e is electrical rotor speed. It can be seen that *Equation 4*

, represents a non-linear cross dependent system, with cross-coupling terms in both d and q axis and back-EMF voltage component in the q-axis. When FOC concept is employed, both cross-coupling terms shall be compensated in order to allow independent control of current d and q components. Design of the controllers is then governed by following pair of equations, derived from *Equation 4* after compensation:

$$u_d = R_s i_d + L_d \frac{di_d}{dt}$$

Equation 5

$$u_q = R_s i_q + L_q \frac{di_q}{dt}$$

Equation 6

This equation describes the model of the plant for d and q current loop. Both equations are structurally identical, therefore the same approach of controller design can be adopted for both d and q controllers. The only difference is in values of d and q axis inductances, which results in different gains of the controllers. Considering closed loop feedback control of a plant model as in either equation, using standard PI controllers, then the controller proportional and integral gains can be derived, using a pole-placement method, as follows:

$$K_p = 2\xi\omega_0 L - R$$

Equation 7

$$K_i = \omega_0{}^2 L$$

Equation 8

where ω_0 represents the system *natural frequency* [rad/sec] and ξ is the Damping factor [-] of the current control loop.

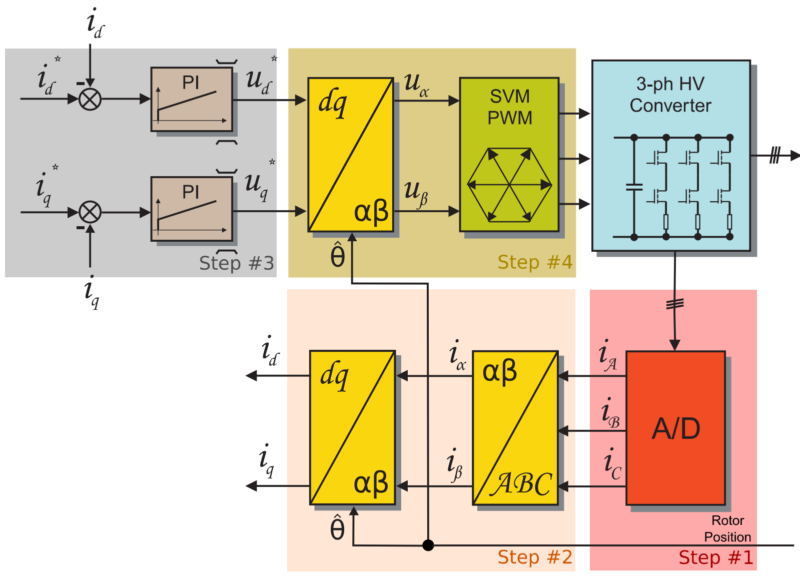


Figure 3. **FOC Control Structure**

3.3. Output voltage actuation and phase current measurement

The 3-phase voltage source inverter shown in *Figure 4* uses three shunt resistors (R56, R57, R58) placed in three legs of the inverter as phase current sensors. Stator phase current which flows through the shunt resistor produces a voltage drop which is interfaced to the AD converter of microcontroller through conditional circuitry (refer to DEVKIT-MOTORGD Schematic available at nxp.com).

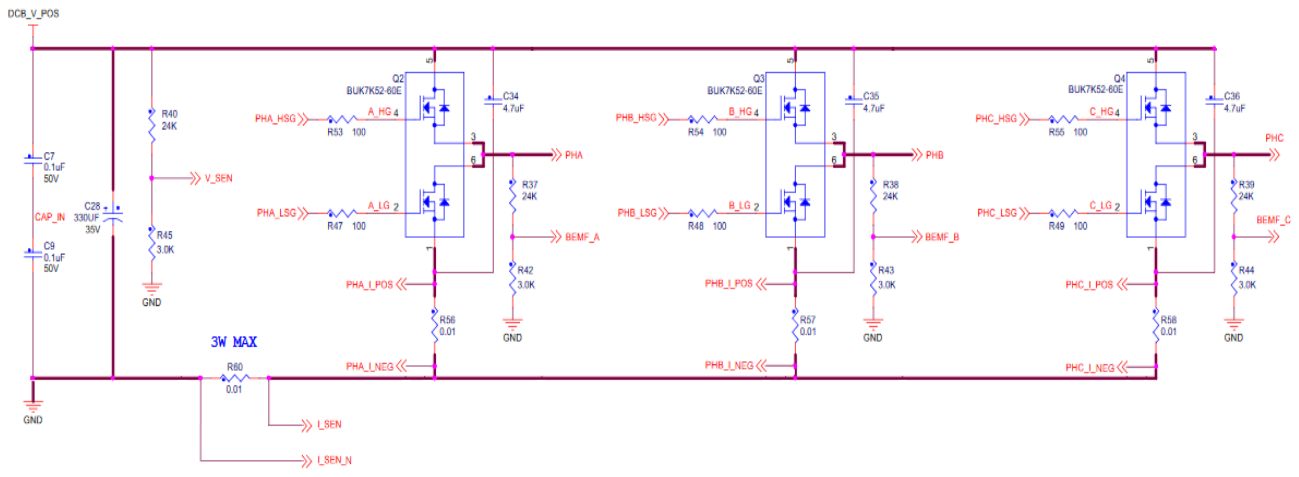


Figure 4. **3-phase DC/AC inverter with shunt resistors for current measurement**

Figure 5 shows a gain setup and input signal filtering circuit for operational amplifier which provides the conditional circuitry and adjusts voltages to fit into the ADC input voltage range.

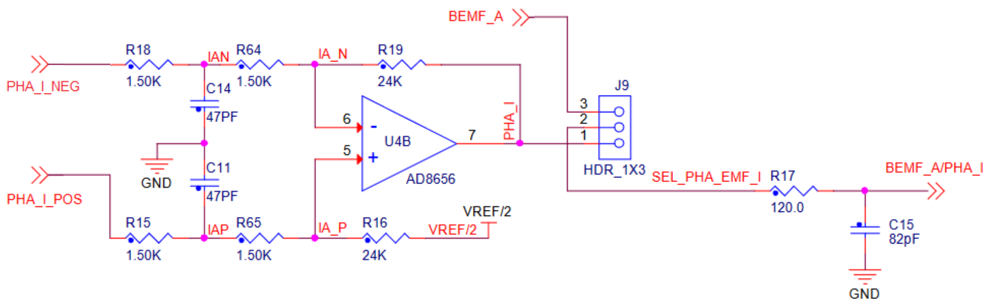


Figure 5. **Phase current measurement conditional circuitry**

The phase current sampling technique is a challenging task for detection of phase current differences and for acquiring full three phase information of stator current by its reconstruction. Phase currents flowing through shunt resistors produces a voltage drop which needs to be appropriately sampled by the AD converter when low-side transistors are switched on. The current cannot be measured by the current shunt resistors at an arbitrary moment. This is because that the current only flows through the shunt resistor when the bottom transistor of the respective inverter leg is switched on. Therefore, considering *Figure 4*, phase A current is measured using the R56 shunt resistor and can only be sampled when the low side transistor Q2 is switched on. Correspondingly, the current in phase B can only be measured if the low side transistor Q3 is switched on, and the current in phase C can only be measured if the low side transistor Q4 is switched on. To get an actual instant of current sensing, voltage waveform analysis has to be performed.

Generated duty cycles (phase A, phase B, phase C) of two different PWM periods are shown in *Figure 6*. These phase voltage waveforms correspond to a center-aligned PWM with sine-wave modulation. As shown in the following figure, (PWM period I), the best sampling instant of phase current is in the middle of the PWM period, where all bottom transistors are switched on. However, not all three currents can be measured at an arbitrary voltage shape. PWM period II in the following figure shows the case when the bottom transistor of phase A is ON for a very short time. If the ON time is shorter than a certain critical time (depends on hardware design), the current cannot be correctly measured.

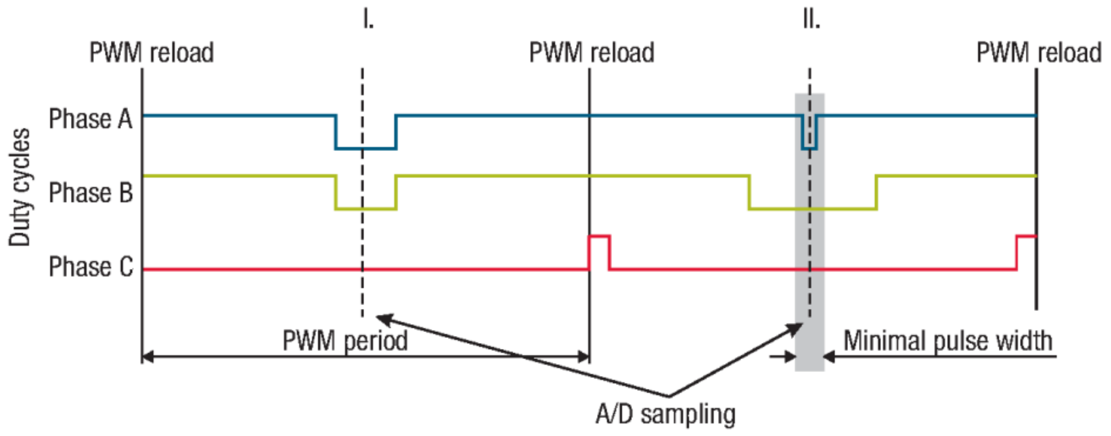


Figure 6. **Generated phase duty cycles in different PWM periods**

In standard motor operation, where the supplied voltage is generated using the space vector modulation, the sampling instant of phase current takes place in the middle of the PWM period in which all bottom transistors are switched on. If the duty cycle goes to 100%, there is an instant when one of the bottom transistors is switched on for a very short time period. Therefore, only two currents are measured and the third one is calculated from equation:

$$i_A + i_B + i_C = 0$$

Equation 9

NOTE

Although, there are three shunt resistors available on the power stage board (R56, R57, R58), S32K144 has only two AD converters that measure two currents simultaneously in this application. Third stator current is calculated based on *Equation 9*. To measure two stator currents in two inverter legs correctly, minimum ON times for the low-side switches are ensured by appropriate duty cycle limit.

3.4. Rotor position/speed estimation

In this application, rotor position and speed are either estimated by back-EMF observer or obtained by Encoder sensor. Back-EMF observer as well as incremental Encoder sensor provide only relative position. To get absolute position, initial position must be known. This application uses mechanical rotor alignment when the rotor is moved from unknown to known position applying DC voltage.

The alignment algorithm applies DC voltage to d-axis resulting full DC voltage applied to phase A and negative half of the DC voltage applied to phase B, C for a certain period. This will cause the rotor to move to "align" position, where stator and rotor fluxes are aligned. The rotor position in which the rotor stabilizes after applying DC voltage is set as zero position. Motor is ready to produce full startup torque once the rotor is properly aligned.

Application in Sensorless mode must start with open loop start-up sequence to move the motor up to a speed value where the observer provides sufficiently accurate speed and position estimations. As soon as the observer provides appropriate estimates, application transits to closed-loop mode, when the rotor speed and position calculation is based on the estimation of a BEMF in the stationary reference frame using a Luenberger type of observer. BEMF observer is as a part of the NXP's Automotive Math and Motor Control library. Structure and implementation details are discussed in section *4.3.4*.

3.5. Field weakening

Field weakening is an advanced control approach that extends standard FOC to allow electric motor operation beyond a base speed. The back electromotive force (EMF) is proportional to the rotor speed and counteracts the motor supply voltage. If a given speed is to be reached, the terminal voltage must be increased to match the increased stator back-EMF. A sufficient voltage is available from the inverter in the operation up to the base speed. Beyond the base speed, motor voltages u_d and u_q are limited and cannot be increased because of the ceiling voltage given by inverter. Base speed defines the rotor speed at which the back-EMF reaches maximal value and motor still produces the maximal torque.

As the difference between the induced back-EMF and the supply voltage decreases, the phase current flow is limited, hence the currents i_d and i_q cannot be controlled sufficiently. Further increase of speed would eventually result in back-EMF voltage equal to the limited stator voltage, which means a complete loss of current control. The only way to retain the current control even beyond the base speed is to lower the generated back-EMF by weakening the flux that links the stator winding. Base speed splits the whole speed motor operation into two regions: constant torque and constant power, see *Figure 7*.

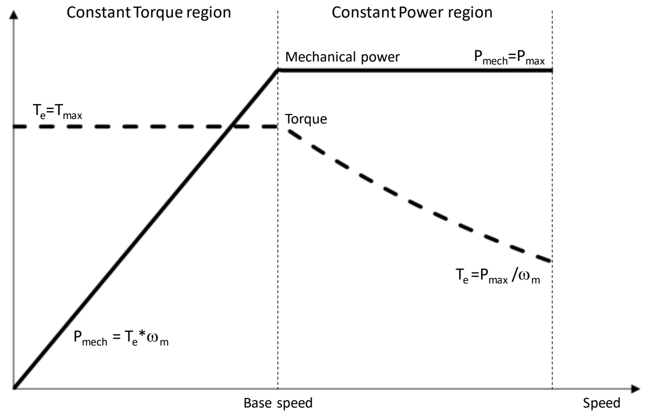


Figure 7. **Constant torque/power operating regions**

Operation in constant torque region means that maximal torque can be constantly developed while the output power increases with the rotor speed. The phase voltage increases linearly with the speed and the current is controlled to its reference. The operation in constant power region is characterized by a rapid decrease in developed torque while the output power remains constant. The phase voltage is at its limit while the stator flux decreases proportionally with the rotor speed, see *Figure 8*.

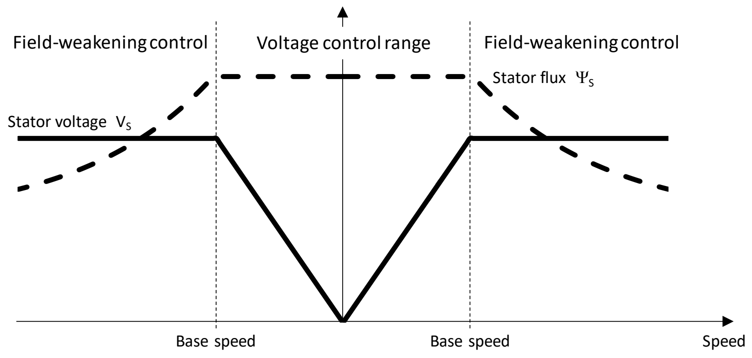


Figure 8. **Constant flux/voltage operational regions**

FOC splits phase currents into the q-axis torque component and d-axis flux component. The flux current component I_d is used to weaken the stator magnetic flux linkage Ψ_S. Reduced stator flux Ψ_S yields to lower Back-EMF and condition of Field Weakening is met. More details can be seen from the following phasor diagrams of the PMSM motor operated exposing FOC control without (left) and with FW (right), *Figure 9*.

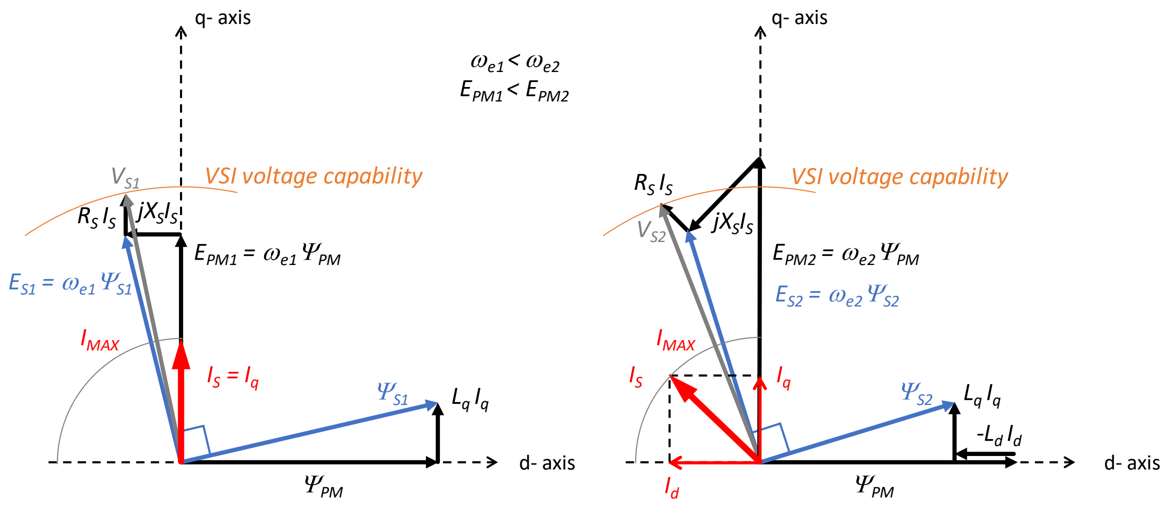


Figure 9. **Steady-state phasor diagram of PMSM operation up to base speed (left) and above speed (right)**

FOC without FW is operated demanding d-axis current component to be zero (I_d=0) to excite electric machine just by permanent magnets mounted on the rotor. This is an operation within constant torque region (see *Figure 7*), since whole amount of the stator current consists of the torque producing component I_q only (see *Figure 9* left). Stator magnetic flux linkage Ψ_{S1} is composed of rotor magnetic flux linkage Ψ_{PM}, which represents the major contribution and small amount of the magnetic flux linkage in q-axis L_qI_q produced by q-axis current component I_q. Based on the Faraday's law, rotor magnetic flux linkage Ψ_{PM} and stator magnetic flux linkage Ψ_{S1} produce back-EMF voltage $E_{PM1}=\omega_{e1}\Psi_{PM}$ perpendicularly oriented to rotor magnetic flux Ψ_{PM} in q-axis and back EMF voltage $E_{S1}=\omega_{e1}\Psi_{S1}$ perpendicularly oriented to stator magnetic flux Ψ_{S1}, respectively (see *Figure 9* left). Both voltages are directly proportional to the rotor speed ω_{e1}. If the rotor speed exceeds the base speed, the back-EMF voltage $E_{S1}=\omega_{e1}\Psi_{S1}$ approaches the limit given by VSI and I_q current cannot be controlled. Hence, field weakening has to take place.

In FW operation, I_d current is controlled to negative values to “weaken” stator flux linkage Ψ_{S2} by $-L_dI_d$ component as shown in *Figure 9* right. Thanks to this field weakening approach, back-EMF voltage induced in the stator windings E_{S2} is reduced below the VSI voltage capability even though E_{PM2} exceeds it. I_q current can be controlled again to develop torque as demanded. Unlike the previous case, this is an operation within constant power region (see *Figure 7*), where I_q current is limited due to I_s current vector size limitation (see *Figure 9* right). In FW operation, stator magnetic flux linkage Ψ_S consists of three components now: rotor magnetic flux linkage Ψ_{PM}, magnetic flux linkage in q-axis $\Psi_q = L_qI_q$ produced by q-axis current component I_q and magnetic flux linkage in d-axis $\Psi_d = -L_dI_d$ produced by negative d-axis I_d current component that counteracts to Ψ_{PM}.

There are some limiting factors that must be taken into account when operating FOC control with field weakening:

- Voltage amplitude *u_max* is limited by power as shown in *Figure 10* left
- Phase current amplitude *i_max* is limited by capabilities of power devices and motor thermal design as shown in *Figure 10* right
- Flux linkage in d-axis is limited to prevent demagnetization of the permanent magnets

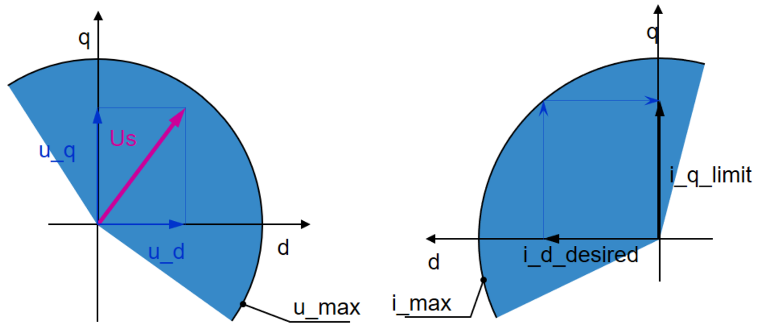


Figure 10. **Voltage (left) and current (right) limits for PMSM drive operation**

NXP’s Automotive Math and Motor Control library offers a software solution for the FOC with field weakening respecting all limitations discussed above. This library based function is discussed in section *4.3.4*.

4. Software implementation on the S32K144

4.1. S32K144 – Key modules for PMSM FOC control

The S32K144 device includes modules such as the FlexTimer Module (FTM), Trigger MUX Control (TRGMUX), Programmable Delay Block (PDB) and Analogue-to-Digital Converter (ADC) suitable for control applications, in particular, motor control applications. These modules are directly interconnected and can be configured to meet various motor control application requirements. *Figure 11* shows module interconnection for a typical PMSM FOC application working in Sensorless or Sensorbased mode using dual shunt current sensing. The modules are described below and a detailed description can be found in the S32K1xx Series Reference Manual (see section *References*).

4.1.1. Module interconnection

The modules involved in output actuation, data acquisition and the synchronization of actuation and acquisition, form the so-called Control Loop. This control loop consists of the FTM, TRGMUX, PDB, and ADC modules. The control loop is very flexible in operation and can support static, dynamic or asynchronous timing.

Each control loop cycle can be initiated either by FTM initialization trigger *init_trig* or by FTM external trigger *ext_trig*. While *init_trig* signal is generated at beginning of PWM cycle, *ext_trig* can be generated any time within the PWM period based on the value defined in the corresponding FTM Channel Value register CnV.

FTM trigger signal is routed to hardware trigger input of the PDB module through flexible TRGMUX unit. In S32K14x, there are two ADC modules and two PDB modules that work in pairs. This means that PDB0 is linked with ADC0 and PDB1 is linked with ADC1.

PDB pre-triggers *ch0pretrigx* are used as a precondition for ADC module. They are directly connected to ADHWTS ports to select ADC channels as well as order of the channels by configurable pre-triggers delays. When ADC receives rising edge of the trigger, ADC will start conversion according to the order defined by pre-triggers *ch0pretrigx*.

PDB pre-trigger delays must be properly set to allow reliable operation between PDB and corresponding ADC module. When the first pre-trigger is asserted, associated lock of the pre-trigger becomes active until corresponding conversion is not completed. This associated lock is released by corresponding ADC conversion complete flag ADC_SC1[COCOx]. This means that next pre-trigger can be generated only if the ongoing conversion is completed.

Second FTM module can work in Quadrature Decoder mode counting rising/falling edges of the Phase A and Phase B Encoder signals to determine the rotor position and speed independently from the control loop (see section *4.2.2.2*).

Detailed description can be found in the S32K1xx Series Reference Manual (see section *References*).

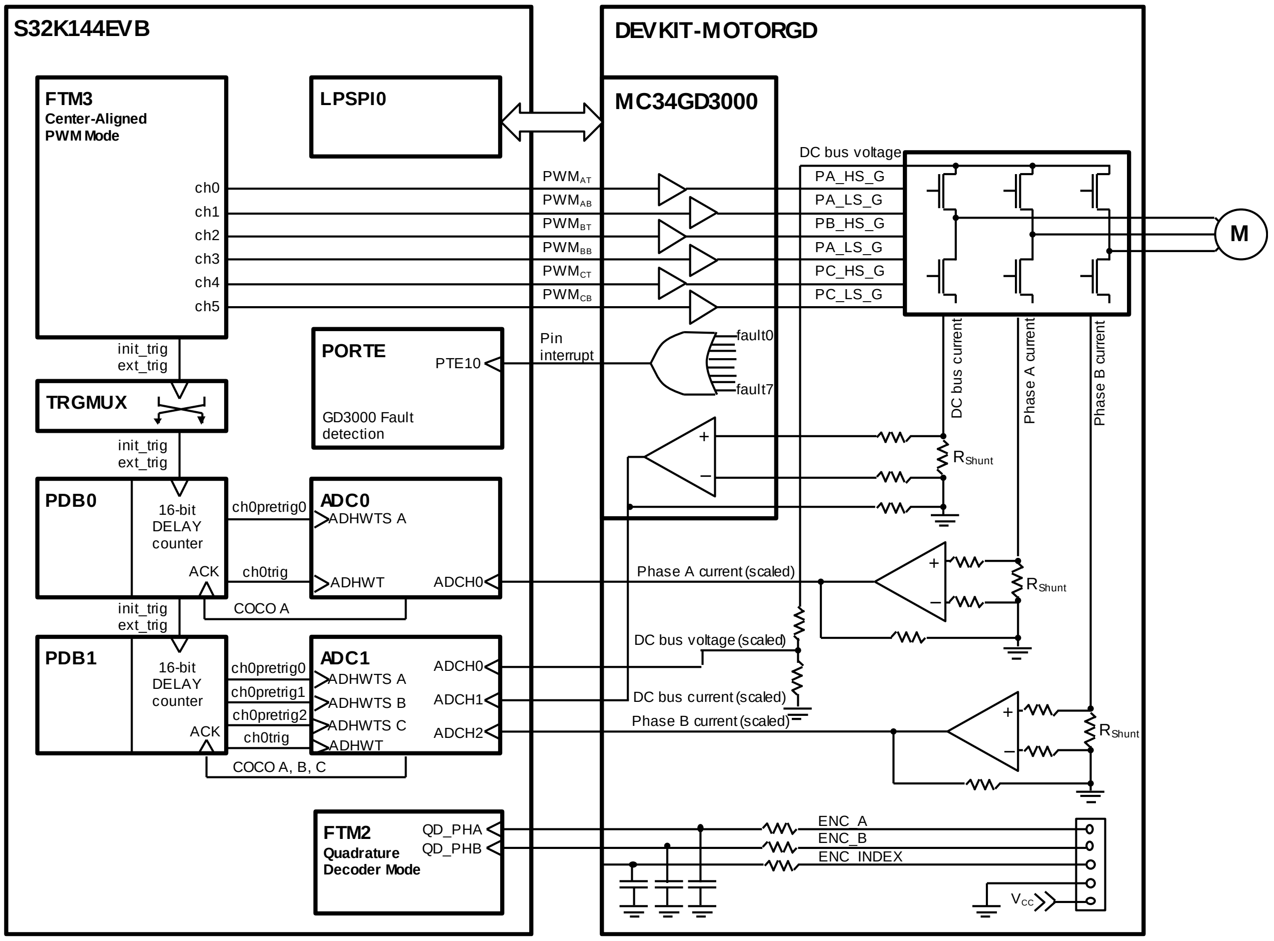


Figure 11. **S32K144 module interconnection**

4.1.2. S32K144 and FETs pre-driver interconnection

Excitation of power FETs is ensured by NXP MC34GD3000 pre-driver. This analog device is equipped with charge pump that ensures external FETs drive at low power supply voltages. Moreover, three external bootstrap capacitors provide gate charge to the high-side FETs (see section *References*).

Configuration of MC34GD3000 pre-driver is realized via LPSPI0 module. The MC34GD3000 allows different operating modes to be set and locked by SPI commands. SPI commands also report condition of the MC34GD3000 based on the internal monitoring circuits and fault detection logic. S32K144 detects fault state of the MC34GD3000 by means of interrupt signal on PTE10 pin. Integrated current sensing amplifier with analog comparator allow to measure DC bus current and detect overcurrent. Interconnection between S32K144 and MC34GD3000 is briefly depicted in *Figure 11*.

4.1.3. Module involvement in digital PMSM Sensorless control loop

This section will discuss timing and modules synchronization to accomplish PMSM Sensorless FOC on the S32K144 and the internal hardware features.

The time diagram of the automatic synchronization between PWM and ADC in the PMSM application is shown in *Figure 12*.

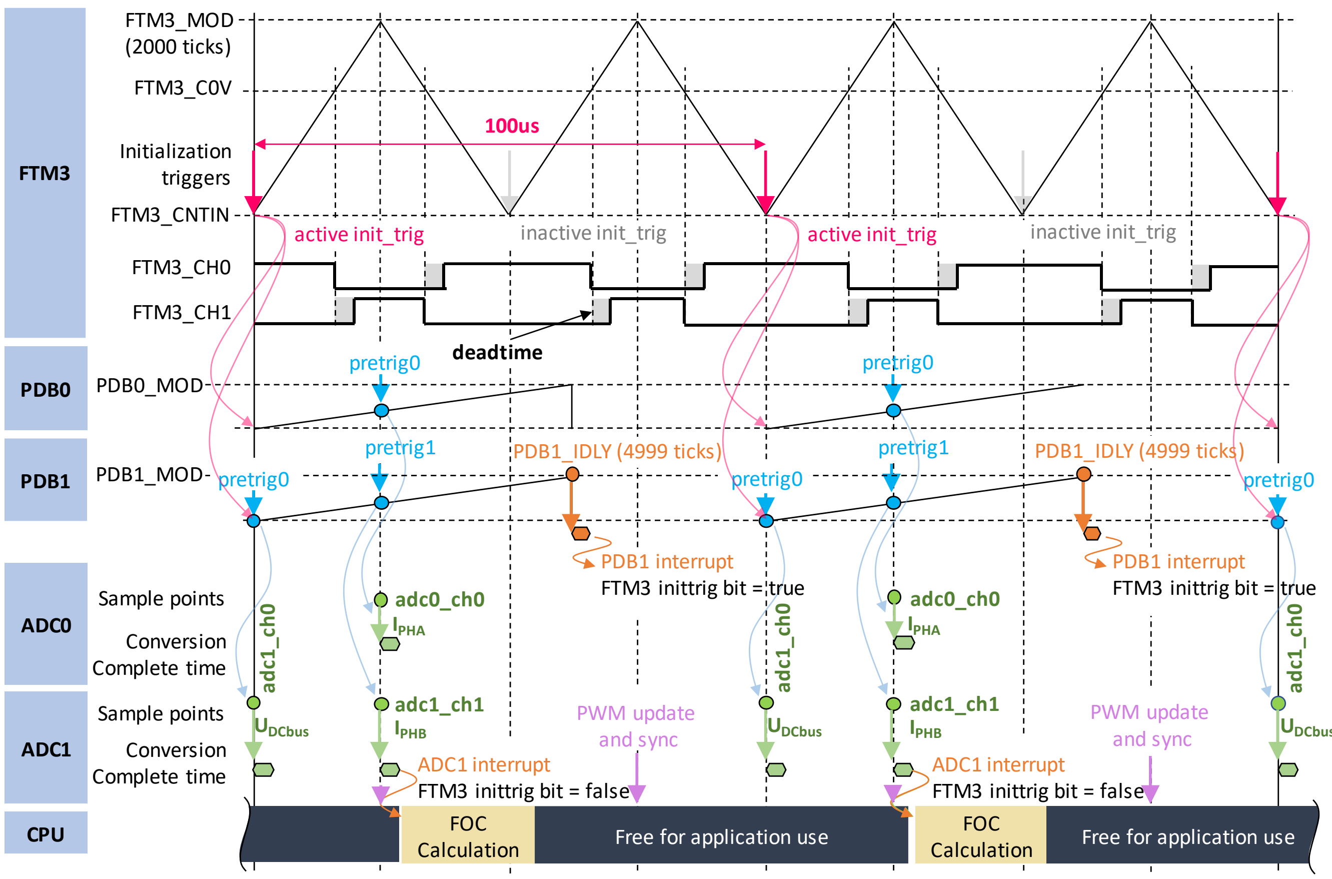


Figure 12. **Time Diagram of PWM and ADC Synchronization**

The PMSM Sensorless FOC control with two shunt current measurement is based on static timing; meaning the trigger point instances of the ADC conversions are located at same place within one control loop cycle.

Each control cycle starts with FTM3 initialization trigger *init_trig*, which is generated at beginning of the PWM cycle as shown in *Figure 12*. Initialization trigger restarts PDB0 and PDB1 modules and updates their double buffered registers. ADC0 and ADC1 channels are triggered based on the PDB0 and PDB1 pre-trigger delays. When PDB counter reaches first pre-trigger delay value, PDB initiates first ADC channel measurement.

DC bus voltage measurement is triggered first by PDB1, at beginning of the PWM cycle (pretrig0). Phase A and Phase B stator currents are measured simultaneously in the middle of the PWM cycle, when bottom transistors of both inverter legs are closed, and currents flow through shunt resistors. While PDB0 triggers Phase A current measurement at pretrig0, Phase B current measurement is triggered by PDB1 at pretrig1. The ADC conversion results are automatically stored into a predefined queue in memory.

The CPU is triggered by the ADC1 conversion complete interrupt service routine. Based on the stored ADC0 and ADC1 values, the current PI controllers calculate new PWM duty cycles. These are then sent as a new reference for PWM module (FTM3) and become effective in the next PWM cycle.

FTM3 initialization trigger is disabled in ADC1 Conversion Complete interrupt service routine. As a consequence, PDB0 and PDB1 are not triggered in the next PWM period due to the missing *init_trig* signal. FTM3 initialization trigger is reenabled again in PDB1 interrupt service routine as soon as the opportunity for generating *init_trig* signal elapses. This strategy ensures ADC0 and ADC1 sampling every second opportunity as depicted in *Figure 12*.

4.2. S32K144 Device initialization

To simplify and accelerate application development, embedded part of the PMSM Sensorless motor control application has been created using S32 Software Development Kit – S32 SDK. S32K144 can be configured either by means of the Processor Expert extension, or programmed directly using SDK drivers. Peripherals are initialized at beginning of the main() function. For each S32K144 module, there is a specific configuration function that uses S32 SDK APIs and configuration structures generated by PEx to configure the MCU.

- McuClockConfig() – MCU clock configuration
- McuPowerConfig() – MCU power management configuration
- McuTrigmuxConfig() – TRGMUX module configuration
- McuPinsConfig() – PINs and PORT modules configuration
- McuLpuartConfig() – LPUART module configuration
- McuAdcConfig() – ADC modules configuration
- McuPdbConfig() – PDB modules configuration
- McuFtmConfig() – FTM modules configuration

Detailed SDK documentation can be found in folder created with S32 Design Studio installation. (*References*).

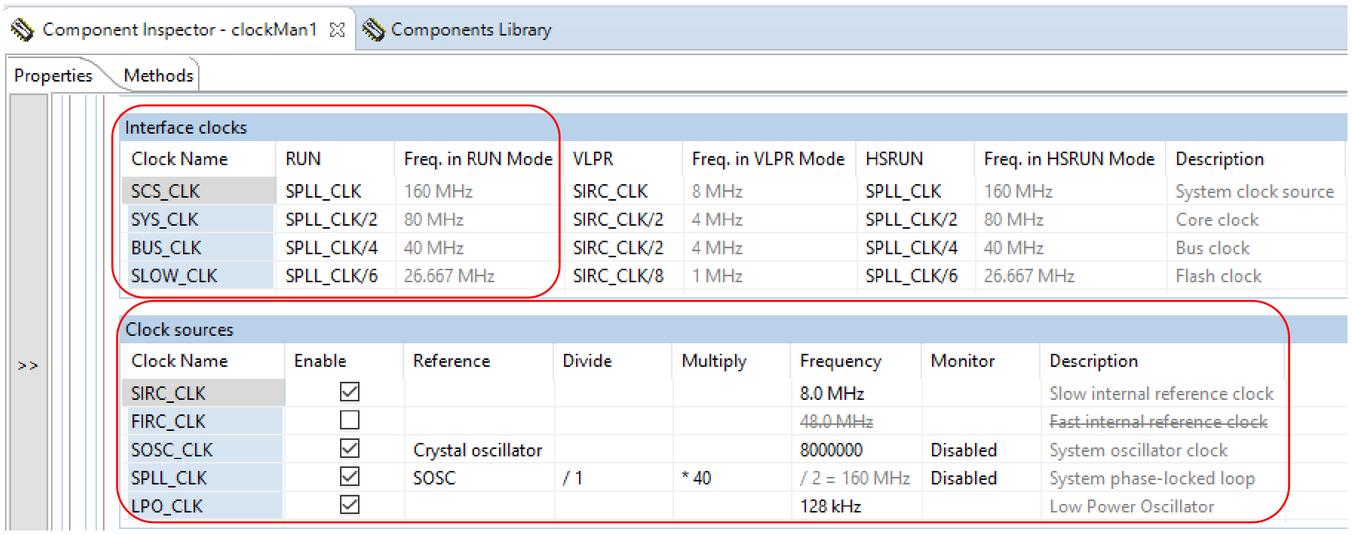
4.2.1. Clock configuration and power management

S32K144 features a complex clocking sourcing, distribution and power management. To run a core of the S32K144 as well as some MCU peripherals at maximum frequency 80 MHz in normal RUN mode, S32K144 is supplied externally by 8 MHz crystal. This clock source supplies Phase-lock-loop (PLL), which circuit multiplies frequency by 40 and divides by 2 resulting 160 MHz frequency on output. PLL output is then divided by 2 to supply core and system (80 MHz), further divided by two and three to supply bus clock (40 MHz) and flash clock (26.67 MHz), respectively. This clock configuration belongs to one of the typical and recommended. It is summarized in *Table 1*.

Table 1. **S32K144 clock configuration in RUN mode**

Clock	Frequency
CORE_CLOCK	80 MHz
SYS_CLK	80 MHz
BUS_CLK	40 MHz
FLASH_CLK	26.67 MHz (max freq. in RUN mode)

This clock configuration and power management can be setup by S32 Processor Expert. Preview of the S32K144 clock sourcing and distribution is shown in *Figure 13*.



Interface clocks

Clock Name	RUN	Freq. in RUN Mode	VLPR	Freq. in VLPR Mode	HSRUN	Freq. in HSRUN Mode	Description
SCS_CLK	SPLL_CLK	160 MHz	SIRC_CLK	8 MHz	SPLL_CLK	160 MHz	System clock source
SYS_CLK	SPLL_CLK/2	80 MHz	SIRC_CLK/2	4 MHz	SPLL_CLK/2	80 MHz	Core clock
BUS_CLK	SPLL_CLK/4	40 MHz	SIRC_CLK/2	4 MHz	SPLL_CLK/4	40 MHz	Bus clock
SLOW_CLK	SPLL_CLK/6	26.667 MHz	SIRC_CLK/8	1 MHz	SPLL_CLK/6	26.667 MHz	Flash clock

Clock sources

Clock Name	Enable	Reference	Divide	Multiply	Frequency	Monitor	Description
SIRC_CLK	☑				8.0 MHz		Slow internal reference clock
FIRC_CLK	☐				~~48.0 MHz~~		~~Fast internal reference clock~~
SOSC_CLK	☑	Crystal oscillator			8000000	Disabled	System oscillator clock
SPLL_CLK	☑	SOSC	/ 1	* 40	/ 2 = 160 MHz	Disabled	System phase-locked loop
LPO_CLK	☑				128 kHz		Low Power Oscillator

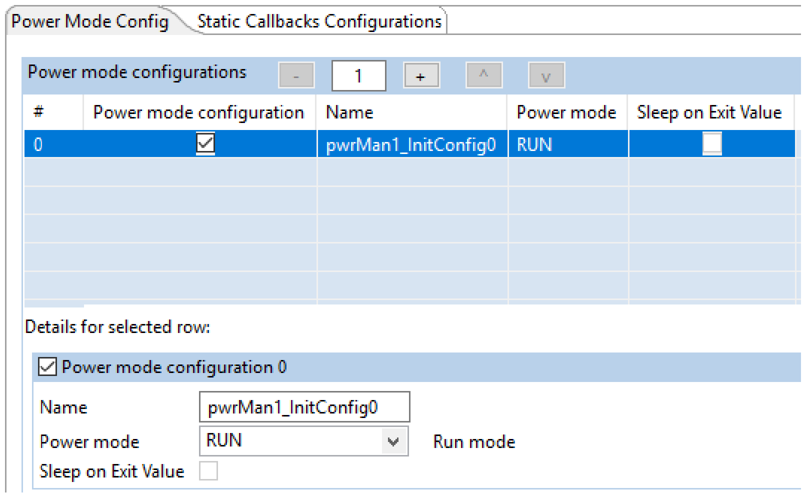
Figure 13. **S32K144 clock configuration in Processor Expert**

Once the clock configuration is set, Processor Expert generates static configuration structure *clockMan1_InitConfig0*, that is called by SDK's *CLOCK_SYS_Init* function through array of the configuration pointers *g_clockManConfigsArr*, *Example 1*.

Example 1. S32K144 clock configuration controlled by S32 SDK

```
void McuClockConfig(void)
{
    /* Clock configuration for MCU and MCU's peripherals */
    CLOCK_SYS_Init(g_clockManConfigsArr,
                   CLOCK_MANAGER_CONFIG_CNT,
                   g_clockManCallbacksArr,
                   CLOCK_MANAGER_CALLBACK_CNT);

    /* Clock configuration update */
    CLOCK_SYS_UpdateConfiguration(0, CLOCK_MANAGER_POLICY_FORCIBLE);
}

...

/*! @brief Array of pointers to User configuration structures */
clock_manager_user_config_t const * g_clockManConfigsArr[] = {
    &clockMan1_InitConfig0
};
/*! @brief Array of pointers to User defined Callbacks configuration structures */
clock_manager_callback_user_config_t * g_clockManCallbacksArr[] = {(void*)0};
/* END clockMan1. */
```

As discussed at beginning of this chapter, power management of the S32K144 is configured for normal RUN mode. This power mode can be forced by Processor Expert as well, *Figure 14*.



#	Power mode configuration	Name	Power mode	Sleep on Exit Value
0	☑	pwrMan1_InitConfig0	RUN	☐

Details for selected row:

☑ Power mode configuration 0

Name	pwrMan1_InitConfig0	
Power mode	RUN	Run mode
Sleep on Exit Value	☐	

Figure 14. **S32K144 power management configuration in Processor Expert**

Static configuration generated by Processor Expert is called by SDK's *POWER_SYS_Init* function to update power mode of the S32K144 device, *Example 2*.

Example 2. S32K144 power management controlled by S32 SDK

```
void McuPowerConfig(void)
{
    /* Power mode configuration for RUN mode */
    POWER_SYS_Init(&powerConfigsArr, 0, &powerStaticCallbacksConfigsArr,0);
    /* Power mode configuration update */
    POWER_SYS_SetMode(0,POWER_MANAGER_POLICY_AGREEMENT);
}

...

/*! @brief User Configuration structure power_managerCfg_0 */
power_manager_user_config_t pwrMan1_InitConfig0 = {
    .powerMode = POWER_MANAGER_RUN,                                   /*!< Power manager mode  */
    .sleepOnExitValue = false,                                        /*!< Sleep on exit value */
};

/*! @brief Array of pointers to User configuration structures */
power_manager_user_config_t * powerConfigsArr[] = {
    &pwrMan1_InitConfig0
};
/*! @brief Array of pointers to User defined Callbacks configuration structures */
```

Same mechanism between Processor Expert and S32 SDK works for all S32K144 peripherals, which are discussed below.

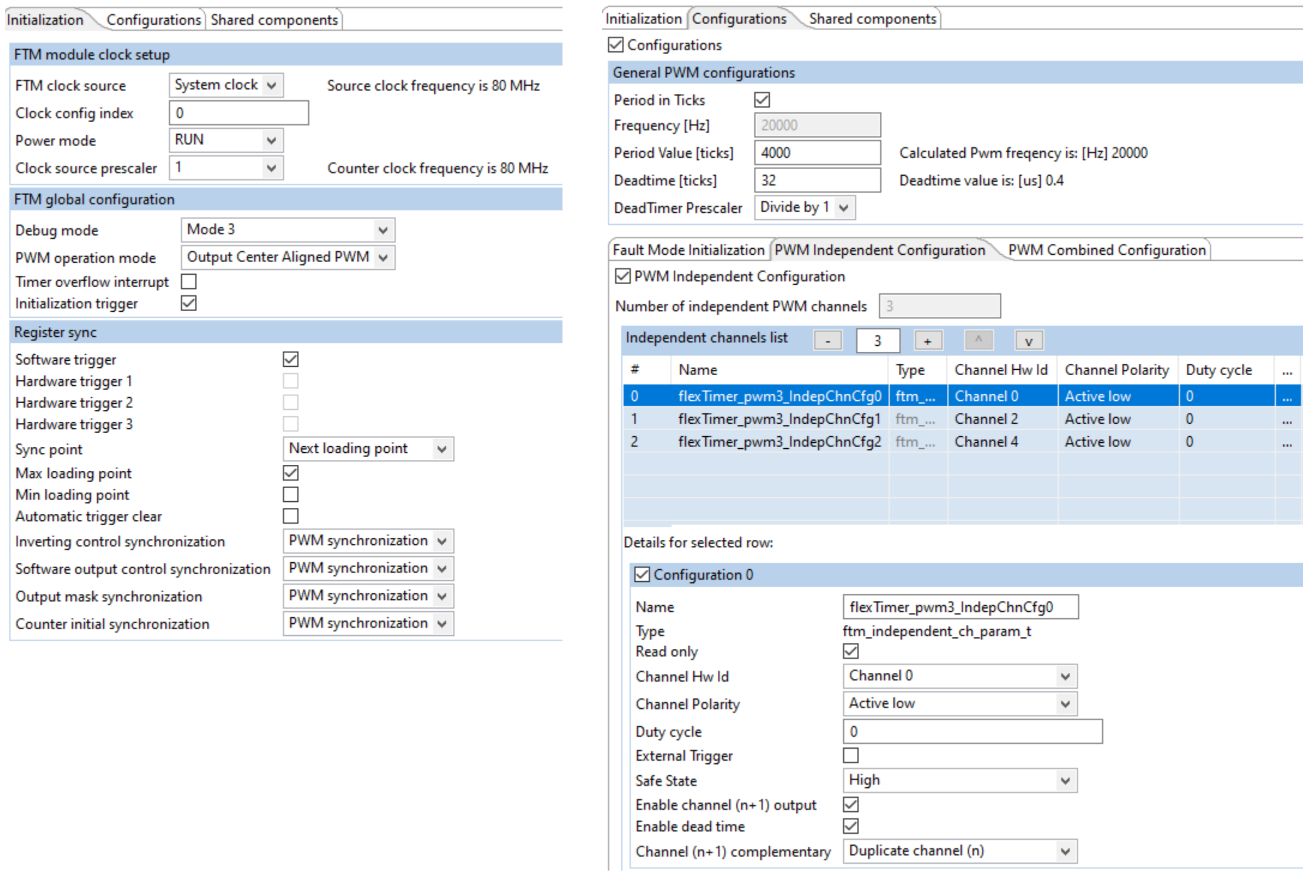
4.2.2. FlexTimer Module (FTM)

FlexTimer module (FTM) is built upon a timer with a 16-bit counter. It contains an extended set of features that meet the demands of motor control, including the signed up-counter, dead time insertion hardware, fault control inputs, enhanced triggering functionality, and initialization and polarity control.

4.2.2.1. Center-aligned PWM mode

FTM3 instance is used in PMSM Sensorless motor control application to generate center-align PWM by six, complementary oriented channels to control power MOSFETs of the DEVKIT-MOTORGD board.

As depicted in *Figure 12*, up-down counting mode is selected as a dedicated counting mode for center-align PWM. Due to the inverted logic of the high-side control inputs of the MC34GD3000 pre-driver, even channels of the FTM3 must have inverted polarity. 20 kHz PWM frequency is adjusted by FTM3 Modulo register (FTM3_MOD = 2000) taking 80MHz clock source frequency into account. To protect power MOSFETs against short circuit, deadtime 0.4μs is inserted for each complementary channels pair in number of clock ticks 32 with default deadtime prescaler 1. This FTM3 configuration can be carried out by using Processor Expert, *Figure 15*.



FTM module clock setup

FTM clock source	System clock	Source clock frequency is 80 MHz
Clock config index	0	
Power mode	RUN	
Clock source prescaler	1	Counter clock frequency is 80 MHz

FTM global configuration

Debug mode	Mode 3
PWM operation mode	Output Center Aligned PWM
Timer overflow interrupt	☐
Initialization trigger	☑

Register sync

Software trigger	☑
Hardware trigger 1	☐
Hardware trigger 2	☐
Hardware trigger 3	☐
Sync point	Next loading point
Max loading point	☑
Min loading point	☐
Automatic trigger clear	☐
Inverting control synchronization	PWM synchronization
Software output control synchronization	PWM synchronization
Output mask synchronization	PWM synchronization
Counter initial synchronization	PWM synchronization

☑ Configurations

General PWM configurations

Period in Ticks	☑	
Frequency [Hz]	20000	
Period Value [ticks]	4000	Calculated Pwm freqency is: [Hz] 20000
Deadtime [ticks]	32	Deadtime value is: [us] 0.4
DeadTimer Prescaler	Divide by 1	

Fault Mode Initialization
PWM Independent Configuration
PWM Combined Configuration

☑ PWM Independent Configuration

Number of independent PWM channels: 3

Independent channels list: 3

#	Name	Type	Channel Hw Id	Channel Polarity	Duty cycle	...
0	flexTimer_pwm3_IndepChnCfg0	ftm_...	Channel 0	Active low	0	...
1	flexTimer_pwm3_IndepChnCfg1	ftm_...	Channel 2	Active low	0	...
2	flexTimer_pwm3_IndepChnCfg2	ftm_...	Channel 4	Active low	0	...

Details for selected row:

☑ Configuration 0

Name	flexTimer_pwm3_IndepChnCfg0
Type	ftm_independent_ch_param_t
Read only	☑
Channel Hw Id	Channel 0
Channel Polarity	Active low
Duty cycle	0
External Trigger	☐
Safe State	High
Enable channel (n+1) output	☑
Enable dead time	☑
Channel (n+1) complementary	Duplicate channel (n)

Figure 15. **S32K144 FTM3 configuration in Processor Expert**

While *Initialization* tab on the left allows to configure general features of the FTM module such as clock sourcing, counter mode and register synchronization method, more specific settings related to the PWM modulation such as PWM frequency, deadtime value, channels pairs setting are configured in *Configuration* tab on the right, *Figure 15*.

As discussed in chapter *4.1.3*, to initiate control loop every second PWM cycle at beginning of the PWM period, initialization trigger is enabled. To be able to synchronize PWM and update FTM double buffered registers at certain synchronization point simultaneously, *software PWM synchronization* and *Next loading point* are enabled in *Initialization* tab, *Figure 15*. It should be noticed that *Max loading point* is the time instant, when FTM3 counter equals Modulo register value (FTM3_MOD = 2000).

Once the FTM3 setting is completed, Processor Expert generates two configuration structures *flexTimer_pwm3_InitConfig* and *flexTimer_pwm3_PwmConfig* that access and set corresponding FTM3 registers executing *FTM_DRV_Init* and *FTM_DRV_InitPwm* functions, *Example 3*.

Example 3. S32K144 FTM3 configured by S32 SDK

```
void McuFtmConfig(void)
{
    /* FTM3 module initialized as PWM signals generator */
    FTM_DRV_Init(INST_FLEXTIMER_PWM3, &flexTimer_pwm3_InitConfig, &statePwm);

    /* FTM3 module PWM initialization */
    FTM_DRV_InitPwm(INST_FLEXTIMER_PWM3, &flexTimer_pwm3_PwmConfig);

    /* Mask all FTM3 channels to disable PWM output */
    FTM_DRV_MaskOutputChannels(INST_FLEXTIMER_PWM3, 0x3F, true);
}
```

FTM_DRV_MaskOutputChannels function disables PWM output masking all FTM channels.

4.2.2.2. Quadrature decoder mode

The FTM module offers a Quadrature decoder mode to decode the quadrature signals generated by rotary sensors used in motor control domain. This mode is used to process encoder signals and determine rotor position and speed.

There are three output signals generated by incremental encoder as shown in *Figure 16*. Phase A and Phase B signals consist of a series of pulses which are phase-shifted by 90° (therefore the term "quadrature" is used). The third signal (called "Index") provides the absolute position information. In the motion control, it is used to check the pulse-counting consistency.

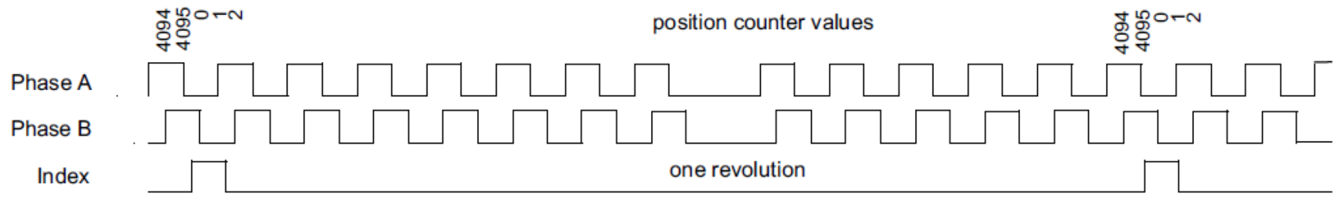
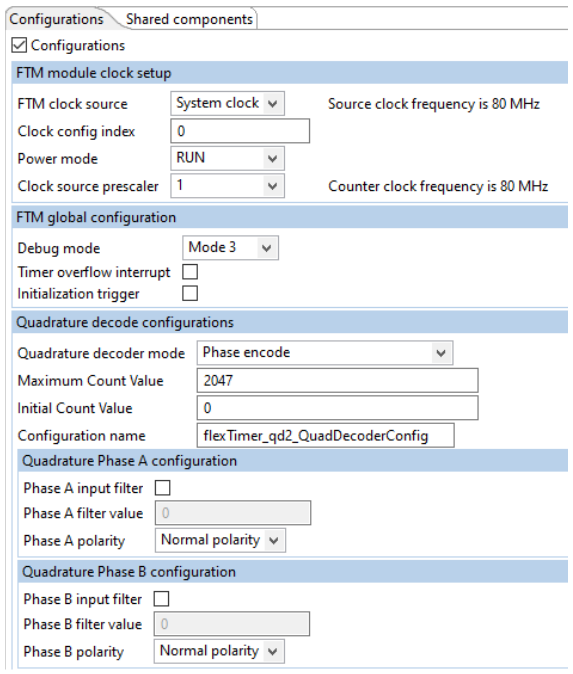


Figure 16. **Output signals of the 1024 pulses Encoder**

To process the Phase A and Phase B signals from the encoder sensor, *Quadrature decoder mode* with *Phase encode mode* have to be enabled in Processor Expert, *Figure 17*. In addition, *Maximum Count Value* has to be set according to the number of the encoder edges. In Quadrature decoder mode, the Phase A and Phase B signals indicate the counting direction as well as the counting rate. If the Phase B signal lags the Phase A signal, the FTM2 counter increments after every detected rising/falling edge of both signals. If the Phase B signal leads the Phase A signal, the FTM2 counter decrements after every detected rising/falling edge of both signals and the QUADIR bit in the FTM_QDCTRL register indicates the counting direction.



Configurations | Shared components

☑ Configurations

FTM module clock setup

FTM clock source	System clock	Source clock frequency is 80 MHz
Clock config index	0	
Power mode	RUN	
Clock source prescaler	1	Counter clock frequency is 80 MHz

FTM global configuration

Debug mode	Mode 3
Timer overflow interrupt	☐
Initialization trigger	☐

Quadrature decode configurations

Quadrature decoder mode	Phase encode
Maximum Count Value	2047
Initial Count Value	0
Configuration name	flexTimer_qd2_QuadDecoderConfig

Quadrature Phase A configuration

Phase A input filter	☐
Phase A filter value	0
Phase A polarity	Normal polarity

Quadrature Phase B configuration

Phase B input filter	☐
Phase B filter value	0
Phase B polarity	Normal polarity

Figure 17. **S32K144 FTM2 configuration in Processor Expert**

Configuration structures of the Quadrature decoder mode generated by Processor Expert are shown in *Example 4*. While more general structure flexTimer_qd2_InitConfig takes effect calling McuFtmConfig() function at very beginning in the initialization phase (see section 4.2), more specific structure for Quadrature decoder mode *quadrature_decoder_configuration* takes effect calling function *FTM_DRV_QuadDecodeStart* as soon as motor is aligned into d-axis (see section 3.4). FTM2 counter is reset in this stage, thus initial rotor position is zero.

Example 4. S32K144 FTM2 configured by S32 SDK

```
ftm_quad_decode_config_t flexTimer_qd2_QuadDecoderConfig =
    {
        FTM_QUAD_PHASE_ENCODE,
        0U,
        2047U,
        {
            false,
            0U,
            FTM_QUAD_PHASE_NORMAL, /* Phase A polarity */
        },
        {
            false,
            0U,
            FTM_QUAD_PHASE_NORMAL, /* Phase B polarity */
        }
    };

    /* Global configuration of flexTimer_qd2 */
    ftm_user_config_t  flexTimer_qd2_InitConfig =
    {
        {
            true,    /* Software trigger state */
            false,   /* Hardware trigger 1 state */
            false,   /* Hardware trigger 2 state */
            false,   /* Hardware trigger 3 state */
            false, /* Max loading point state */
            false, /* Min loading point state */
            FTM_SYSTEM_CLOCK, /* Update mode for INVCTRL register */
            FTM_SYSTEM_CLOCK, /* Update mode for SWOCTRL register */
            FTM_SYSTEM_CLOCK, /* Update mode for OUTMASK register */
            FTM_SYSTEM_CLOCK, /* Update mode for CNTIN register */
            false, /* Automatic clear of the trigger*/
            FTM_UPDATE_NOW, /* Synchronization point */
        },
         FTM_MODE_QUADRATURE_DECODER, /*!< Mode of operation for FTM */
         FTM_CLOCK_DIVID_BY_1, /* FTM clock prescaler */
         FTM_CLOCK_SOURCE_SYSTEMCLK,    /* FTM clock source */
         FTM_BDM_MODE_11, /* FTM debug mode */
         false, /* Interrupt state */
         false /* Initialization trigger */
};
```

NOTE

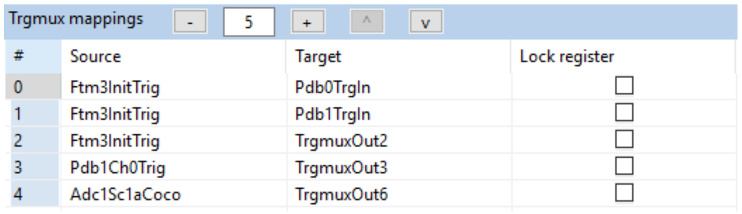
S32K144EVB and DEVKIT-MOTORGD boards are designed to process encoder signals through FTM2 module. Software example contains routine for encoder signal processing. This routine is disabled by default, since PM motor of the S32K144 motor control kit is not equipped with encoder sensor. To enable encoder signal processing routine, set ENCODER macro to *true*.

4.2.3. Trigger MUX Control (TRGMUX)

The TRGMUX provides an extremely flexible mechanism for connecting various trigger sources to multiple pins/peripherals. With the TRGMUX, each peripheral that accepts external triggers usually has one specific 32-bit trigger control register. Each control register supports up to four triggers, and each trigger can be selected from the available input triggers.

To trigger PDB0 and PDB1 modules by FTM3 initialization trigger signal *init_trig*, selection bit field SEL0 of the TRGMUX_PDB0 and TRGMUX_PDB1 registers have to be specified to define trigger source.

Processor Expert allows to generate configuration structure *trgmux1_InitConfig0* that sets all TRGMUX registers to assign trigger inputs with trigger outputs as demanded, *Figure 18* and *Example 5*.



#	Source	Target	Lock register
0	Ftm3InitTrig	Pdb0TrgIn	☐
1	Ftm3InitTrig	Pdb1TrgIn	☐
2	Ftm3InitTrig	TrgmuxOut2	☐
3	Pdb1Ch0Trig	TrgmuxOut3	☐
4	Adc1Sc1aCoco	TrgmuxOut6	☐

Figure 18. **S32K144 TRGMUX configuration in Processor Expert**

In particular, FTM3 initialization trigger signal as a source is assigned to three targets namely: PDB0, PDB1 trigger inputs and TRGMUX output 2. PDB1 channel 0 trigger is routed to TRGMUX output 3 and ADC1 conversion complete flag COCO is assigned to TRGMUX output 6. TRGMUX outputs are directly assigned to chip pins, so that triggering scheme between FTM3, PDB1 and ADC1 can be visualized by means of oscilloscope as depicted in *Figure 12*.

Example 5. S32K144 TRGMUX module controlled by S32 SDK

```
void McuTrigmuxConfig(void)
{
    /* TRGMUX module initialization */
    TRGMUX_DRV_Init(INST_TRGMUX1, &trgmux1_InitConfig0);
}

...

const trgmux_inout_mapping_config_t trgmux1_InOutMappingConfig0[5] =
{
    {TRGMUX_TRIG_SOURCE_FTM3_INIT_TRIG, TRGMUX_TARGET_MODULE_PDB0_TRG_IN, false},
    {TRGMUX_TRIG_SOURCE_FTM3_INIT_TRIG, TRGMUX_TARGET_MODULE_PDB1_TRG_IN, false},
    {TRGMUX_TRIG_SOURCE_FTM3_INIT_TRIG, TRGMUX_TARGET_MODULE_TRGMUX_OUT2, false},
    {TRGMUX_TRIG_SOURCE_PDB1_CH0_TRIG, TRGMUX_TARGET_MODULE_TRGMUX_OUT3, false},
    {TRGMUX_TRIG_SOURCE_ADC1_SC1A_COCO, TRGMUX_TARGET_MODULE_TRGMUX_OUT6, false},
};

/*! trgmux1 configuration structure */
const trgmux_user_config_t trgmux1_InitConfig0 = {
   .numInOutMappingConfigs = 5,
   .inOutMappingConfig = trgmux1_InOutMappingConfig0,
};
```

4.2.4. Programmable delay block (PDB)

The Programable Delay Block (PDB) is intended to completely avoid CPU involvement in the timed acquisition of state variables during the control cycle. The PDB module contains a 16-bit programmable delay counter that delays FTM3 initialization trigger and schedules ADC channels sampling through PDB pre-triggers delays. When FTM3 initialization trigger is detected on the PDB0 and PDB1 trigger input, PDB0 and PDB1 generate hardware signal to trigger ADC0 and ADC1 channels in order defined by pre-trigger delays, *Figure 19*.

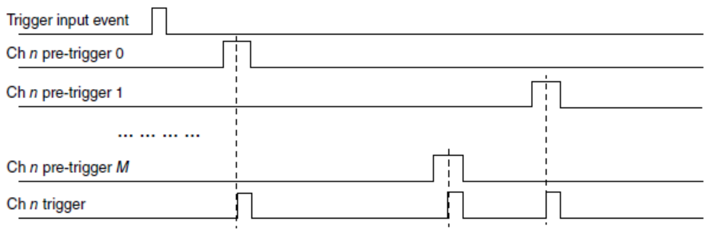


Figure 19. **PDB pre-triggers and trigger output**

PDB pre-trigger delays can be set independently using CHnDLYm registers. Since the PDB0, PDB1 and FTM3 modules are synchronized and share the same source frequency 80MHz, values of the CHnDLYm registers are set using the same time base as for PWM. *Table 2* shows all PDB0 and PDB1 pre-triggers used in PMSM Sensorless FOC motor control application.

Table 2. **PDB0 and PDB1 pre-triggers**

FOC state variable	PDB pre-trigger	CHnDLYm value [ticks]	Relation to PWM
Phase A stator current	pdb0_ch0_pretrig0	2000	In ½ of the PWM
DC bus voltage	pdb1_ch0_pretrig0	0	At beginning of the PWM
Phase B stator current	pdb1_ch0_pretrig1	2000	In ½ of the PWM

To initiate control loop every second PWM cycle, PDB1 modulo is intentionally greater than FTM3 modulo. While modulo of the FTM3 is set to 4000 clock ticks, PDB1 modulo is set to 5000 clock ticks to be able to adjust Interrupt Delay register PDB1_IDLY to 4999. This allows to invoke PDB1 interrupt in the next PWM cycle and enable FTM initialization trigger in PDB1 ISR which was disabled in the

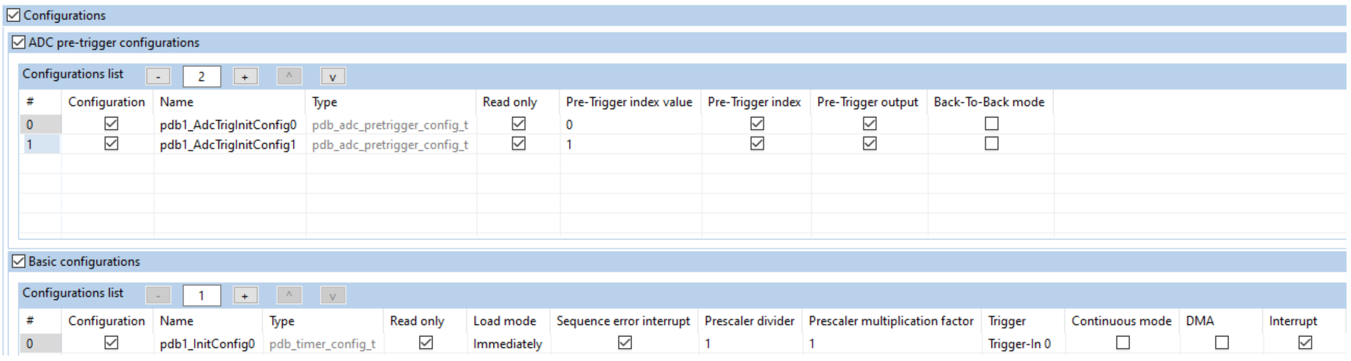
ADC1 ISR. This mechanism generates the *init_trig* signal every second reload opportunity, so that PDB and ADC instances are triggered every second PWM, *Figure 12*.

PDB Sequence Error Interrupt is activated as redundancy to protect triggering mechanism once blocked due to the wrong PDB pre-trigger delay timing. Pre-triggers delays must respect ADC conversion time that typically takes ~1.25µs considering short ADC sample time and 40MHz ADC input frequency. This time can be converted to PDB pre-trigger delay format defined in number of ticks 100.

Pre-triggers delays are static values defined only once at the initialization phase respecting ADC conversion time, hence PDB Sequence Error does not take place.

It should be also noticed that MOD, IDLY and CHnDLYx are double buffered registers, meaning values are loaded from their buffers based on the selected updating method. In this application, double buffering approach is bypassed, since the values are static values, defined only once.

General settings of the PDB module such as clock pre-scaler, input trigger source, loading mechanism for double buffered registers as well as operation mode for pre-triggers can be configured by means of Processor Expert as shown in *Figure 20*.


☑ Configurations

☑ ADC pre-trigger configurations

Configurations list 2

#	Configuration	Name	Type	Read only	Pre-Trigger index value	Pre-Trigger index	Pre-Trigger output	Back-To-Back mode
0	☑	pdb1_AdcTrigInitConfig0	pdb_adc_pretrigger_config_t	☑	0	☑	☑	☐
1	☑	pdb1_AdcTrigInitConfig1	pdb_adc_pretrigger_config_t	☑	1	☑	☑	☐

☑ Basic configurations

Configurations list 1

#	Configuration	Name	Type	Read only	Load mode	Sequence error interrupt	Prescaler divider	Prescaler multiplication factor	Trigger	Continuous mode	DMA	Interrupt
0	☑	pdb1_InitConfig0	pdb_timer_config_t	☑	Immediately	☑	1	1	Trigger-In 0	☐	☐	☑

Figure 20. **S32K144 PDB1 module and pre-triggers configuration in Processor Expert**

Processor Expert generates configuration structures *pdbN_InitConfigX* and *pdbN_AdcTrigInitConfigX* that access appropriate PDB registers *Example 6*. To set PDB modulo and PDB pre-triggers delays, *PDB_DRV_SetTimerModulusValue* and *PDB_DRV_SetAdcPreTriggerDelayValue* are used and specified by values listed in *Table 2*. This configuration is loaded calling *McuPdbConfig()* entering *main()* function.

Example 6. S32K144 PDB instances controlled by S32 SDK

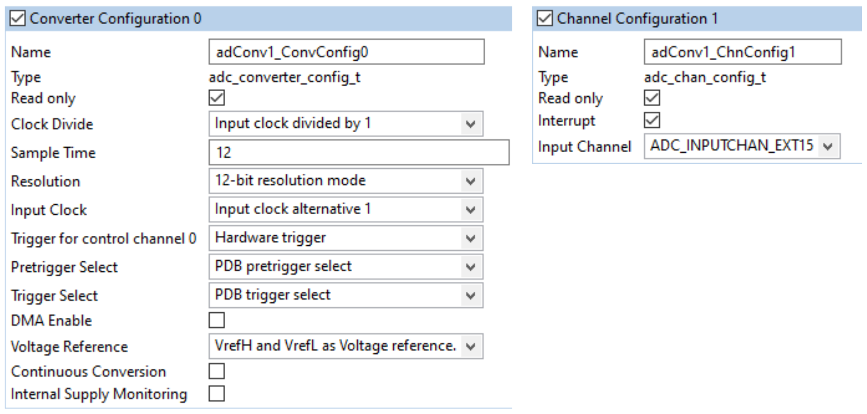
```
void McuPdbConfig(void)
{
    /* PDB0 module initialization */
    PDB_DRV_Init(INST_PDB0, &pdb0_InitConfig0);
    /* PDB1 module initialization */
    PDB_DRV_Init(INST_PDB1, &pdb1_InitConfig0);

    /* PDB0 CH0 pre-trigger0 initialization */
    PDB_DRV_ConfigAdcPreTrigger(INST_PDB0, 0, &pdb0_AdcTrigInitConfig0);
    /* PDB1 CH0 pre-trigger0 initialization */
    PDB_DRV_ConfigAdcPreTrigger(INST_PDB1, 0, &pdb1_AdcTrigInitConfig0);
    /* PDB1 CH0 pre-trigger1 initialization */
    PDB_DRV_ConfigAdcPreTrigger(INST_PDB1, 0, &pdb1_AdcTrigInitConfig1);

    /* Set PDB0 modulus value */
    PDB_DRV_SetTimerModulusValue(INST_PDB0, 5000);
    /* Set PDB1 modulus value */
    PDB_DRV_SetTimerModulusValue(INST_PDB1, 5000);
    /* PDB0 CH0 pre-trigger0 delay set to sense PhaseA stator current in the middle of the PWM cycle */
    PDB_DRV_SetAdcPreTriggerDelayValue(INST_PDB0, 0, 0, 2000);
    /* PDB1 CH0 pre-trigger0 delay set to sense DC bus voltage at the beginning of the PWM cycle */
    PDB_DRV_SetAdcPreTriggerDelayValue(INST_PDB1, 0, 0, 0);
    /* PDB1 CH0 pre-trigger2 delay set to sense PhaseB stator current in the middle of the PWM cycle */
    PDB_DRV_SetAdcPreTriggerDelayValue(INST_PDB1, 0, 1, 2000);

    /* Set PDB1 interrupt delay value */
    PDB_DRV_SetValueForTimerInterrupt(INST_PDB1, 4999);

    // enable PDB before LDOK
    PDB_DRV_Enable(INST_PDB0);
    // enable PDB before LDOK
    PDB_DRV_Enable(INST_PDB1);

    /* Load PDB0 configuration */
    PDB_DRV_LoadValuesCmd(INST_PDB0);
    /* Load PDB1 configuration */
    PDB_DRV_LoadValuesCmd(INST_PDB1);
}
```

4.2.5. Analog-to-Digital Converter (ADC)

The S32K144 device has two 12-bit Analog-to-Digital Converters (ADCs). These are 32-channel multiplexed input successive approximation ADCs with 16 result registers.

Both ADC instances are triggered independently by two PDBs. ADC channels are sampled in the order defined by PDB pre-triggers. When the first pre-trigger is asserted, associated lock of the pre-trigger becomes active waiting for the conversion complete flag COCO generated by the corresponding ADC channel. This sequence is repeated for each PDB pre-trigger and ADC channel couple.

Clock source of the ADC module is derived from the system clock frequency, further divided by 2 resulting 40MHz supply frequency. To combine high conversion resolution and short conversion time, 12-bit resolution mode with sample time 12 clock cycles are set in the *Converter Configuration* tab in the Processor Expert, *Figure 21*.



☑ Converter Configuration 0

Name	adConv1_ConvConfig0
Type	adc_converter_config_t
Read only	☑
Clock Divide	Input clock divided by 1
Sample Time	12
Resolution	12-bit resolution mode
Input Clock	Input clock alternative 1
Trigger for control channel 0	Hardware trigger
Pretrigger Select	PDB pretrigger select
Trigger Select	PDB trigger select
DMA Enable	☐
Voltage Reference	VrefH and VrefL as Voltage reference.
Continuous Conversion	☐
Internal Supply Monitoring	☐

☑ Channel Configuration 1

Name	adConv1_ChnConfig1
Type	adc_chan_config_t
Read only	☑
Interrupt	☑
Input Channel	ADC_INPUTCHAN_EXT15

Figure 21. **S32K144 ADC1 module and channels configuration in Processor Expert**

ADC0 measures only the Phase A stator current through *adc_ch0*. Other analog values, namely DC bus voltage, and Phase B stator current are measured by ADC1_CH0 and ADC1_CH1 respectively. Conversion complete interrupt is activated for ADC1_CH1 to invoke interrupt as soon as last conversion is completed. To measure Phase B stator current, *ADC_INPUTCHAN_EXT15* is selected as an input channel, *Figure 21*.

Example 7 shows ADC0 and ADC1 modules configuration. Processor Expert generates module configuration structures *adConvN_ConvConfigX* as well as channel configuration structures *adConvN_ChnConfigX*, which are present at the bottom of the example. These configuration structures take effect calling SDK APIs in *McuAdcConfig* function, *Example 7*.

Example 7. S32K144 ADC instances and channels controlled by S32 SDK

```
void McuAdcConfig(void)
{
    /* ADC0 module initialization */
    ADC_DRV_ConfigConverter(INST_ADCONV0, &adConv0_ConvConfig0);
    /* ADC1 module initialization */
    ADC_DRV_ConfigConverter(INST_ADCONV1, &adConv1_ConvConfig0);

    /* AD4 input channel is used for PhaseA stator current sensing */
    ADC_DRV_ConfigChan(INST_ADCONV0, 0, &adConv0_ChnConfig0);
    /* AD7 input channel is used for DC bus voltage sensing */
    ADC_DRV_ConfigChan(INST_ADCONV1, 0, &adConv1_ChnConfig0);
    /* AD15 input channel is used for PhaseB stator current sensing */
    ADC_DRV_ConfigChan(INST_ADCONV1, 1, &adConv1_ChnConfig1);
}

…

/*! adConv1 configuration structure */
const adc_converter_config_t adConv1_ConvConfig0 = {
  .clockDivide = ADC_CLK_DIVIDE_1,
  .sampleTime = 12U,
  .resolution = ADC_RESOLUTION_12BIT,
  .inputClock = ADC_CLK_ALT_1,
  .trigger = ADC_TRIGGER_HARDWARE,
  .pretriggerSel = ADC_PRETRIGGER_SEL_PDB,
  .triggerSel = ADC_TRIGGER_SEL_PDB,
  .dmaEnable = false,
  .voltageRef = ADC_VOLTAGEREF_VREF,
  .continuousConvEnable = false,
  .supplyMonitoringEnable = false,
};

const adc_chan_config_t adConv1_ChnConfig0 = {
  .interruptEnable = false,
  .channel = ADC_INPUTCHAN_EXT7,
};

const adc_chan_config_t adConv1_ChnConfig1 = {
  .interruptEnable = true,
  .channel = ADC_INPUTCHAN_EXT15,
};
```

4.2.6. Low Power Serial Peripheral Interface (LPSPI) and FETs pre-driver (MC34GD3000)

LPSPI is used as communication interface between S32K144 processor and analog FET pre-driver MC34GD3000. NXP's Three-Phase Brushless Motor Pre-Driver Software Driver (TPP), based on the S32 SDK is used to configure LPSPI of the S32K144 as well as MC34GD3000 properly. Included embedded driver provides access to all features of MC34GD3000 FETs driver such as writing/reading status registers, dead time insertion and fault protection.

Example 8 represents initialization of the S32K144 LPSPI0, MC34GD3000 and some important S32K144 GPIOs. TPP configures and later controls GPIO pins to enable/disable or reset MC34GD3000 in the application. Operation mode, deadtime and interrupt mask of the MC34GD3000 are specified at next paragraphs. Parameters, such as LPSPI instance, chip select pin are defined at bottom of the *Example 8*.

LPSPI0 communication frequency 2MHz is derived from the LPSPI0 input frequency 8MHz sourced from the system oscillator clock (SOSC_CLK).

GPIOs, LPSPI0 and MC34GD3000 are configured and enabled by *TPP_ConfigureGpio* and *TPP_ConfigureSpi*, *TPP_Init* functions, respectively.

Detailed description of the MC34GD3000 and its software driver (TPP) can be found at www.nxp.com.

Example 8. S32K144 LPSPI0 and MC34GD3000 controlled by TPP (S32 SDK)

```
void GD3000_Init(void)
{
    /* GD3000 pin configuration - EN1:PTA2 EN2:PTA2 & RST:PTA3 */
    tppDrvConfig.en1PinIndex     = 2U;
    tppDrvConfig.en1PinInstance = instanceA;
    tppDrvConfig.en2PinIndex     = 2U;
    tppDrvConfig.en2PinInstance = instanceA;
    tppDrvConfig.rstPinIndex     = 3U;
    tppDrvConfig.rstPinInstance = instanceA;

    /* GD3000 device configuration */
    tppDrvConfig.deviceConfig.deadtime =         INIT_DEADTIME;
    tppDrvConfig.deviceConfig.intMask0 =         INIT_INTERRUPTS0;
    tppDrvConfig.deviceConfig.intMask1 =         INIT_INTERRUPTS1;
    tppDrvConfig.deviceConfig.modeMask =         INIT_MODE;

    tppDrvConfig.deviceConfig.statusRegister[0U] = 0U;
    tppDrvConfig.deviceConfig.statusRegister[1U] = 0U;
    tppDrvConfig.deviceConfig.statusRegister[2U] = 0U;
    tppDrvConfig.deviceConfig.statusRegister[3U] = 0U;

    tppDrvConfig.csPinIndex =                    5U;
    tppDrvConfig.csPinInstance =                 instanceB;
    tppDrvConfig.spiInstance =                   0;
    tppDrvConfig.spiTppConfig.baudRateHz =       2000000U;
    tppDrvConfig.spiTppConfig.sourceClockHz =    8000000U;

    TPP_ConfigureGpio(&tppDrvConfig);
    TPP_ConfigureSpi(&tppDrvConfig, NULL);
    TPP_Init(&tppDrvConfig, tppModeEnable);
}
```

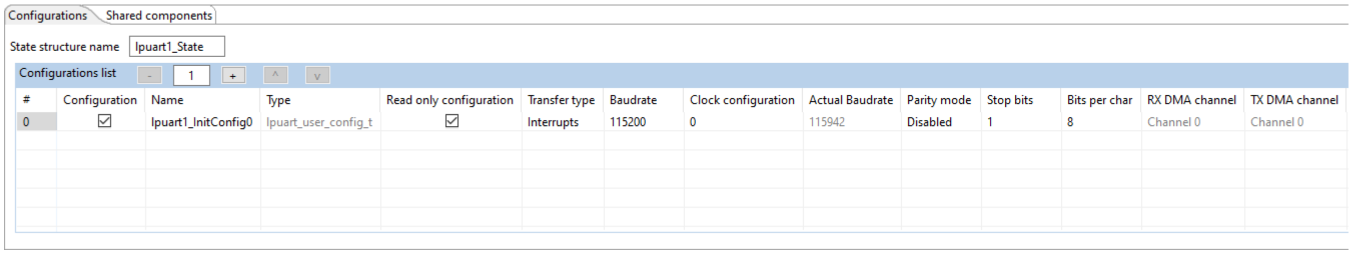
4.2.7. Low Power Universal Asynchronous Receiver/Transmitter (LPUART)

LPUART1 is used as a communication interface between S32K144 processor and FreeMASTER run-time debugging and visualization tool. Function *McuLpuartConfig* initializes LPUART1 module with baud rate 115200, 1 stop bit and 8 bits per channel. This configuration is carried out by SDK's LPUART driver, *Example 9*.

Example 9. S32K144 LPUART1 controlled by S32 SDK

```
void McuLpuartConfig(void)
{
    /* LPUART module initialization */
    LPUART_DRV_Init(INST_LPUART1, &lpuart1_State, &lpuart1_InitConfig0);
}

/*! lpuart1 configuration structure */
const lpuart_user_config_t lpuart1_InitConfig0 = {
  .transferType = LPUART_USING_INTERRUPTS,
  .baudRate = 115200U,
  .parityMode = LPUART_PARITY_DISABLED,
  .stopBitCount = LPUART_ONE_STOP_BIT,
  .bitCountPerChar = LPUART_8_BITS_PER_CHAR,
  .rxDMAChannel = 0U,
  .txDMAChannel = 0U,
};
```

Configuration structure *lpuart1_InitConfig0* can be modified manually or configured by means of Processor Expert as shown in *Figure 22*.


State structure name lpuart1_State

Configurations list 1

#	Configuration	Name	Type	Read only configuration	Transfer type	Baudrate	Clock configuration	Actual Baudrate	Parity mode	Stop bits	Bits per char	RX DMA channel	TX DMA channel
0	☑	lpuart1_InitConfig0	lpuart_user_config_t	☑	Interrupts	115200	0	115942	Disabled	1	8	Channel 0	Channel 0

Figure 22. **S32K144 LPUART1 module configuration in Processor Expert**

4.2.8. Port control and pin multiplexing

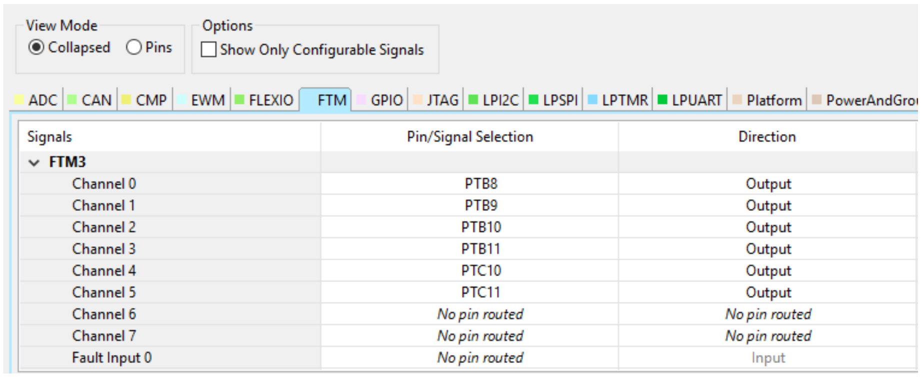
PMSM FOC Sensorless motor control application requires following on chip pins assignment, *Table 3*.

Table 3. **Pins assignment for S32K144 PMSM Sensorless FOC control**

Module	Signal name	Pin name / Functionality	Description
FTM3	PWMA_HS_B	PTB8 / FTM3_CH0	PWM signal for phase A high-side driver (inverted)
	PWMA_LS	PTB9 / FTM3_CH1	PWM signal for phase A low-side driver
	PWMB_HS_B	PTB10 / FTM3_CH2	PWM signal for phase B high-side driver (inverted)
	PWMB_LS	PTB11 / FTM3_CH3	PWM signal for phase B low-side driver
	PWMC_HS_B	PTC10 / FTM3_CH4	PWM signal for phase C high-side driver (inverted)
	PWMC_LS	PTC11 / FTM3_CH5	PWM signal for phase C low-side driver
FTM2	ENC_A	PTD11 / FTM2_QD_PHA	Phase A signal of the Encoder sensor
	ENC_B	PTD10 / FTM2_QD_PHB	Phase B signal of the Encoder sensor
ADC0	PHA_I	PTB0 / ADC0_SE4	Phase A stator current measurement
	PHC_I	PTA6 / ADC0_SE2	Phase C stator current measurement. Available on pin, but not used in this app. For more details see chapter 3.3
ADC1	DCBI	PTD4 / ADC1_SE6	DC bus current measurement Available on pin, but not used in this app. For more details see chapter 3.3
	DCBV	PTB12 / ADC1_SE7	DC bus voltage measurement
	PHB_I	PTB1 / ADC1_SE15	Phase B stator current measurement
LPSPI0	SPI_SCLK	PTB2 / LPSPI0_SCK	SPI clock (2MHz)
	SPI_MISO	PTB3 / LPSPI0_SIN	SPI input data from GD3000
	SPI_MOSI	PTB4 / LPSPI0_SOUT	SPI output data for GD3000
LPUART1	SDA_SPI0_SOUT	PTC6 / LPUART1_RX	UART transmit data (FreeMASTER)
	SDA_SPI0_SIN	PTC7 / LPUART1_TX	UART receive data (FreeMASTER)

Module	Signal name	Pin name / Functionality	Description
TRGMUX	PTD1	PTD1 / TRGMUX_OUT2	FTM3 initialization trigger
	PTA0	PTA0 / TRGMUX_OUT3	PBD1 channel 0 trigger output
	PTE15	PTE15 / TRGMUX_OUT6	ADC1 conversion complete flag
GPIO	GD_EN	PTA2 / PTA2	Enable signal for GD3000
	GD_RST_B	PTA3 / PTA3	Reset signal for GD3000
	SPI_CS_B	PTB5 / PTB5	Chip select signal for GD3000
	BTN0	PTC12 / PTC12	Application control via board button
	BTN1	PTC13 / PTC13	Application control via board button
	RGB_BLUE	PTD0 / PTD0	RGB_BLUE indicating run state
	PTD2	PTD2 / PTD2	GPIO toggling to measure execution time
	BRAKE_PWM	PTD14 / PTD14	Connecting / disconnecting braking resistor
	RGB_RED	PTD15 / PTD15	RGB_RED indicating fault state
	RGB_GREEN	PTD16 / PTD16	RGB_GREEN indicating ready/calib state
	GD_INT	PTE10 / PTE10	Interrupt signal indicating GD3000 fault

This pins assignment can be carried out by means of Processor Expert opening *pin_mux:PinSetting* component. Pin assignment of the FTM3 module is shown in *Figure 23* as an example.


View Mode
● Collapsed ○ Pins
Options
☐ Show Only Configurable Signals
ADC CAN CMP EWM FLEXIO FTM GPIO JTAG LPI2C LPSPI LPTMR LPUART Platform PowerAndGro

Signals	Pin/Signal Selection	Direction
FTM3		
Channel 0	PTB8	Output
Channel 1	PTB9	Output
Channel 2	PTB10	Output
Channel 3	PTB11	Output
Channel 4	PTC10	Output
Channel 5	PTC11	Output
Channel 6	*No pin routed*	*No pin routed*
Channel 7	*No pin routed*	*No pin routed*
Fault Input 0	*No pin routed*	Input

Figure 23. **S32K144 Pins assignment for FTM3 in Processor Expert**

Once the pins are properly assigned meaning functionality for each pin is selected, Processor Expert generates array of the configuration structures *g_pin_mux_InitConfigArr* that individually accesses Pin Control Register PCR and GPIO registers.

One of the configuration structure is shown at bottom of *Example 10*. It defines that PTE10 pin works as GPIO with input direction. In addition, interrupt on rising edge is enabled to be able to detect and monitor fault conditions of the MC34GD3000 FET pre-driver, see chapter *4.1.2*.

Pins of the S32K144 are configured calling *PINS_DRV_Init* function at the top of the *Example 10*.

Example 10. S32K144 pins configuration controlled by S32 SDK

```
void McuPinsConfig(void)
{
    /* MCU Pins configuration */
    PINS_DRV_Init(NUM_OF_CONFIGURED_PINS, g_pin_mux_InitConfigArr);

}


pin_settings_config_t g_pin_mux_InitConfigArr[NUM_OF_CONFIGURED_PINS] =
{
```

```
...

    {
        .base          = PORTE,
        .pinPortIdx    = 10u,
        .pullConfig    = PORT_INTERNAL_PULL_NOT_ENABLED,
        .passiveFilter = false,
        .driveSelect   = PORT_LOW_DRIVE_STRENGTH,
        .mux           = PORT_MUX_AS_GPIO,
        .pinLock       = false,
        .intConfig     = PORT_INT_RISING_EDGE,
        .clearIntFlag  = false,
        .gpioBase      = PTE,
        .direction     = GPIO_INPUT_DIRECTION,
        .digitalFilter = false,
    },
...

}
```

4.3. Software architecture

4.3.1. Introduction

This section describes the software design of the Sensorless PMSM Field Oriented Control framework application. The application overview and description of software implementation are provided. The aim of this chapter is to help in understanding of the designed software.

4.3.2. Application data flow overview

The application software is interrupt driven running in real time. There is one periodic interrupt service routine associated with the ADC conversion complete interrupt, executing all motor control tasks. This includes both fast current and slow speed loop control. All tasks are performed in an order described by the application state machine shown in *Figure 26*, and application flowcharts shown in *Figure 24* and *Figure 25*.

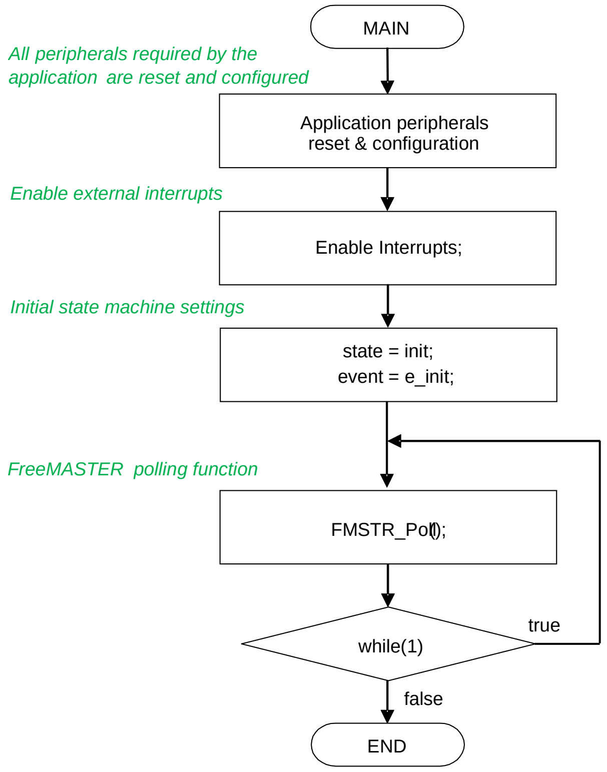

MAIN
All peripherals required by the application are reset and configured
Application peripherals reset & configuration
Enable external interrupts
Enable Interrupts;
Initial state machine settings
state = init;
event = e_init;
FreeMASTER polling function
FMSTR_Poll();
while(1)
true
false
END

Figure 24. **Flow chart diagram of main function with background loop.**

To achieve precise and deterministic sampling of analog quantities and to execute all necessary motor control calculations, the state machine functions are called within a periodic interrupt service routine. Hence, in order to actually call state machine functions, the peripheral causing this periodic interrupt must be properly configured and the interrupt enabled. As described in section *S32K144 Device initialization*, all peripherals are initially configured and all interrupts are enabled after a RESET of the device. As soon as interrupts are enabled and all S32K144 peripherals are correctly configured, the state machine functions are called from the ADC end of sequence interrupt service routine. The background loop handles non-critical timing tasks, such as the FreeMASTER communication polling.

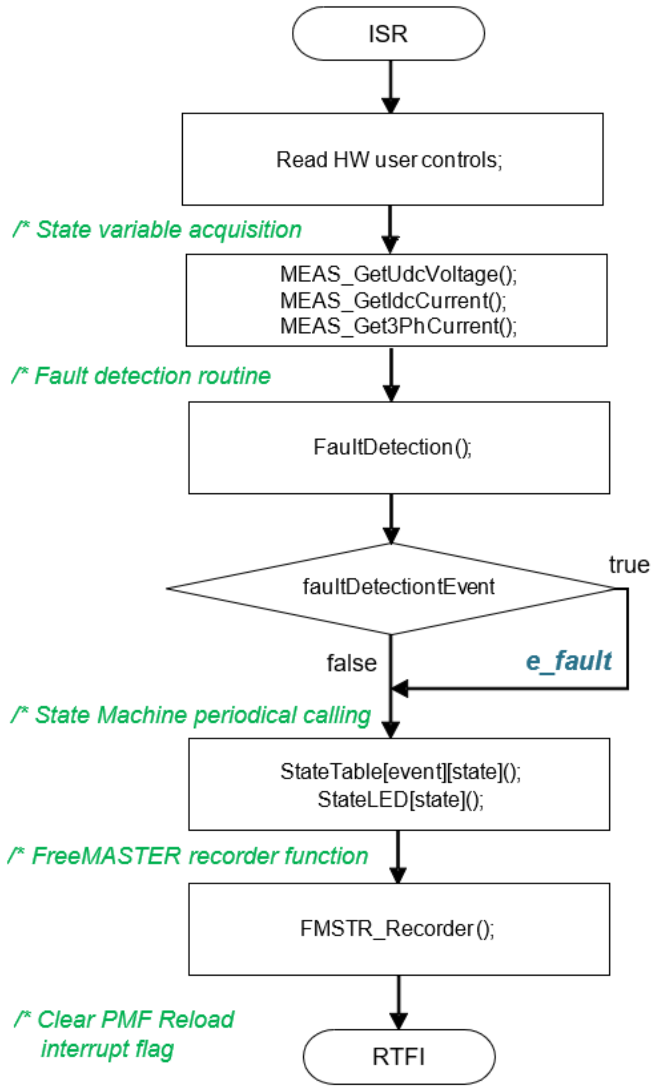

ISR
Read HW user controls;
/* State variable acquisition
MEAS_GetUdcVoltage();
MEAS_GetIdcCurrent();
MEAS_Get3PhCurrent();
/* Fault detection routine
FaultDetection();
faultDetectiontEvent
true
false
e_fault
/* State Machine periodical calling
StateTable[event][state]();
StateLED[state]();
/* FreeMASTER recorder function
FMSTR_Recorder();
/* Clear PMF Reload interrupt flag
RTFI

Figure 25. **Flow chart diagram of periodic interrupt service routine.**

4.3.3. State machine

The application state machine is implemented using a two-dimensional array of pointers to the functions using variable called *StateTable[][]().* The first parameter describes the current application event, and the second parameter describes the actual application state. These two parameters select a particular pointer to state machine function, which causes a function call whenever *StateTable[][]()* is called.

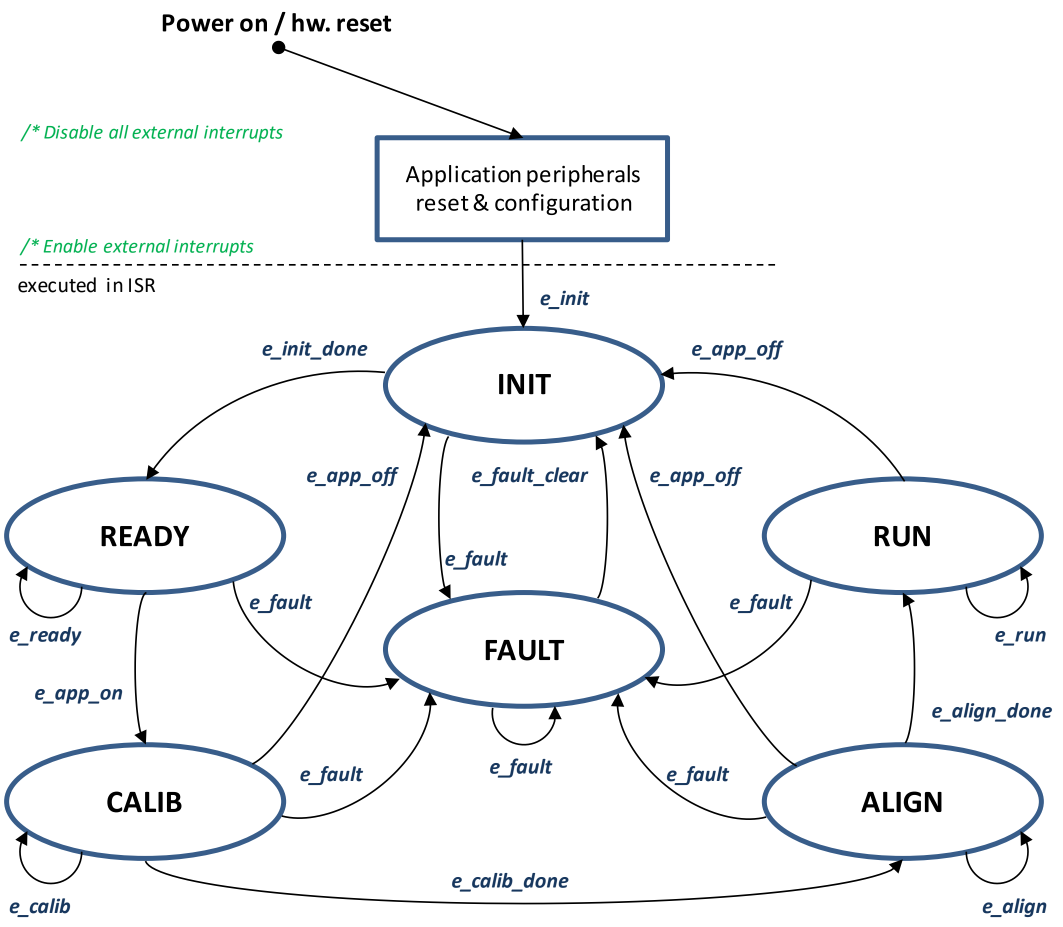


Figure 26. **Application state machine**

The application state machine consists of following six states, which are selected using variable state defined as:

AppStates:

- INIT - state = 0
- FAULT - state = 1
- READY - state = 2
- CALIB - state = 3
- ALIGN - state = 4
- RUN - state = 5

To signalize/initiate a change of state, eleven events are defined, and are selected using variable event defined as:

AppEvents:

- e_fault - event = 0
- e_fault_clear - event = 1
- e_init - event = 2
- e_init_done - event = 3
- e_ready - event = 4
- e_app_on - event = 5
- e_app_off - event = 11
- e_calib - event = 6
- e_calib_done - event = 7
- e_align - event = 8
- e_align_done - event = 9
- e_run - event = 10

4.3.3.1. State – FAULT

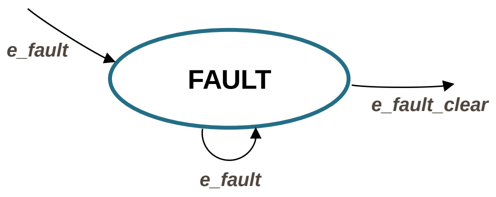


Figure 27. **FAULT state with transitions**

The application goes immediately to this state when a fault is detected. The system allows all states to pass into the FAULT state by setting *cntrState.event* = *e_fault*. State FAULT is a state that transitions back to itself if the fault is still present in the system and the user does not request clearing of fault flags. There are two different variables to signal fault occurrence in the application. The warning register *tempFaults* represents the current state of the fault pin/variable to warn the user that the system is getting close to its critical operation. And the fault register *permFaults* represents a fault flag, which is set and put the application immediately to fault state. Even if fault source disappears, the fault remains set until manually cleared by the user. Such mechanisms allow for stopping the application and analyzing the cause of failure, even if the fault was caused by a short glitch on monitored pins/variables. State FAULT can only be left when application variable *switchFaultClear* is manually set to *true* (using FreeMASTER) or by simultaneously pressing the user buttons (BTN0 and BTN1) on the S32K144EVB evaluation board. That is, the user has acknowledged that the fault source has been removed and the application can be restarted. When the user sets *switchFaultClear* = *true*; the following sequence is automatically executed, *Example 11*.

Example 11. Fault clearing sequence

```
if (cntrState.usrControl.switchFaultClear)
{
    // Clear permanent and temporary SW faults
    permFaults.mcu.R            = 0;                 // Clear mcu faults
    permFaults.motor.R          = 0;                 // Clear motor faults
    permFaults.stateMachine.R   = 0;                 // Clear state machine faults
    gd3000Status.B.gd3000ClearErr = true;            // Clear GD3000 faults
    pdbStatus.PDB0_SeqErrFlags   = 0;                // Clear PDB0 sequence error flags
    pdbStatus.PDB1_SeqErrFlags   = 0;                // Clear PDB1 sequence error flags

    // When all Faults cleared prepare for transition to next state.
    cntrState.usrControl.readFault              = true;
    cntrState.usrControl.switchFaultClear       = false;
    cntrState.event                             = e_fault_clear;

    // Read ADCs Results registers to unlock PDB pre-triggers lock states
    ADC_DRV_GetChanResult(INST_ADCONV0, 0, &adc_r);
    ADC_DRV_GetChanResult(INST_ADCONV1, 0, &adc_r);
    ADC_DRV_GetChanResult(INST_ADCONV1, 1, &adc_r);

    // Enable FTM init trigger for PDBs after cleared PDBs sequence errors
    // and unlocked PDBs pre-triggers
    FTM_RMW_EXTTRIG_REG(FTM3, 0x00, 0x40);
}
```

Setting event to *cntrState.event* = *e_fault_clear* while in FAULT state represents a new request to proceed to INIT state. This request is purely user action and does not depend on actual fault status. In other words, it is up to the user to decide when to set *switchFaultClear* true. However, according to the interrupt data flow diagram shown in *Figure 25*, function *faultDetection()* is called before state machine function *state_table[event][state]()*. Therefore, all faults will be checked again and if there is any fault condition remaining in the system, the respective bits in *permFaults* and *tempFaults* variables will be set. As a consequence of *permFaults* not equal to zero, function *faultDetection()* will modify the application event from *e_fault_clear* back to *e_fault*, which means jump to fall state when state machine function *state_table[event][state]()* is called. Hence, INIT state will not be entered even though the user tried to clear the fault flags using *switchFaultClear*. When the next state (INIT) is entered, all fault bits are cleared, which means no fault is detected (*permFaults* = 0x0) and application variable *switchFaultClear* is manually set to true.

The application is scanning for following system warnings and errors:

- DC bus over voltage
- DC bus under voltage
- DC bus over current
- Phase A and phase B over current

The thresholds for fault detection can be modified in INIT state. Please see MCAT Settings and Tuning for further information on how to set these thresholds using the MCAT. In addition, fault state is entered if following errors are detected:

- PDB Errors (PDB Sequence error)
- GD3000 pre-driver errors (overtemperature, desaturation fault, low supply voltage, DC bus overcurrent, phase error, framing error, write error after block, existing reset). See section *References*.
- FOC Error (irrelevant event call in state machine or Back-EMF failure)

4.3.3.2. State – INIT

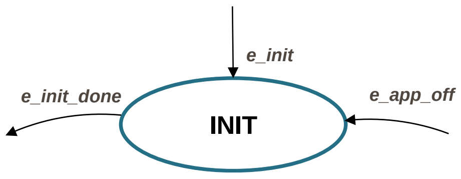


Figure 28. **INIT state with transitions**

State INIT is "one pass" state/function, and can be entered from all states except for READY state, provided there are no faults detected. All application state variables are initialized in state INIT.

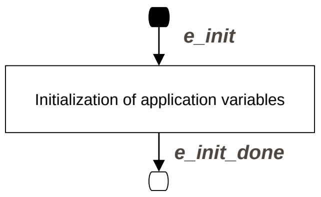


Figure 29. **Flow chart of state INIT**

After the execution of INIT state, the application event is automatically set to *cntrState.event=e_init_done*, and state READY is selected as the next state to enter.

4.3.3.3. State – READY

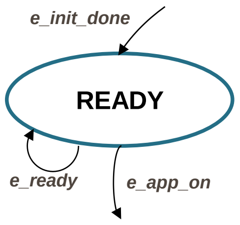


Figure 30. **READY state with transitions**

In READY state, application is waiting for user command to start the motor. The application is released from waiting mode by pressing the on board button BTN0 or BTN1 or by FreeMASTER interface setting the variable *switchAppOnOff* = *true* (see flow chart in *Figure 31*).

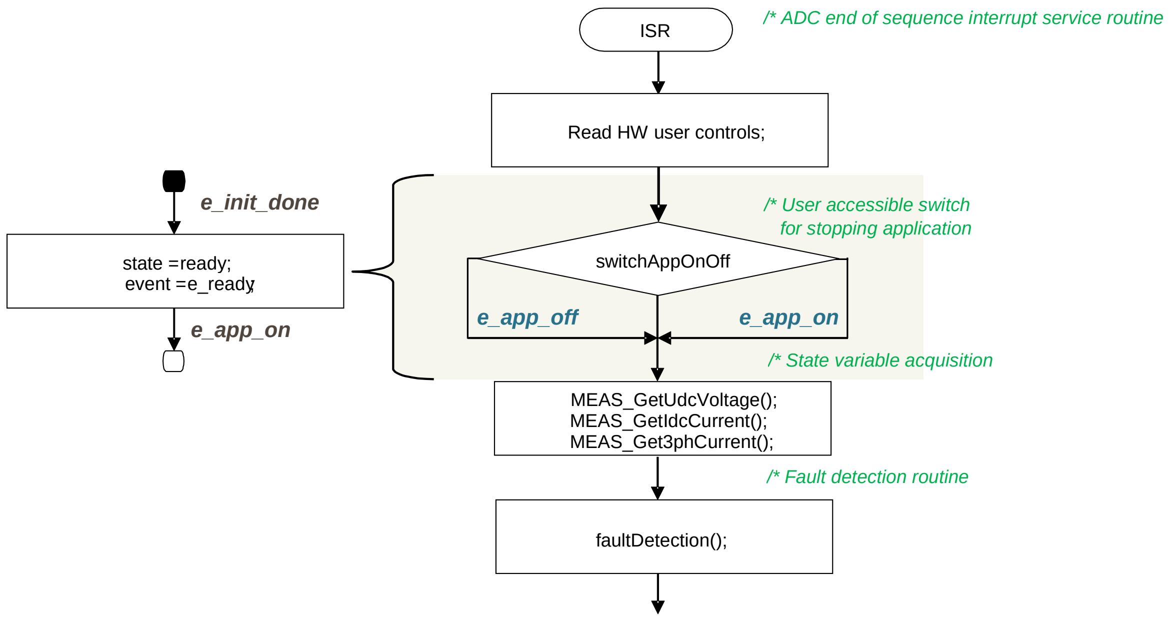


Figure 31. **Flow chart of state READY**

4.3.3.4. State – CALIB

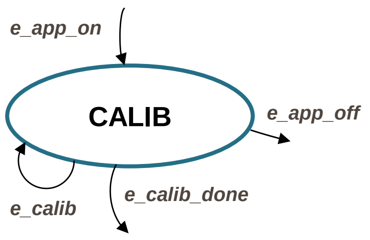


Figure 32. **CALIB state with transitions**

In this state, ADC DC offset calibration is performed. Once the state machine enters CALIB state, all PWM output are enabled. Calibration of the DC offset is achieved by generating 50% duty-cycle on the PWM outputs, and taking several measurements of the ADC0 and ADC1 channels connected to the current sensors. These measurements are then averaged, and the average value for the channel is stored. This value will be subtracted from the measured value when in normal operation. This way the half range DC offset, caused by voltage shift of 2.5V in conditional circuitry (see *Figure 5*), is removed in the measured phase. State CALIB is a state that allows transition back to itself, provided no faults are present, the user does not request stop of the application (by *switchAppOnOff=true*), and the calibration process has not finished. The number of samples for averaging is set by default to 2^10=1024, and can be modified in the state INIT. After all 1024 samples have been taken and the averaged values successfully saved, the application event is automatically set to *cntrState.event=e_calib_done* and state machine can proceed to state ALIGN (see flow chart in *Figure 33*).

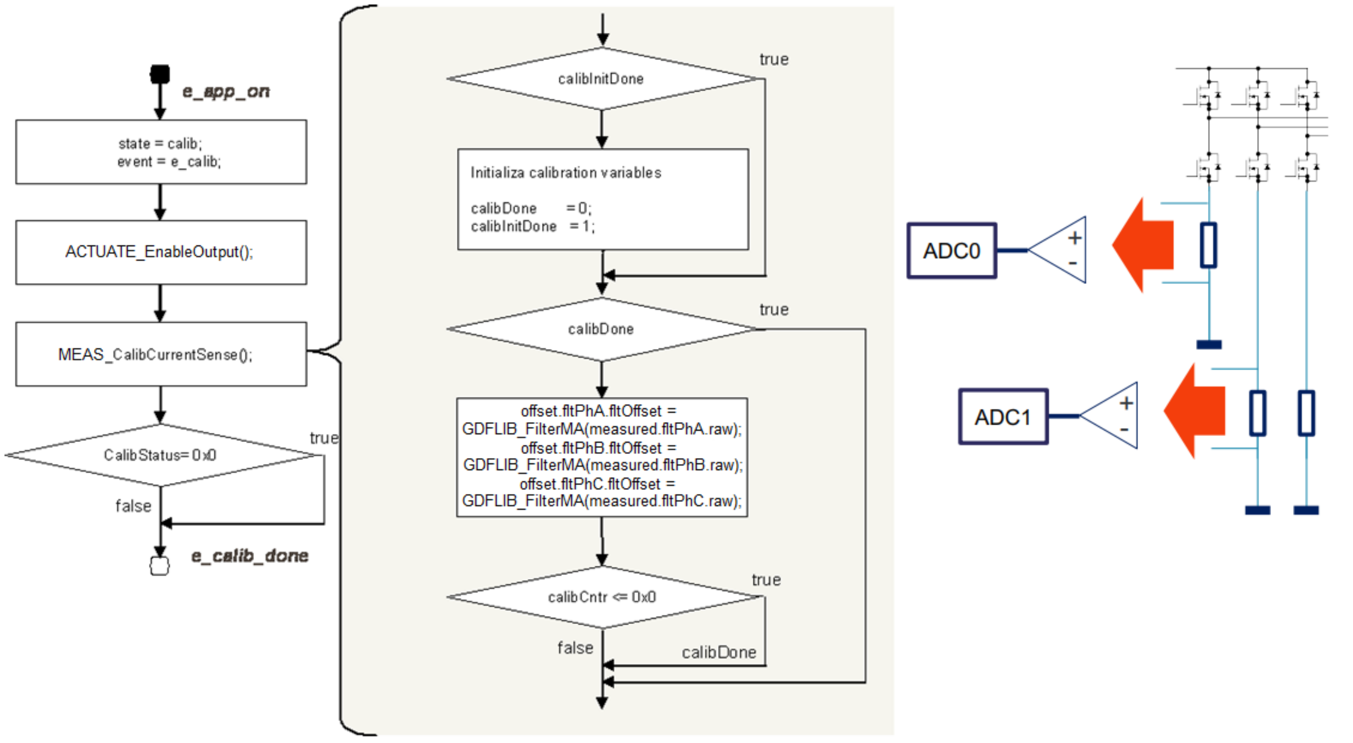

e_app_on
state = calib;
event = e_calib;
ACTUATE_EnableOutput();
MEAS_CalibCurrentSense();
CalibStatus=0x0
true
false
e_calib_done
calibInitDone
true
Initializa calibration variables
calibDone = 0;
calibInitDone = 1;
calibDone
true
offset.fltPhA.fltOffset = GDFLIB_FilterMA(measured.fltPhA.raw);
offset.fltPhB.fltOffset = GDFLIB_FilterMA(measured.fltPhB.raw);
offset.fltPhC.fltOffset = GDFLIB_FilterMA(measured.fltPhC.raw);
calibCntr <= 0x0
true
false
calibDone
ADC0
ADC1
+
-

Figure 33. **Flow chart of state CALIB**

A transition to FAULT state is performed automatically when a fault occurs. A transition to INIT state is performed by setting the event to *cntrState.event=e_app_off*, which is done automatically on falling edge of *switchAppOnOff=false* using FreeMASTER.

4.3.3.5. State – ALIGN

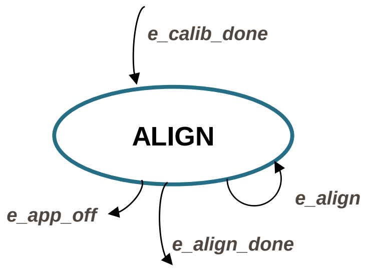


Figure 34. **ALIGN state with transitions**

This state shows alignment of the rotor and stator flux vectors to mark zero position. When using a model based approach for position estimation, the zero position is not known. The zero position is obtained at ALIGN state, where a DC voltage is applied to d-axis voltage for a certain period. This will cause the rotor to rotate to "align" position, where stator and rotor fluxes are aligned. The rotor position in which the rotor stabilizes after applying this DC voltage is set as zero position. In order to wait for rotor to stabilize in an aligned position, a certain time period is selected during which the DC voltage is constantly applied. The period of time and the amplitude of DC voltage can be modified in INIT state. Timing is implemented using a software counter that counts from a pre-defined value down to zero. During this time, the event remains set to *cntrState.event=e_align*. When the counter reaches zero, the counter is reset back to the pre-defined value, and event is automatically set to *cntrState.event=e_align_done*. This enables a transition to RUN state see flow chart in *Figure 35*.

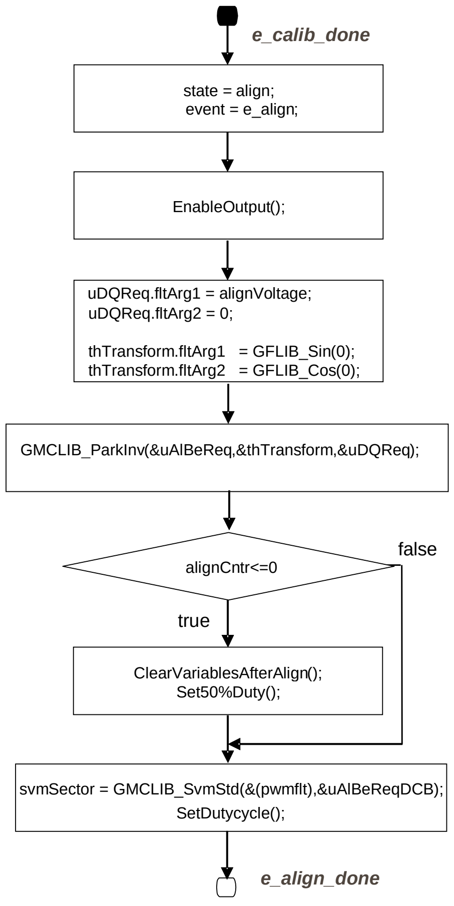

e_calib_done
state = align;
event = e_align;
EnableOutput();
uDQReq.fltArg1 = alignVoltage;
uDQReq.fltArg2 = 0;
thTransform.fltArg1 = GFLIB_Sin(0);
thTransform.fltArg2 = GFLIB_Cos(0);
GMCLIB_ParkInv(&uAlBeReq,&thTransform,&uDQReq);
alignCntr<=0
false
true
ClearVariablesAfterAlign();
Set50%Duty();
svmSector = GMCLIB_SvmStd(&(pwmflt),&uAlBeReqDCB);
SetDutycycle();
e_align_done

Figure 35. **Flow chart of state ALIGN**

A transition to FAULT state is performed automatically when a fault occurs. Transition to INIT state is performed by setting the event to *cntrState.event=e_app_off*, which is done automatically on falling edge of *switchAppOnOff=false* using FreeMASTER or using the switch.

4.3.3.6. State – RUN

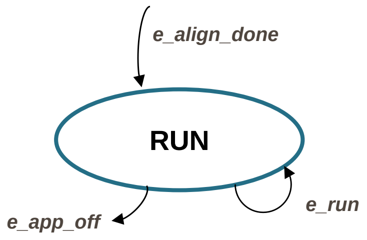


Figure 36. **RUN state with transitions**

In this state, the FOC algorithm is calculated, as described in section *PMSM field oriented control*.

The control is designed such that the drive might be operated in four modes depending on the source of the position information:

1. **Force mode**: The FOC control is based on the generated position (so called open loop position), also this position is supplied to eBEMF observer in order to initialize its state.
2. **Tracking mode:** The FOC control is still using the open loop position, however, the eBEMF observer is left on its own, meaning that the observer is using its own estimated position and speed one calculation step delayed.
3. **Sensorless mode:** Both FOC control and eBEMF observer using estimated position.
4. **Encoder mode:** FOC control uses position and speed obtained from Encoder sensor. This mode is available only if ENCODER macro is set to *true.*

Position mode can be controlled by *pos_mode* variable in FreeMASTER interface. It might be modified manually or automatically depending on the state of the variable *cntrState.usrControl.controlMode*. If *cntrState.usrControl.controlMode = automatic* and *switchSensor = Sensorless*, application automatically transits from Force mode (open loop mode) to Sensorless mode (closed loop mode) through Tracking mode based on the actual rotor speed and speed limits defined for each position mode (see section 3.4). Variable *switchSensor* defines whether position/speed feedback comes from back-EMF Observer or Encoder sensor. It is automatically set to *Sensorless*, if Encoder sensor is not present (ENCODER=*false*).

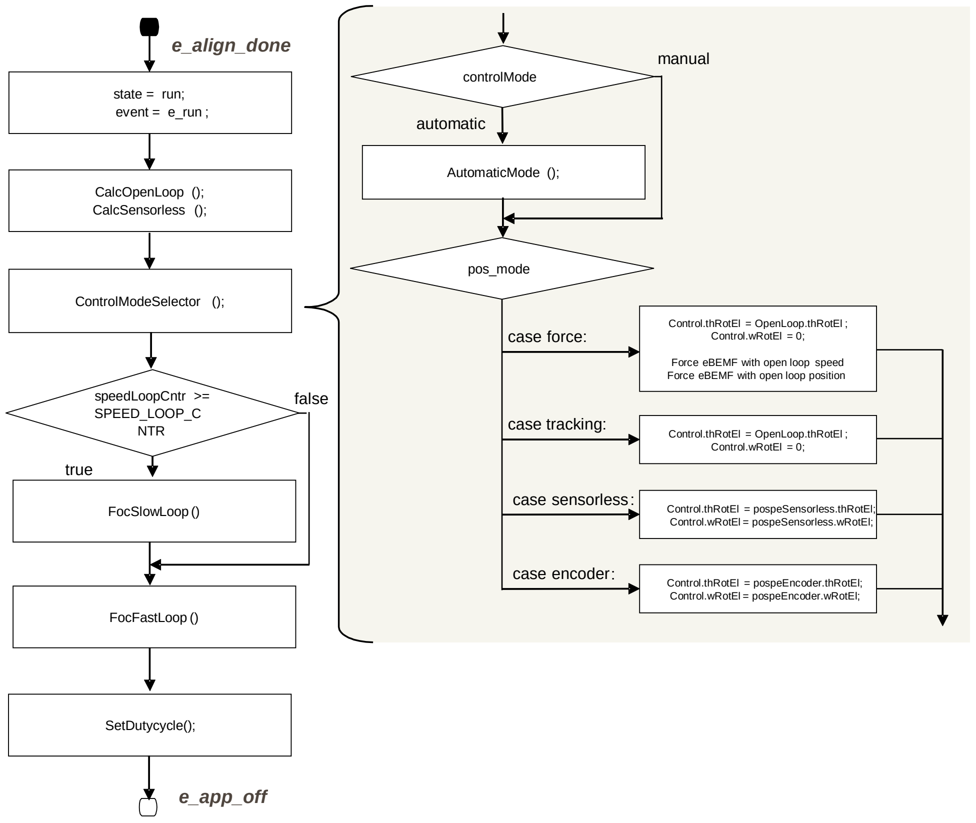

e_align_done
state = run;
event = e_run ;
CalcOpenLoop ();
CalcSensorless ();
ControlModeSelector ();
speedLoopCntr >= SPEED_LOOP_CNTR
false
true
FocSlowLoop ()
FocFastLoop ()
SetDutycycle();
e_app_off
controlMode
manual
automatic
AutomaticMode ();
pos_mode
case force:
Control.thRotEl = OpenLoop.thRotEl ;
Control.wRotEl = 0;
Force eBEMF with open loop speed
Force eBEMF with open loop position
case tracking:
Control.thRotEl = OpenLoop.thRotEl ;
Control.wRotEl = 0;
case sensorless:
Control.thRotEl = pospeSensorless.thRotEl;
Control.wRotEl = pospeSensorless.wRotEl;
case encoder:
Control.thRotEl = pospeEncoder.thRotEl;
Control.wRotEl = pospeEncoder.wRotEl;

Figure 37. **Flow chart of state RUN**

Calculation of fast current loop is executed every ADC end of sequence interrupt, while calculation of slow speed loop is executed every Nth ADC end of sequence interrupt. Arbitration is done using a counter that counts from value N down to zero. When zero is reached, the counter is reset back to N and slow speed loop calculation is performed. This way, only one interrupt is needed for both loops and timing of both loops is synchronized. Slow loop calculations are finished before entering fast loop calculations (see flow chart in *Figure 37*).

Figure 38 shows implementation of FOC algorithm and used functions and variables. As can be seen from the diagram, rotor position and speed are estimated by eBEMF observer. This is a default rotor position and speed feedback for FOC. To test Encoder based FOC, ENCODER macro must be set to *true* and PM motor provided with this motor control kit replaced by PM motor of the comparable power and equipped with Encoder sensor. As mentioned previously, Encoder based FOC can be activated/deactivated by setting *switchSensor* variable to *encoder/sensorless*.

A transition from RUN state to FAULT state is performed automatically when a fault occurs. A transition to INIT state is performed by setting the event to *cntrState.event=e_app_off*, which is done automatically on falling edge of *switchAppOnOff=false* using FreeMASTER or keeping user buttons BTN0 and BTN1 pressed.

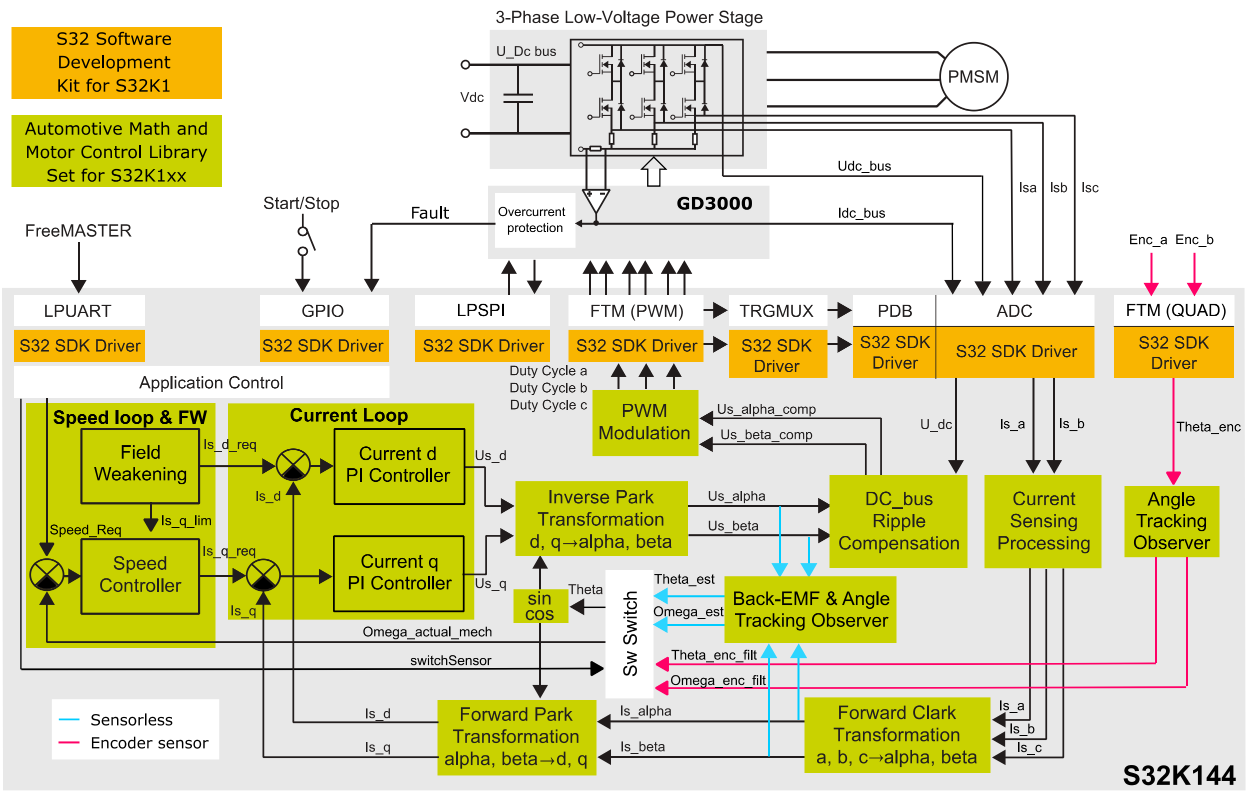


Figure 38. **Sensorless and Sensorbased FOC with FW implementation on S32K144**

4.3.4. AMMCLIB Integration

Application software of the FOC Sensorless control with field weakening is built using NXP’s Automotive Math and Motor Control Library set (AMMCLIB), a precompiled, highly speed-optimized off-the-shelf software library designed for motor control applications. The most essential blocks of the

FOC structure are presented in *Figure 38*. AMMCLIB supports all available data type implementations: 32-bit fixed-point, 16-bit fixed-point and single precision floating-point. In order to achieve high performance of the S32K144 core, floating point arithmetic is used as a reference for this motor control application.

Current Loop function AMCLIB_CurrentLoop unites and optimizes most inner loop of the FOC cascade structure *Figure 38*. It consists of two PI controllers and basic mathematical operations which calculate errors between required and feedback currents and limits for PI controllers based on the actual value of the DC bus voltage. All functions and data structures are presented in *Figure 39*.

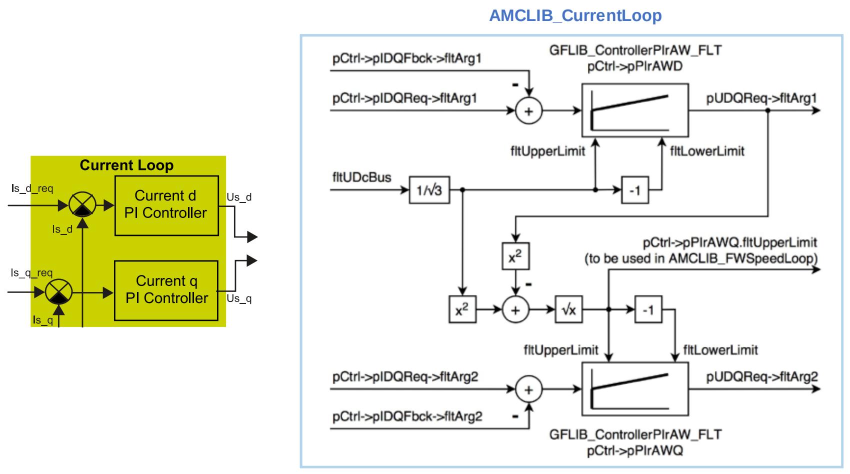


Figure 39. **Functions and data structures in AMCLIB_CurrentLoop**

Required d- and q-axis stator currents can be either manually modified or generated by outer loop of the cascade structure consisting of: Speed Loop and Field Weakening (FW) as shown in *Figure 38*. To achieve highly optimized level, AMCLIB_FWSpeedLoop merges two functions of the AMMCLIB, namely speed control loop AMCLIB_SpeedLoop and field weakening control AMCLIB_FW, *Figure 40*. AMCLIB_SpeedLoop consists of speed PI controller GFLIB_ControllerPIpAW, speed ramp GFLIB_Ramp placed in feedforward path and exponential moving average filter GFLIB_FilterMA placed in the speed feedback. AMCLIB_FW function is NXP's patented algorithm (US Patent No. US 2011/0050152 A1) that extends the speed range of PMSM beyond the base speed by reducing the stator magnetic flux linkage as discussed in section *3.5*. All functions and data structures used in AMCLIB_FW function are shown in *Figure 40*.

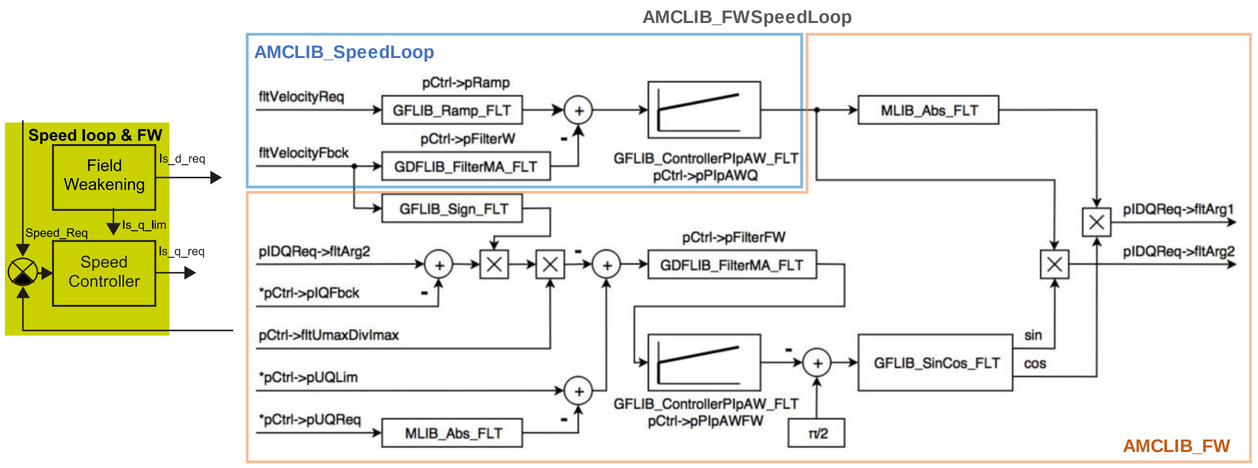


Figure 40. **Functions and data structures in AMCLIB_FWSpeedLoop**

AMCLIB_FW key advantages:

- Fully utilize the drive capabilities (speed range, load torque)
- Reduces stator linkage flux only when necessary
- Supports four quadrant operations
- The algorithm is very robust - as a result, the PMSM behaves as a separately excited wound field synchronous motor drive
- Allows maximum torque optimal control

Back-EMF observer AMCLIB_BemfObsrv and Angle tracking observer AMCLIB_TrackObsrv constitute important blocks in this application, *Figure 38*. They estimate rotor position and speed based on the inputs, namely stator voltages $u_{\alpha\beta}$ and currents $i_{\alpha\beta}$, *Figure 41*. AMCLIB_BemfObsrv transforms inputs quantities from stationary reference frame α/β to quasi-synchronous reference frame γ/δ that follows the real synchronous rotor flux frame d/q with an error θ_{err}. AMCLIB_BemfObsrv algorithm is based on the mathematical model of the PMSM motor with excluded back-EMF terms $e_{\gamma\delta}$. Back-EMF terms are estimated as disturbances in this model, generated by PI controllers. The estimated BEMF values are used for calculating the phase error θ_{err}, which is provided as an output of the BEMF observer.

To align both frames and provide accurate estimates, this phase error θ_{err} must be driven to zero. This is a main role of the Angle tracking observer AMCLIB_TrackObsrv which is attached to function of the back-EMF observer AMCLIB_BemfObsrv, *Figure 41*. AMCLIB_TrackObsrv is an adopted phase-locked-loop algorithm that estimates rotor speed and position keeping $\theta_{err} = 0$. This is ensured by a loop compensator that is PI controller. While PI controller generates estimated rotor speed, integrator used in this phase-locked-loop algorithm serves estimated rotor position.

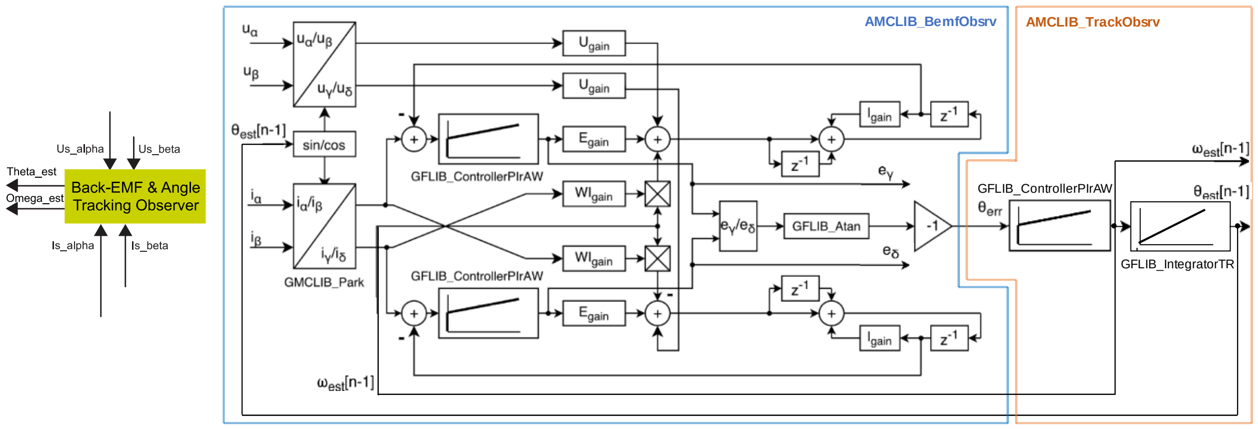


Figure 41. **Structure of the AMCLIB_BemfObsrv and AMCLIB_TrackObsrv**

More details related to AMMCLIB FOC functions can be found in S32K14x AMMCLIB User’s manual (see section *References*). Parameters of the PI controllers placed in the speed control loop, current control loop, back-EMF and Angle tracking observer can be tuned by using NXP’s Motor Control Application Tuning tool (MCAT). Detailed instructions on how to tune parameters of the FOC structure by MCAT are presented in AN4912, AN4642 (see section *References*).

4.3.5. MCAT Integration

MCAT (Motor Control Application Tuning) is a graphical tool dedicated to motor control developers and the operators of modern electrical drives. The main feature of proposed approach is automatic calculation and real-time tuning of selected control structure parameters. Connecting and tuning new electric drive setup becomes easier because the MCAT tool offers a possibility to split the control structure and consequently to control the motor at various levels of cascade control structure.

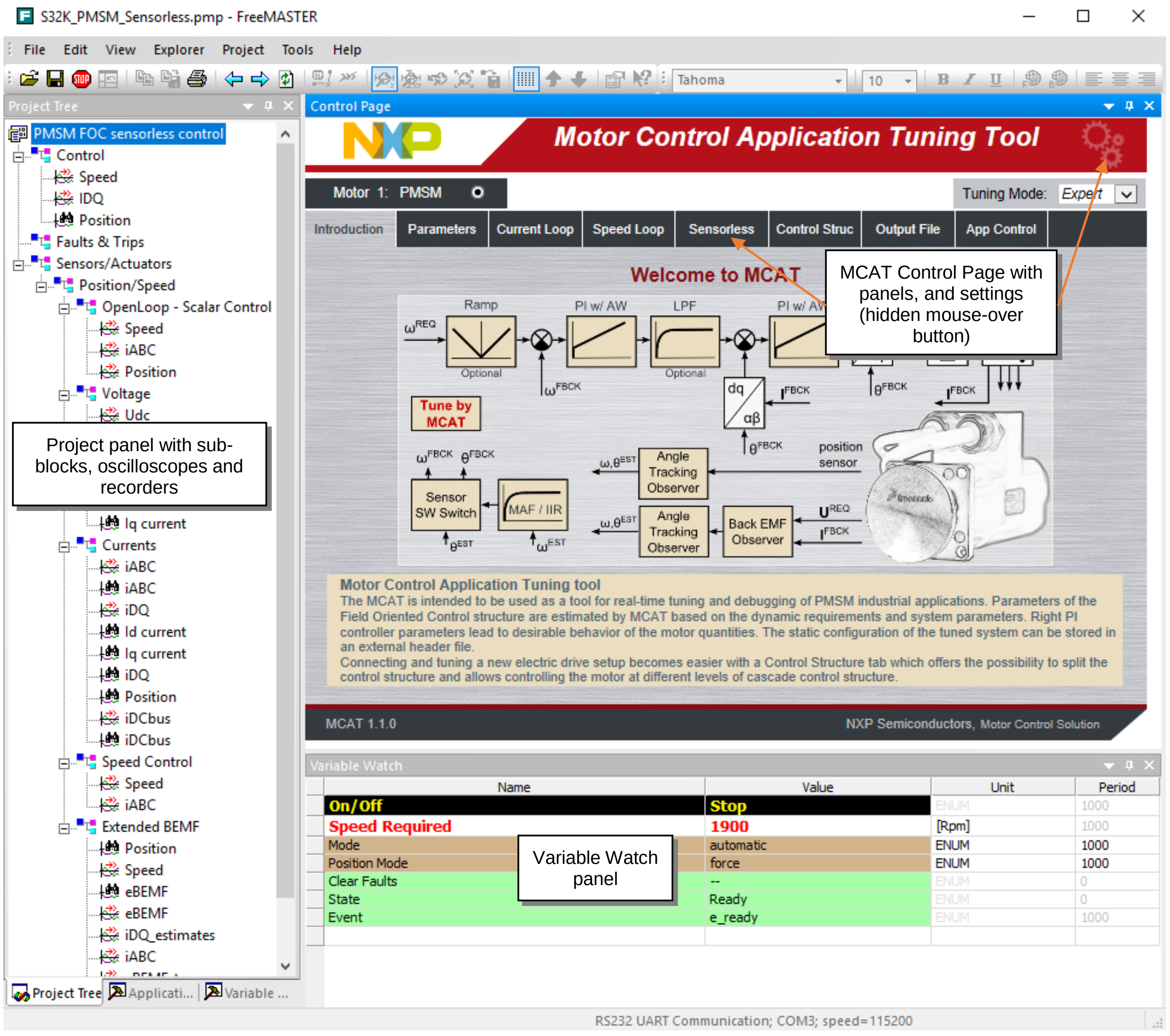
The MCAT tool runs under FreeMASTER online monitor, which allows the real-time tuning of the motor control application. Respecting the parameters of the controlled drive, the correct values of control structure parameters are calculated, which can be directly updated to the application or stored in an application static configuration file. The electrical subsystems are modeled using physical laws and parameters of the PI controllers are determined using Pole-placement method. FreeMASTER MCAT control and tuning is described in *FreeMASTER and MCAT user interface*.

The MCAT tool generates a set of constants to the dedicated header file (for example “{Project Location}\Sources\Config\PMSM_appconfig.h”). The names of the constants can be redefined within the MCAT configuration file “Header_file_constant_list.xml” (“{Project Location}\FreeMASTER_control\ MCAT\src\xml_files\”). The PMSM_appconfig.h contains application scales, fault triggers, control loops parameters, speed sensor and/or observer settings and FreeMASTER scales. The PMSM_appconfig.h should be linked to the project and the constants should be used for the variables initialization.

The FreeMASTER enables an online tuning of the control variables using MCAT control and tuning view. However, the FreeMASTER must be aware of the used control-loop variables. A set of the names is stored in “FM_params_list.xml” (“{Project Location}\FreeMASTER_control\MCAT\src\xml_files\”).

5. FreeMASTER and MCAT user interface

The FreeMASTER debugging tool is used to control the application and monitor variables during run time. Communication with the host PC passes via USB. However, because FreeMASTER supports RS232 communication, there must be a driver for the physical USB interface, OpenSDA, installed on the host PC that creates a virtual COM port from the USB. The driver shall be installed automatically plugging S32K144EVB to USB port. Alternatively, it can be downloaded from www.pemicro.com/opensda/. The application configures the LPUART module of the S32K144 for a communication speed of 115200bps. Therefore, the FreeMASTER user interface also needs to be configured respectively.



Motor Control Application Tuning tool

The MCAT is intended to be used as a tool for real-time tuning and debugging of PMSM industrial applications. Parameters of the Field Oriented Control structure are estimated by MCAT based on the dynamic requirements and system parameters. Right PI controller parameters lead to desirable behavior of the motor quantities. The static configuration of the tuned system can be stored in an external header file.
Connecting and tuning a new electric drive setup becomes easier with a Control Structure tab which offers the possibility to split the control structure and allows controlling the motor at different levels of cascade control structure.

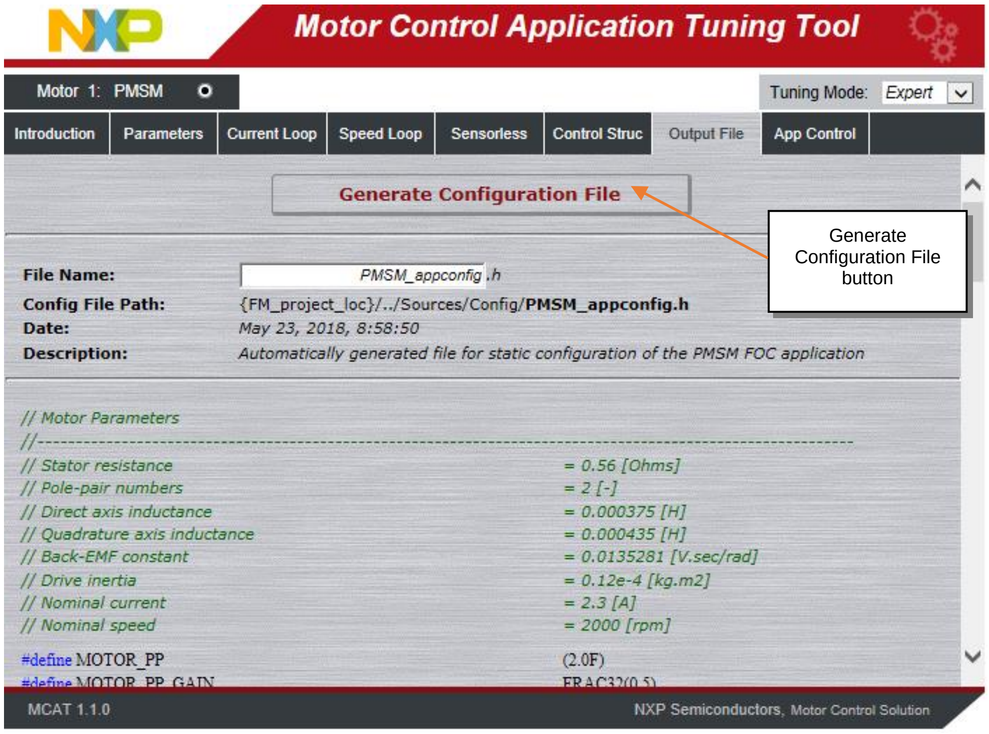
Name	Value	Unit	Period
On/Off	Stop	ENUM	1000
Speed Required	1900	[Rpm]	1000
Mode	automatic	ENUM	1000
Position Mode	force	ENUM	1000
Clear Faults	--	ENUM	0
State	Ready	ENUM	0
Event	e_ready	ENUM	1000

Figure 42. **FreeMASTER and Motor Control Application Tunning Tool**

5.1. MCAT Settings and Tuning

5.1.1. Application configuration and tuning

FreeMASTER and MCAT interface (*Figure 42*) enables online application tuning and control. The MCAT tuning shall be used before the very first run of the drive to generate the configuration header file (PMSM_appconfig.h). Most of the variables are accessible via MCAT online tuning (thus can be updated anytime), but some of them (especially the fault limit thresholds) must be set using the configuration header file generation, which can be done on the "Output File" panel by clicking the "Generate Configuration File" (see *Figure 43*).


Motor Control Application Tuning Tool
Motor 1: PMSM
Tuning Mode: Expert
Introduction
Parameters
Current Loop
Speed Loop
Sensorless
Control Struc
Output File
App Control
Generate Configuration File
Generate Configuration File button
MCAT 1.1.0
NXP Semiconductors, Motor Control Solution

File Name: PMSM_appconfig .h
Config File Path: {FM_project_loc}/../Sources/Config/**PMSM_appconfig.h**
Date: May 23, 2018, 8:58:50
Description: Automatically generated file for static configuration of the PMSM FOC application

```
// Motor Parameters
//----------------------------------------------------------------------------------------------------
// Stator resistance                              = 0.56 [Ohms]
// Pole-pair numbers                              = 2 [-]
// Direct axis inductance                         = 0.000375 [H]
// Quadrature axis inductance                     = 0.000435 [H]
// Back-EMF constant                              = 0.0135281 [V.sec/rad]
// Drive inertia                                  = 0.12e-4 [kg.m2]
// Nominal current                                = 2.3 [A]
// Nominal speed                                  = 2000 [rpm]

#define MOTOR_PP                                  (2.0F)
#define MOTOR_PP_GAIN                             FRAC32(0.5)
```

Figure 43. **Output File panel and "Generate Configuration File" button**

Parameters runtime update is done using the "Update Target" button (see *Figure 44*). Changes can be also saved using "Store Data" button, or reloaded to previously saved configuration using "Reload Data" button.

Any change of parameters highlights the cells that have not been saved using "Store data". Changes can be reverted using "Reload Data" to previously saved configuration. This button is disabled if no change has been made.

NOTE

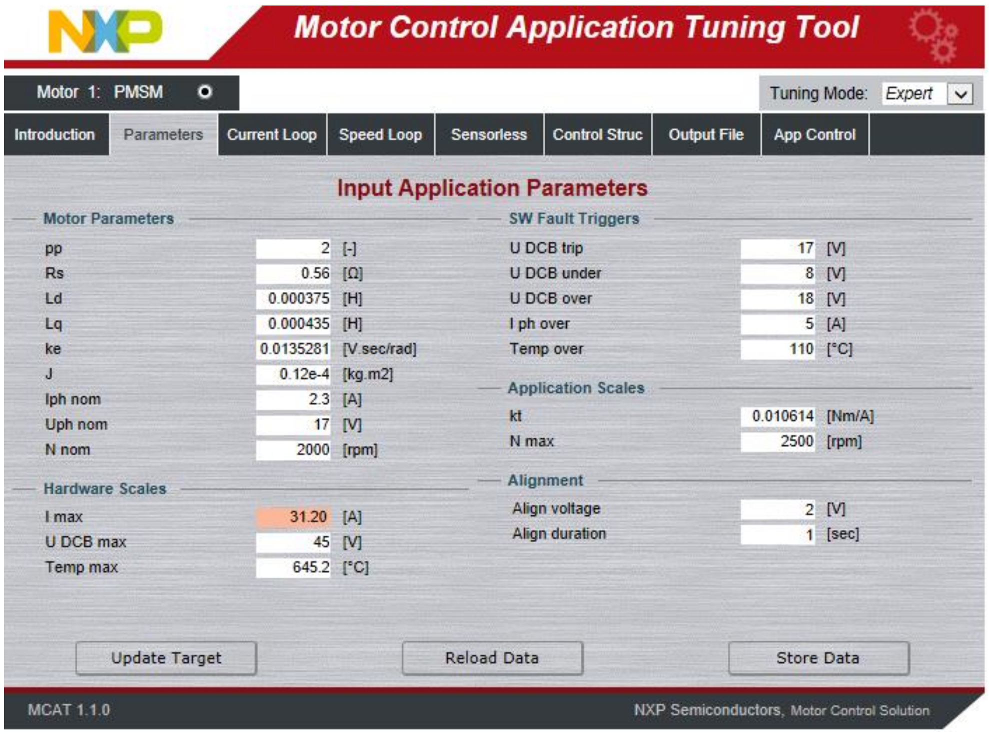
MCAT tool can be configured using hidden mouse-over "Settings" button (see *Figure 42*), where a set of advanced settings, for example PI controller types, speed sensors and other blocks of the control structure can be changed. However, it is not recommended to change these settings since it will force the MCAT to look for a different variables names and to generate different set of constants than the application is designed for. See MCAT tool documentation available at nxp.com.

The application tuning is provided by a set of MCAT pages dedicated to every part of the control structure. An example of the Application Parameters Tuning page is in *Figure 44*. Following list of settings pages is based on the PMSM sensorless application.

- Parameters
  - Motor Parameters
  - Hardware Scales
  - SW Fault Triggers
  - Application Scales
  - Alignment
- Current Loop
  - Loop Parameters
  - D axis PI Controller
  - Q axis PI Controller
  - Current PI Controller Limits
  - DC-bus voltage IIR filter settings
- Speed Loop
  - Loop Parameters
  - Speed PI Controller Constants
  - Speed Ramp
  - Speed Ramp Constants
  - Actual Speed Filter
  - Speed PI Controller Limits
- Sensorless
  - BEMF Observer Parameters
  - BEMF DQ Observer Coefficients
  - Tracking Observer PI Constants
  - Tracking Observer Integrator
  - Open Loop Start-up Parameters
  - BEMF DQ Observer PI Controller Constants

Changes can be tested using MCAT “Control Struc” page (*Figure 45*), where the following control structures can be enabled:

- Scalar Control
- Voltage FOC (Position & Speed Feedback is enabled automatically)
- Current FOC (Position & Speed Feedback is enabled automatically)
- Speed FOC (Position & Speed Feedback is enabled automatically)



Tuning Mode: Expert

Input Application Parameters

Motor Parameters

pp	2	[-]
Rs	0.56	[Ω]
Ld	0.000375	[H]
Lq	0.000435	[H]
ke	0.0135281	[V.sec/rad]
J	0.12e-4	[kg.m2]
Iph nom	2.3	[A]
Uph nom	17	[V]
N nom	2000	[rpm]

Hardware Scales

I max	31.20	[A]
U DCB max	45	[V]
Temp max	645.2	[°C]

SW Fault Triggers

U DCB trip	17	[V]
U DCB under	8	[V]
U DCB over	18	[V]
I ph over	5	[A]
Temp over	110	[°C]

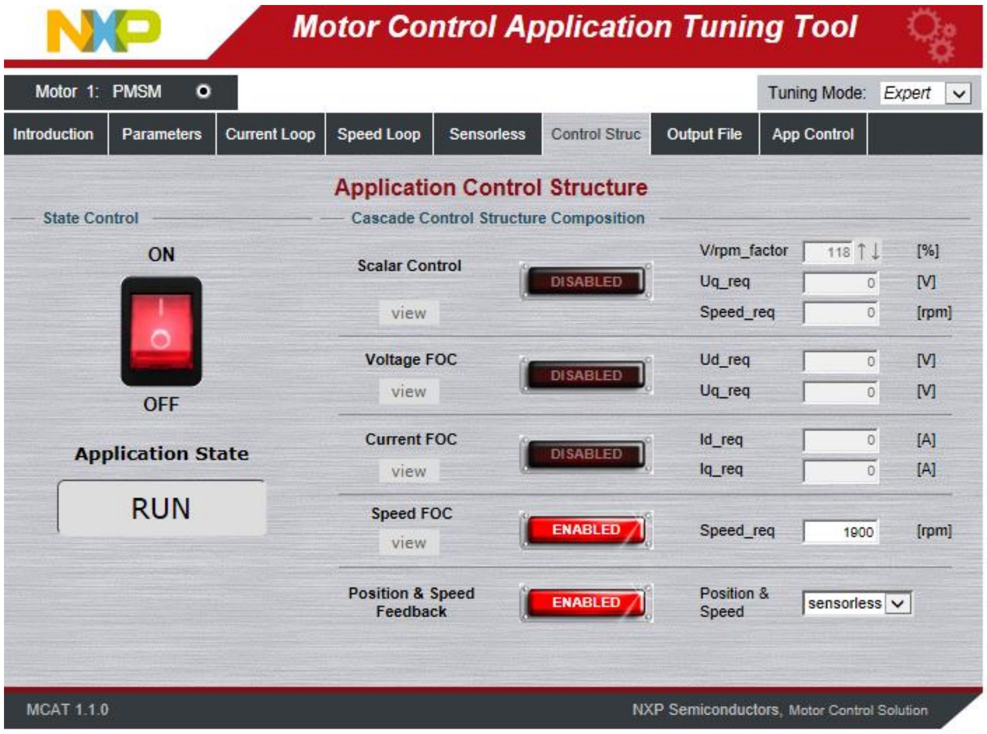
Application Scales

kt	0.010614	[Nm/A]
N max	2500	[rpm]

Alignment

Align voltage	2	[V]
Align duration	1	[sec]

Figure 44. **MCAT input application parameters page**



Tuning Mode: Expert

Application Control Structure

State Control

Application State: RUN

Cascade Control Structure Composition

Scalar Control	DISABLED	V/rpm_factor	118	[%]
		Uq_req	0	[V]
		Speed_req	0	[rpm]
Voltage FOC	DISABLED	Ud_req	0	[V]
		Uq_req	0	[V]
Current FOC	DISABLED	Id_req	0	[A]
		Iq_req	0	[A]
Speed FOC	ENABLED	Speed_req	1900	[rpm]
Position & Speed Feedback	ENABLED	Position & Speed	sensorless	

Figure 45. **MCAT application control structure page**

5.2. MCAT application Control

All application state machine variables can be seen on the FreeMASTER MCAT App control page as shown in *Figure 46*. Warnings and faults are signaled by a highlighted red color bar with name of the fault source. The warnings are signaled by a round LED-like indicator, which is placed next to the bar with the name of the fault source. The status of any fault is signaled by highlighting respective indicators. In *Figure 46*, for example, there is pending fault flag and one warning indicated ("Udcb LO" - DC bus voltage is close to its under voltage conditions). That means that the measured voltage on the DC bus exceeds the limit set in the MCAT_Init function. The warning indicator is still on if the voltage is higher than the warning limit set in INIT state. In this case, the application state FAULT is selected, which is shown by a frame indicator hovering above FAULT state. After all actual fault sources have been removed, no warning indicators are highlighted, but the fault indicators will remain highlighted. The pending faults can now be cleared by pressing the "FAULT" button. This will clear all pending faults and will enable transition of the state machine into INIT and then READY state. After the application faults have been cleared and the application is in READY state, all variables should be set to their default values. The application can be started by selecting APP_ON on application On/Off switch. Successful selection is indicated by highlighting the On/Off button in green.

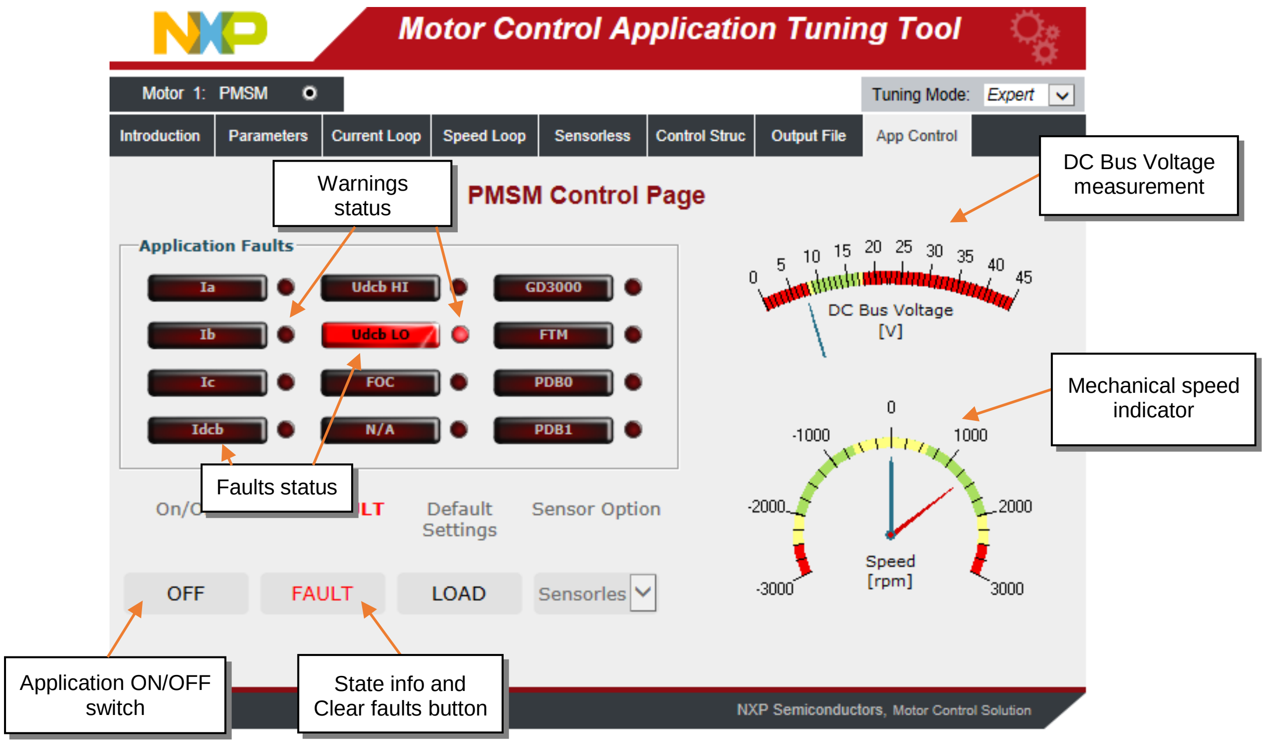


Figure 46. **FreeMASTER MCAT Control Page for controlling the application**

6. Conclusion

Design, described in this application note shows the simplicity and efficiency in using the S32K144 microcontroller for Sensorless PMSM motor control and introduces it as an appropriate candidate for various low-cost applications in the automotive area. MCAT tool provides interactive online tool which makes the PMSM drive application tuning friendly and intuitive.

7. References

- MCSPTE1AK144: S32K144 Development Kit for BLDC and PMSM motor control
- S32 Design Studio IDE for ARM® based MCUs
- FreeMASTER Run-Time Debugging Tool
- S32K14XMCLUG , Automotive Math and Motor Control Library Set for S32K14x User Manual
- S32K1XXRM, S32K1xx Series Reference Manual
- S32K144EVB: S32K144 Evaluation Board
- DEVKIT-MOTORGD: Low-Cost Motor Control Solution for DEVKIT Platform
- GD3000: 3-Phase Brushless Motor Pre-Driver
- Rashid, M. H. Power Electronics Handbook, 2nd Edition. Academic Press
- Motor Control Application Tuning (MCAT) Tool

8. Revision history

Revision Number	Date	Substantive changes
0	08/2018	Initial release
1	05/2020	• Updated Figure 12, Figure 13, Figure 15, Figure 17, Figure 20, Figure 21 and Figure 22. • Updated Example 3, Example 4, Example 6, Example 7 and Example 11. • In Module involvement in digital PMSM Sensorless control loop updated the text "DC bus voltage measurement….. queue in memory". • In Center-aligned PWM mode updated the text "FTM_DRV_MaskOutputChannels function disables PWM output masking all FTM channels". • In Quadrature decoder mode updated the text "While more general structure flexTimer_qd2_InitConfig takes effect calling

		McuFtmConfig() function at very beginning in the initialization phase”. • Updated Table2 and Table3. • Updated the first bullet in References. • Updated Figure38.

Document Number:AN12235
Rev. 1
06/2020