NXP Semiconductors
Application Notes

Document Number:AN13000
Rev. 0, 09/2020

3-phase Sensorless BLDC Motor Control Kit with S32K116

Featuring Motor Control Application Tuning (MCAT) Tool

Contents

1. Introduction

This application note describes the design of a 3-phase Brushless DC (BLDC) motor control drive using a sensorless algorithm and 3-phase low-voltage power stage DEVKIT-MOTORGD based on SMARTMOS® MC34GD3000 pre-driver. DEVKIT-MOTORGD is designed to supply low power 3-phase PM motors and measure analog and digital quantities required by this application.

This design serves as an example of motor control design using NXP family of automotive motor control MCUs based on a 32-bit ARM® Cortex-M0®+ optimized for a full range of automotive applications.

Following are the supported features:

- 3-phase BLDC speed control based on Six-step commutation control
- Shaft position obtained by Hall sensor or by BEMF (Back Electromotive Force) voltage zero-crossing detection technique
- DC-bus current, DC-bus voltage and BEMF voltage sensing
- Motor speed determined by Hall sensor period or BEMF zero-crossing period
- 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 BLDC motor. The application meets the following performance specifications:

- Targeted at the S32K116EVB Evaluation Board (refer to dedicated user manual for S32K116EVB available at www.nxp.com) See section *References* for more information.
- Control technique incorporating:
  - Six-step commutation control of 3-phase brushless DC motor with and without position sensor
  - Rotor position is obtained by Hall sensor or by BEMF (Back Electromotive Force) voltage zero-crossing detection technique
  - Closed-loop speed control with action period 1ms
  - Bi-directional rotation
  - Motor current limitation
  - Alignment and start-up
  - 50µs sampling period
- Automotive Math and Motor Control Library (AMMCLIB) – Speed control loop 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, speed loop, sensorless parameters)
- FreeMASTER software MCAT graphical control page (required speed, actual motor speed, start/stop status, DC-Bus voltage level, DC-Bus current, system status)
- FreeMASTER software speed scope (observes actual and desired speeds, DC-Bus voltage and DC-Bus current)
- FreeMASTER software high-speed recorder (six-step commutation control quantities)
- DC-Bus over-voltage and under-voltage, over-current, overload and start-up fail protection.

3. Sensorless BLDC control

3.1. Overview of the brushless DC motor

The BLDC motor (*Figure 1*) is a rotating electric machine with a classic slotted stator filled by 3-phase winding similar to an induction motor. The phases mounted on the stator are connected to form a star or delta connection. The rotor has surface-mounted permanent magnets. The motor can have more than one pole pair per phase. The number of pole pairs per phase defines the ratio between the electrical revolution and the mechanical revolution.

The BLDC motor is equivalent to an inverted DC brushed motor, where the magnet rotates while the conductors remain stationary. In the DC brushed motor, the commutator and brushes reverse the current polarity in such a way that stator and rotor magnetic fields are perpendicular. However, in the brushless DC motor, a power transistor (which must be switched in synchronization with the rotor position) performs the polarity reversal. This process is also known as electronic commutation.

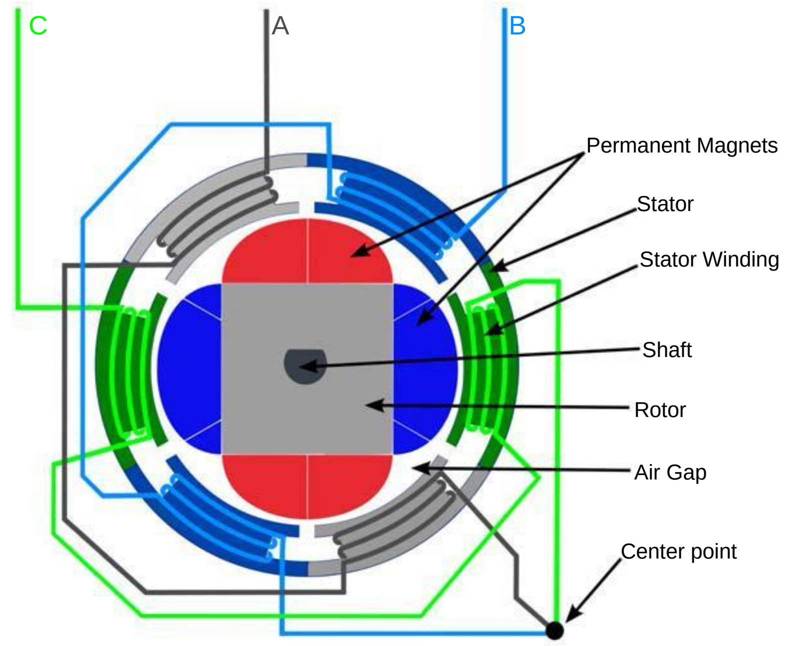


Figure 1. **BLDC motor – cross-section**

The arrangement of the magnets on the rotor creates a trapezoidal back electromotive force (BEMF) shape when the rotor is spinning. Neglecting the higher-order harmonic terms, the BEMF in the motor phase (e_a,e_b,e_c) is as indicated in *Figure 2*. Each BEMF has a constant amplitude for 120 electrical degrees, followed by a 60 electrical degree transition in each half-cycle. The ideal current waveforms in each phase (i_a,i_b,i_c) need to be quasi-square waveforms of 120 electrical degrees of conduction angle in each half-cycle. The conduction of current in each phase must coincide with the flat part of the BEMF waveforms; this guarantees that the developed torque is constant or ripple-free at all times. In order to align current conduction in each phase with the flat part of the BEMF, the rotor position must be known.

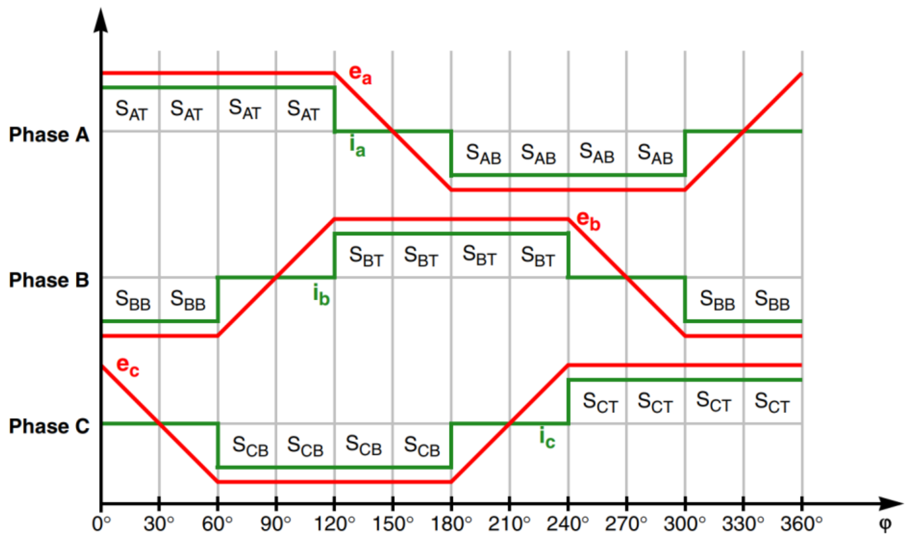


Figure 2. **3-phase BEMF voltages and phase currents of a BLDC motor**

The position of the rotor can be obtained by a position sensor or a sensorless algorithm. Various kinds of position sensors are used. However, since the rotor is a permanent magnet, it is a very simple matter to determine where the physical pole edges are using a simple, reliable, and inexpensive Hall effect sensor.

The following techniques are commonly used to estimate rotor position in applications that rely on sensorless control of a BLDC motor:

- BEMF zero-crossing detection method
- Flux level detection method
- Various kinds of system state observers
- Signal injection methods

From a control perspective, two logical mechanisms must be employed:

- *Commutation control*, where the phases are energized according to rotor position with the quasi-square current waveforms.
- *Speed/torque control*, where the amplitude of the quasi-square current waveform applied to the phases is controlled to achieve the desired speed/torque performance.

The following sections discuss the concept of the BEMF zero-crossing detection method, as well as the methods and conditions for its correct evaluation.

3.1.1. Electronic commutation control

The commutation process provides a mechanism to energize phases according to the rotor position with the quasi-square current waveforms. Since only six discreet outputs per electrical cycle are required (as shown in *Figure 2*), six semiconductor power switches are sufficient to create quasi-square current waveforms for the phases. Six semiconductor power switches form a 3-phase power inverter, designed using IGBT or MOSFET switches. The power for the system is provided by the DC bus voltage U_{DCB}. The semiconductor switches and diodes are modeled as ideal devices in *Figure 3*.

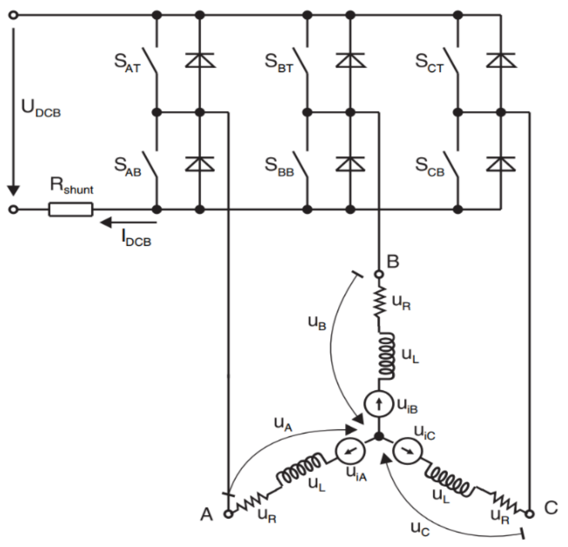


Figure 3. **Power stage and motor topology**

Six-step commutation is a very common method for driving a 3-phase star-connected BLDC motor. In six-step commutation control, the BLDC motor is operated in a two-phase model. Two phases are energized while the third phase is disconnected as the space between the magnet poles passes over it and produces a zero BEMF voltage. Selection of the two energized phases is carried out by a position sensor or a position observer. *Table 1* shows the output current waveforms for a 3-phase inverter and the switching devices that conduct during the six switching intervals per cycle.

Table 1 **Six-step switching sequence**

Rotor position	Sector number	Switch closed		Phase current		
				Phase A	Phase B	Phase C
0°-60°	0	S_{AT}	S_{BB}	+	–	Off
60°-120°	1	S_{AT}	S_{CB}	+	Off	–
120°-180°	2	S_{BT}	S_{CB}	Off	+	–
180°-240°	3	S_{BT}	S_{AB}	–	+	Off
240°-300°	4	S_{CT}	S_{AB}	–	Off	+
300°-360°	5	S_{CT}	S_{BB}	Off	–	+

3.1.2. Speed/torque control

Commutation ensures the proper direction of the phase current according to the rotor position of the BLDC motor, while the motor torque/speed only depends on the amplitude of the quasi-square current waveform. Continued control of the amplitude of the quasi-square current waveform for each phase of the motor is ensured by hysteresis or PWM control.

PWM control is commonly used in applications where microcontrollers are employed. The duty cycle for the PWM modulator is obtained by the speed PI controller. The speed PI controller amplifies the error between the required and actual speeds, and its output, appropriately scaled, is assigned to the PWM modulator.

The actual mechanical speed can be calculated as a time derivative of the shaft position φ_{mech}.

$$\omega_{mech} = \frac{d\varphi_{mech}}{dt} = \frac{1}{p}\frac{d\varphi_{el}}{dt} \approx \frac{1}{p}\frac{\Delta\varphi_{el}}{\Delta T}$$

Equation 1

Since the shaft travels exactly 1/6 of one electrical revolution (2π in radians) between two commutations, the above equation can be rewritten to the following form:

$$\omega_{mech} = \frac{1}{p}\frac{d\varphi_{el}}{dt} = \frac{1}{p}\frac{\frac{360°}{6}}{T_{CM}} = \frac{1}{p}\frac{360°}{T_{(0°\to60°)} + T_{(60°\to120°)} + T_{(120°\to180°)} + T_{(180°\to240°)} + T_{(240°\to300°)} + T_{(300°\to360°)}} = \frac{360°}{p\sum_{n=0}^{5} T_{CM}{}^{n}}$$

Equation 2

Where:

- p is the number of pole pairs
- T_{CM} is the time between two consecutive commutations
- $T_{CM}{}^{n}$ is the time between commutations in sector n = 0, 1, 2, 3, 4, 5
- φ_{el} is the electrical position

3.2. Output voltage actuation and complementary unipolar PWM modulation technique

The 3-phase voltage source inverter is shown in *Figure 4.* Voltage dividers connected to motor phases serve on BEMF voltage measurement. Shunt resistor R60 is used for DC Bus current measurement.

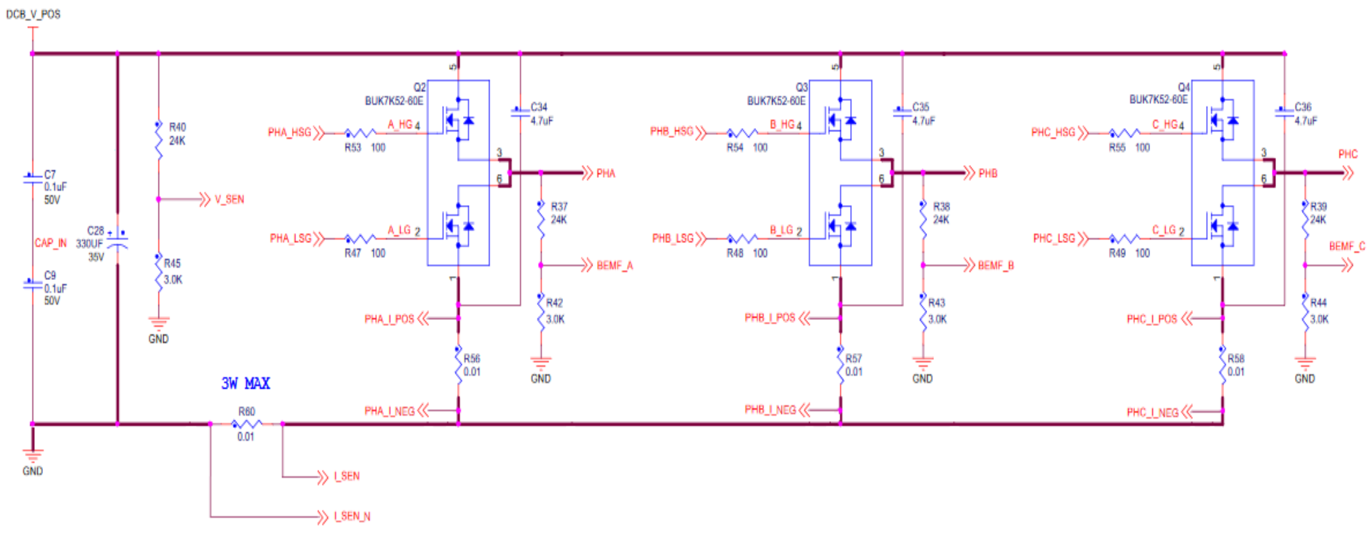


Figure 4. **3-phase DC/AC inverter with shunt resistors for current measurement**

There are different methodologies for powering and switching the phases. The unipolar PWM control technique combines commutation control and torque control. While the state of the switches is determined by commutation control, the torque is controlled by the applied duty cycle. An application with BLDC control where the unipolar PWM control technique is employed, benefits from a reduction in the MOSFET switching losses and an improvement in the system’s EMC robustness.

The unipolar PWM control means that the motor phase sees only the positive polarity of the voltage. To achieve the unipolar PWM pattern, one phase is in complementary PWM mode while the second phase is grounded and the third phase stays unpowered, as shown in *Figure 5*. This PWM pattern can be seen every 60 electrical degrees, and they differ only in phase order. The phase order is determined according to the shaft position by commutation control.

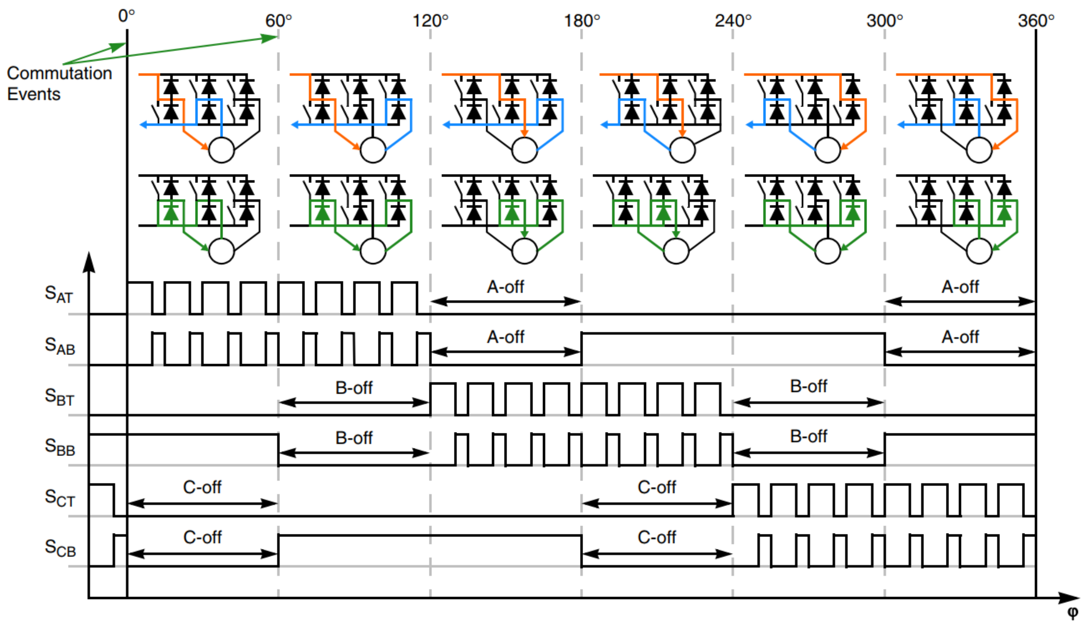


Figure 5. **Complementary unipolar PWM switching**

For example, in the first cycle, Phase A is powered by the complementary PWM signal while the bottom transistor of Phase B is grounded and Phase C is unpowered. After the commutation event at 60° electrical degrees, Phase A is still powered by the complementary PWM signal, Phase B is unpowered, and Phase C becomes grounded instead.

The control described in this document is based on the complementary/independent unipolar PWM modulation technique.

The following section explains sensorless position estimation by means of BEMF zero-crossing detection for commutation control purposes.

3.3. Position estimation based on BEMF zero-crossing detection

Figure 2 shows ideal BEMF waveforms (e_a, e_b, e_c) and depicts a commutation event occurring at a position of 30 electrical degrees after the point where a BEMF zero-crossing arises. The BEMF zero-crossing happens at a position of 30 electrical degrees after the point of the last commutation event. Let us assume that the motor is spinning at a constant velocity; in this case, the motor needs the same amount of time to travel from the position of the last commutation event to a BEMF zero-crossing and from the BEMF zero-crossing to the following commutation event. In the time domain, a BEMF zero-crossing is right in the middle of two commutation events. Therefore, the BEMF zero-crossing event, with help of a timer, can simply be used to estimate the right commutation point as well as the velocity of the rotor.

3.3.1. BEMF zero-crossing principle

To explain and simulate the idea of BEMF sensing techniques, this document provides a simplified mathematical model based on the basic circuit topology (see *Figure 6*). The goal of the mathematical model is to identify dependencies between the measurable motor waveforms and a BEMF zero-crossing. The BEMF zero-crossing, in turn, helps to identify the commutation event.

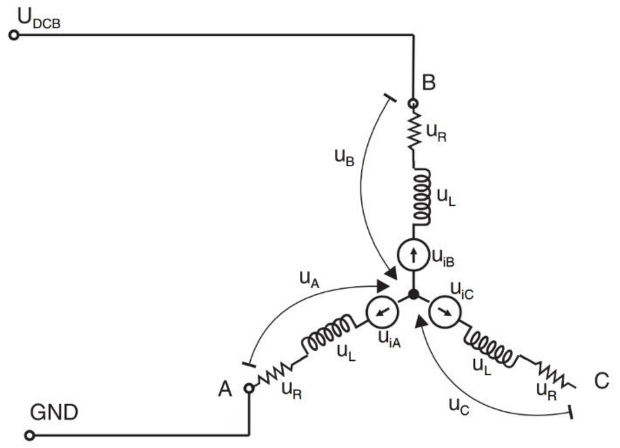


Figure 6. **Basic BLDC motor circuit topology**

The mathematical model is based on the fact that only two phases of a motor are energized and the third is disconnected. The natural voltage level of the whole model is referenced to half of the DC bus voltage, which simplifies the mathematical expressions. The mathematical model assumes that the motor phases are symmetrical (see *Figure 6*).

$$\left.\begin{aligned} u_N &= U_{DCB} - Ri_b - L\frac{di_b}{dt} - e_b \\ u_N &= Ri_a + L\frac{di_a}{dt} - e_a \end{aligned}\right\} \underset{i_a=i_b}{\Longrightarrow} u_N = \frac{U_{DCB}}{2} - \frac{e_b+e_a}{2}$$

Equation 3

For a symmetrical 3-phase motor, the sum of all BEMF voltages is zero, therefore:

$$e_c + e_b + e_a = 0 \rightarrow e_c = -(e_b - e_a)$$

Equation 4

The unpowered phase has the following voltage equation, since there is no current flowing:

$$u_N = u_C - e_C$$

Equation 5

By substituting *Equation 3* with *Equation 4* and *Equation 5*, the phase voltage on the unpowered phase can be derived as:

$$u_c = \frac{U_{DCB}}{2} + \frac{3}{2}e_c$$

Equation 6

At the time of the BEMF zero-crossing, the BEMF voltage (e_c in this case) is zero as the name implies. Therefore, by measuring voltage at the unpowered phase (e_c) and comparing it to half of the DC bus voltage, the BEMF zero-crossing can be accurately identified.

3.3.2. BEMF zero-crossing event detection and phase current measurement

The exact position of the rotor can be sensed by measuring the BEMF voltage induced by the rotating permanent magnet in the unpowered phase, *Figure 7*.

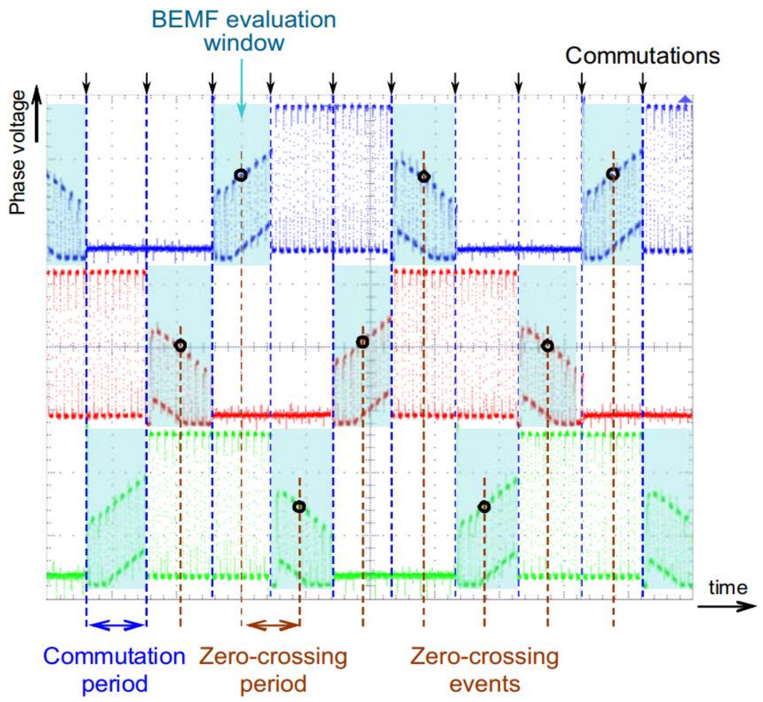


Figure 7. **BEMF zero-crossing and commutation events, and their relationship to complementary unipolar PWM switching**

In *Figure 7*, the blue windows mark the time periods in which the respective phase is unpowered. The voltage measured in this time window is the BEMF voltage. At the BEMF zero-crossing event, the permanent magnet is right in front of a coil and the rotor field is positioned 90° versus the stator field. This event happens in the middle of a commutation period and is marked as the black circles in the blue BEMF window. At this time, the phase voltage is equal to half of the DC bus voltage, as described in *3.3.1*. In the case of a constant shaft velocity, the period between two following zero-crossing events is equal to the commutation period.

Figure 8 zooms in closer to one of the PWM cycles. At the top of the figure is the PWM pattern, where Phase A is controlled by PWM and Phase C is grounded for the entire PWM period. During the PWM On cycle, the top switch of Phase A is turned on and the bottom switch of Phase C is grounded. Current flows from the DC bus into Phase A, and back through Phase C and the DC bus shunt resistor. In this cycle, the center point of the motor shows a voltage level of $U_{DCB}/2$. The BEMF voltage in the unpowered phase changes relatively to $U_{DCB}/2$ in the positive and negative directions, which means that the zero-crossing is detectable when the phase voltage on the unpowered phase is equal to $U_{DCB}/2$. Also, the phase current is measurable on the DC bus shunt.

During the Off cycle of the PWM period, both the Phase A and Phase C bottom switches are on. Therefore, phase current circulates through Phase A, Phase C, and the two bottom switches back. During this cycle, the phase current is unable to reach the DC bus shunt resistor and the phase current cannot be measured. The center point of the motor as well is connected to ground, and the zero-crossing cannot precisely be measured in that cycle.

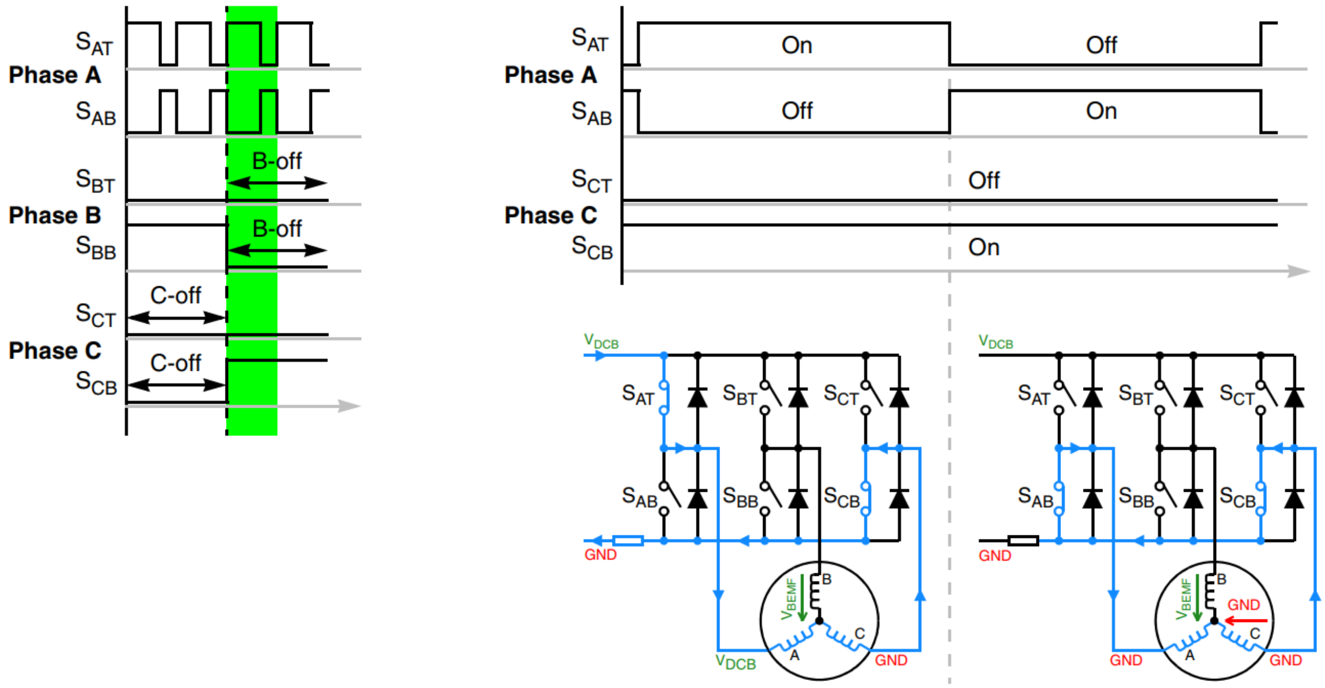


Figure 8. **BEMF zero-crossing detection with complementary unipolar PWM switching**

Following on from the discussion above, phase current and BEMF voltage measurements must be performed in the active phase of the PWM cycle.

3.3.3. BEMF voltage measurement

As we learned earlier, the BEMF voltage can only be measured during the active phase of the PWM. Importantly, this is measured towards the end of the active cycle due to switching noises. In *Figure 9*, the green marked area shows the window in which the BEMF should be measured.

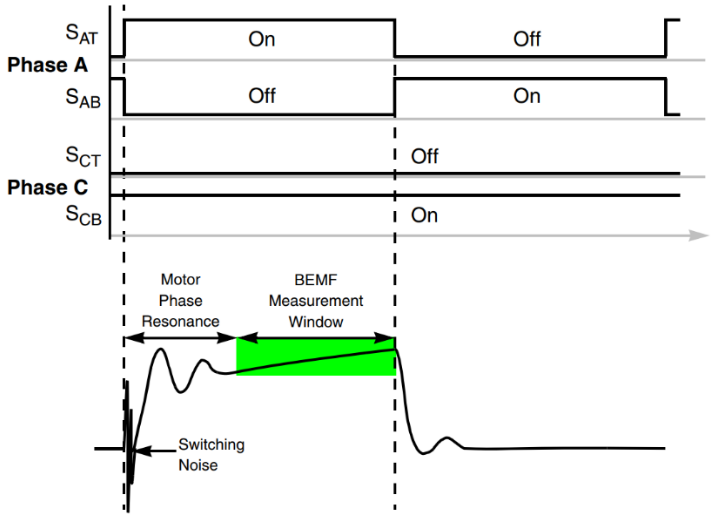


Figure 9. **BEMF voltage measurement**

It should be noted that, depending on the motor and power stage parameters, the amplitude, period, and damping of the voltage ringing vary. As a result, it is recommended that the BEMF voltage is measured close to the end of the window. The time of this sample point also needs to be stored, as it is used to enhance zero-crossing detection.

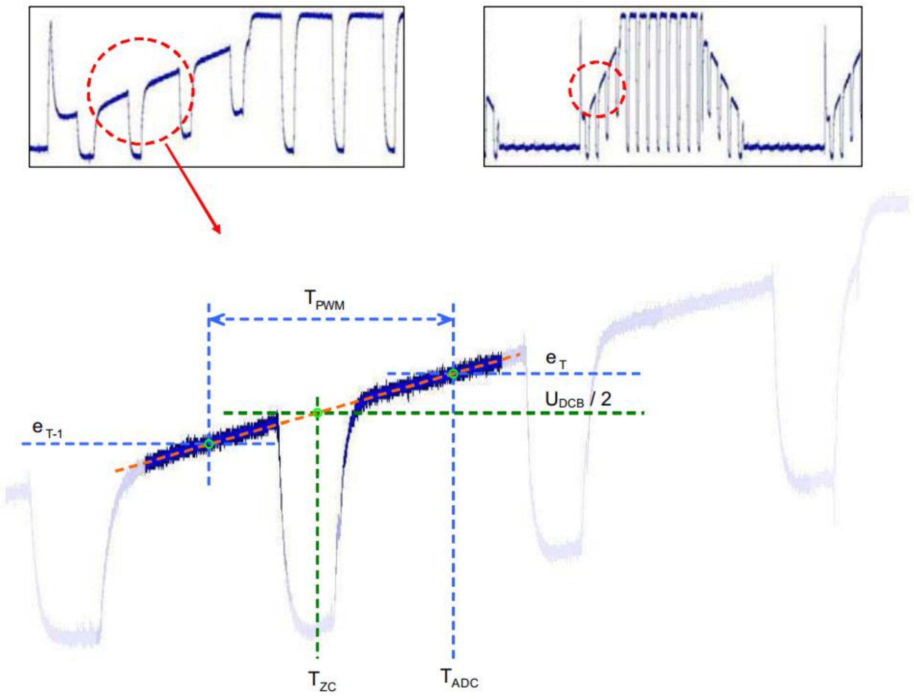


Figure 10. **Precise BEMF zero-crossing identification**

If we zoom in again and look at the BEMF voltage cycles (see *Figure 10*), it can be seen that the crossing of the BEMF voltage and level can take place wherever between two following BEMF voltage measurements. For accurate position estimation, an exact zero-crossing point has to be identified. This exact zero-crossing point identification is done by an approximation based on the interpolation of two following BEMF measurements.
Assuming that the shaft is not accelerating, actual BEMF voltage was measured at time T_{ADC} with the voltage level of e_T, and the previous measurement was taken at the time of $T_{ADC} - T_{PWM}$ with the voltage level of e_{T-1}, then the equation to calculate the exact time of the zero-crossing event could be derived as follows:

$$\frac{e_T - e_{T-1}}{T_{PWM}} = \frac{e_T - \frac{U_{DCB}}{2}}{T_{ADC} - T_{ZC}} \Rightarrow T_{ZC} = T_{ADC} - \frac{e_T - \frac{U_{DCB}}{2}}{e_T - e_{T-1}} T_{PWM}$$

Equation 7

This formula is calculated in the commutation period when two following comparisons of the BEMF voltage to half of the DC bus have the opposite signs.

In order to enhance the accuracy of the zero-crossing event even further, the DC bus voltage and BEMF voltage need to be measured simultaneously. DC bus voltage and phase BEMF voltage are scaled by voltage dividers to respect 5V ADC input voltage range (*Figure 11*).

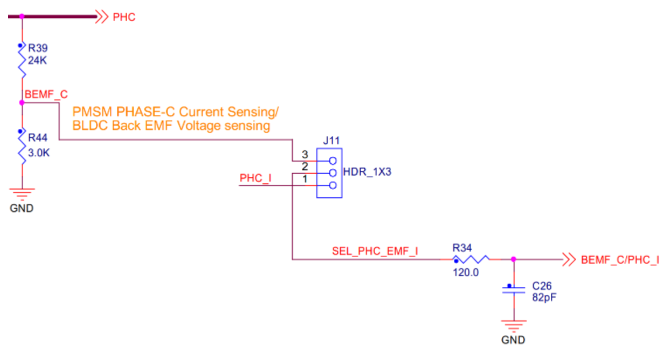


Figure 11. **Phase C Back EMF voltage sensing circuit**

3.3.3.1. BEMF voltage measurement limitations

The accuracy of the sensorless BLDC motor control algorithm based on the BEMF voltage measurement is mostly limited by the precision of the BEMF voltage measured on a non-fed motor's phase. For example, the ADC accuracy, precision of the phase voltage sensing circuitry, signal noise, and distortion caused by the power switching modules, all these factors need to be taken into account. Noise generated by power switching modules can be eliminated by correctly setting the measurement event to be far away from the switching edges (PWM to ADC synchronization). There still exists some limitation that cannot be eliminated, namely the decay or freewheeling period. As soon as the phase is disconnected from the power by the commutation event, there is still a current flowing through the freewheeling diode. The conducting freewheeling diode connects the released phase to either a positive or a negative DC bus voltage. The conduction time depends on the momentary load of the motor. In some circumstances, the conduction time is so long that it doesn't allow the detection of BEMF voltage, as represented in *Figure 12*.

It is important to differentiate between the BEMF voltage generated by the motor and the phase voltage tied to a positive or negative DC bus voltage during the decay period. For this purpose, a blanking time period after the commutation event has to be employed. During this period, the BEMF voltage is not sensed or used for sensorless control. The blanking period duration should reflect the motor, load, and dynamic application parameters.

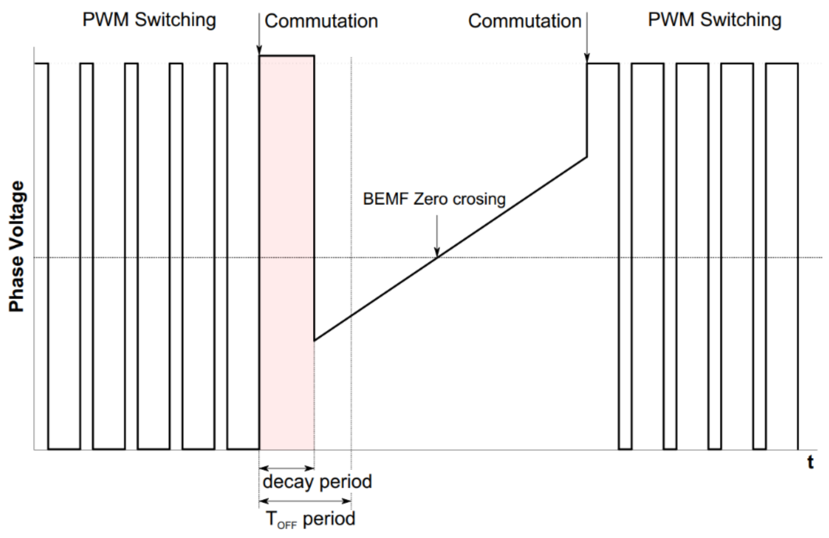


Figure 12. **BEMF decay period**

3.3.4. DC bus current measurement

DC bus current flows through R60 shunt resistor and produces voltage drop that is amplified by internal MC34GD3000 OAMP to fit ADC input voltage range (see section *References* for more details).

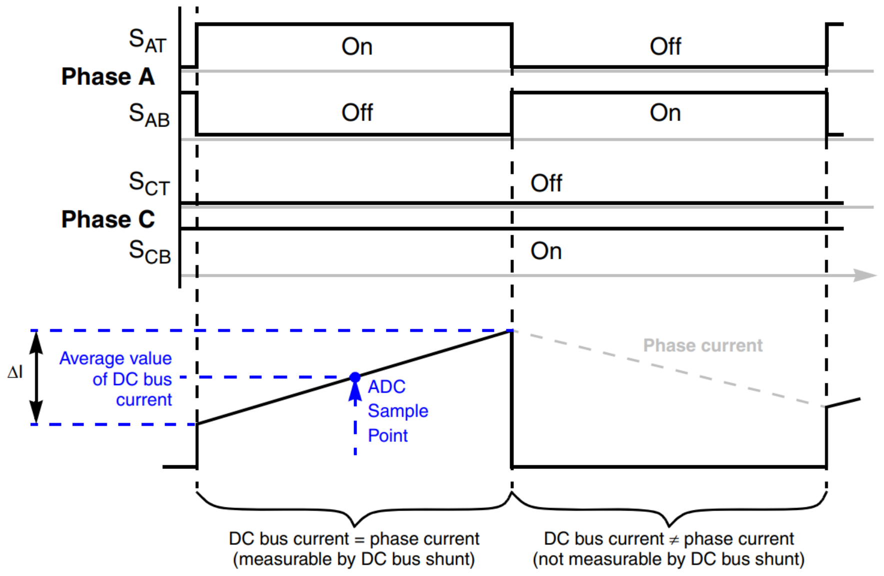

S_{AT}
Phase A
S_{AB}
On
Off
Off
On
S_{CT}
Phase C
S_{CB}
Off
On
ΔI
Average value of DC bus current
ADC Sample Point
Phase current
DC bus current = phase current (measurable by DC bus shunt)
DC bus current ≠ phase current (not measurable by DC bus shunt)

Figure 13. **DC bus current measurement**

As mentioned in *3.3.2*, the DC bus current has to be measured in the active cycle of the PWM period due to the fact, that the DC bus current equals the phase current only in the active cycle, as illustrated in *Figure 13*.

During the active cycle of the PWM period, the phase current is rising. The slope of the rising current is defined by the motor phase coil inductance; the lower the phase inductance, the steeper the slope of the rising current.

To obtain the average value of the DC bus current directly, the voltage on the DC bus shunt resistor has to be measured in the middle of the active PWM cycle *Figure 13*.

3.4. States of the sensorless BLDC control based on BEMF zero-crossing detection

In order to start and run the BLDC motor, the control algorithm has to go through the following states:

- Alignment (initial position setting)
- Start-up (forced commutation or open-loop mode)
- Run (sensorless running with BEMF acquisition and zero-crossing detection)

3.4.1. Alignment

As mentioned previously, the main task for sensorless control of a BLDC motor is position estimation. Before starting the motor, however, the rotor position is not known. The aim of the alignment state is to align the rotor to a known position. This known position enables starting the rotation of the shaft in the desired direction and generating the maximal torque during start-up. During the alignment state, all three phases are powered in order to get the best performance behavior in either direction of shaft rotation. Phase C is connected to the positive DC bus voltage and phases A and B are grounded. The alignment time depends on the mechanical constant of the motor, including load, and also on the applied motor current.

3.4.2. Start-up

In the start-up state, motor commutation is controlled in an open-loop mode without any rotor position feedback. The commutation period is controlled by an open-loop starting curve. The open-loop start is required only until the shaft speed is high enough (approximately 5% of nominal motor speed) to produce an identifiable BEMF voltage.

3.4.3. Run

The block diagram of the run state is represented by *Figure 14* and includes the BEMF acquisition with zero-crossing detection in order to control the commutations. The motor speed is estimated based on zero-crossing time periods. The difference between the demanded and estimated speeds is fed into the speed PI controller. The output of the speed PI controller is proportional to the voltage to be applied to the BLDC motor. The motor current is measured and filtered during the BEMF zero-crossing event and used as feedback into the current controller. The output of the current PI controller limits the output of the speed PI controller. The limitation of the speed PI controller output protects the motor current from exceeding the maximal allowed motor current.

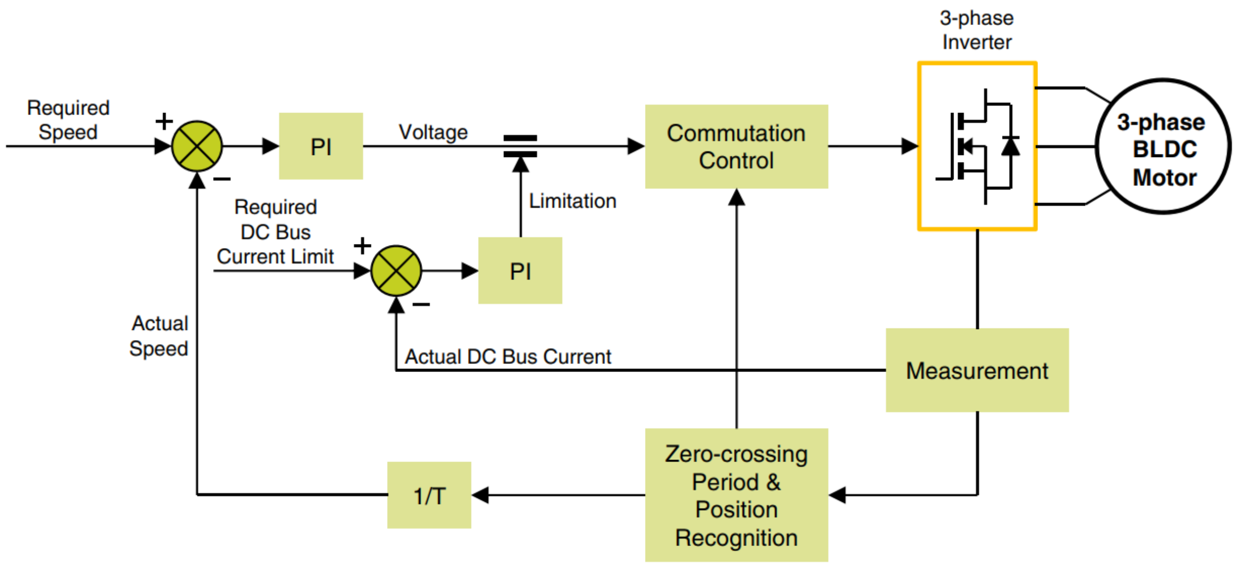


Figure 14. **Speed control with current limitation**

4. Software implementation on the S32K116

4.1. S32K116 – Key modules for BLDC six-step control

The S32K116 device includes modules such as the FlexTimer Module (FTM), Trigger MUX Control (TRGMUX) a Programmable delay block (PDB), an 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 15* shows module interconnection for BLDC sensorless application using zero-crossing detection algorithm and sensor-based application based on Hall sensor. 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 2 x FTM, TRGMUX, PDB, and ADC modules as shown in *Figure 15*. 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 FTM0 PWM initialization trigger *init_trig* or by FTM0 PWM 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 FTM0 Channel Value register CnV.

FTM0 trigger signal is routed to hardware trigger input of the PDB module through flexible TRGMUX unit. In S32K11x, there is one ADC module and one PDB module that work in pair meaning PDB0 is directly linked with ADC0.

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_SC0[COCOx]. This means that next pre-trigger can be generated only if the ongoing conversion is completed.

Another FTM1 is used for commutation control. In sensorless mode, it is configured as a simple timer that schedules and forces commutation events which are determined from the actual BEMF zero-crossing period. For Hall based driven control, commutation is managed by FTM1 and three GPIs which are sensitive on rising/falling edge. Every edge detected on any GPIO input indicates new commutation event. FTM1 is reset and generates init trigger that forces FTM0 PWM module to settings of the new commutation sector. Thus, actual rotor position and commutation sector is derived from the three GPIOs input logic.

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

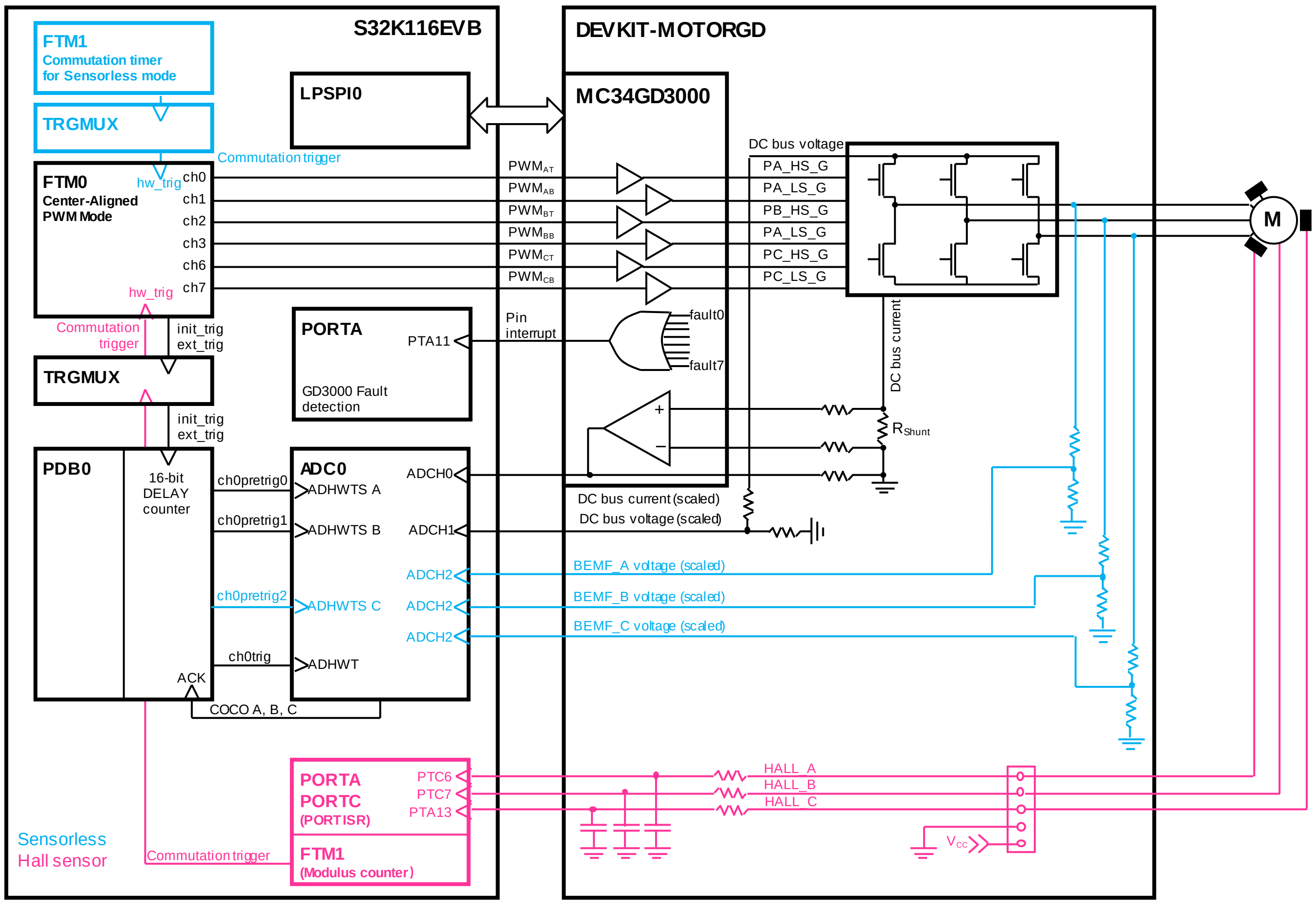


Figure 15. **S32K116 module interconnection**

4.1.2. S32K116 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. S32K116 detects fault state of the MC34GD3000 by means of interrupt signal on PTA11 pin. Integrated current sensing amplifier with analog comparator allow to measure DC bus current and detect overcurrent. Interconnection between S32K116 and MC34GD3000 is briefly depicted in *Figure 15*.

4.1.3. Module involvement in digital BLDC control loop

This section will discuss timing and modules synchronization to accomplish BLDC Six-step control on the S32K116 and the internal hardware features. The time diagram of the automatic synchronization between PWM and ADC in the BLDC application is shown in *Figure 16*.

In Sensorless mode, each commutation event gets triggered the FTM1 *init_trig* signal. This trigger signal is routed to FTM0 trigger input through TRGMUX module, causing the reset of the FTM0 counter to its initial value. It also generates the FTM0 PWM initialization trigger event starting the configurable PDB0 counter. ADC0 is triggered based on the PDB0 pre-trigger delays. When PDB counter reaches first pre-trigger delay value, PDB initiates first ADC channel measurement.

DC bus current measurement is triggered first, at beginning of the PWM cycle by *pretrig0*. DC Bus voltage and BEMF voltage are sampled consecutively towards the end of the active PWM pulse. While DC Bus voltage measurement is triggered at *pretrig1*, BEMF voltage measurement is triggered by PDB0 at *pretrig2*. The ADC conversion results are automatically stored into a predefined queue in memory. This sampling approach respect measurement principles of BEMF phase voltage, DC bus current, and DC bus voltage measurement described in *3.3.3* and *3.3.4*.

The ADC conversion complete interrupt notifies the CPU that the ADC conversion result values are available for reading and further processing to identify the zero-crossing event and determine rotor speed for speed control loop. Commutation event is then calculated based on the actual zero-crossing period.

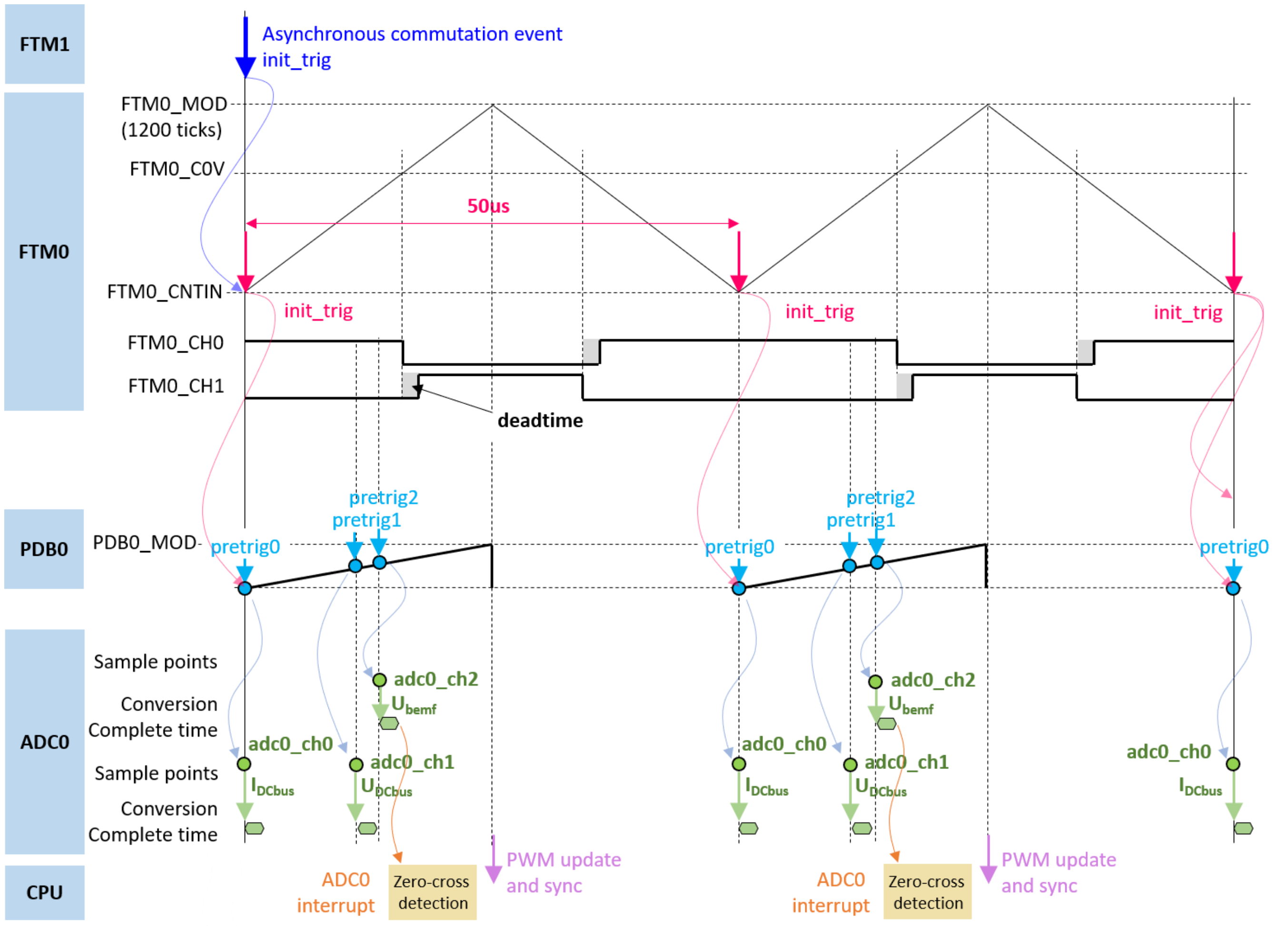


Figure 16. **Module involvement in the sensorless BLDC software control loop**

Module involvement and timing diagram in Hall sensor operation is slightly different, since the commutation control is managed by hardware mainly. Three MCU pins configured as GPI pins sensitive on rising/falling edge are used to detect commutation event from the Hall sensor. Every time there is a rising/falling edge detected on any of three GPI inputs, corresponding GPI invokes interrupt. In the interrupt service routine, counter of the FTM1 is reset and FTM1 generates *init_trig*. This trigger restarts FTM0 counter to initial value and generates the FTM0 PWM initialization trigger event in same way as FTM0 does in Sensorless operation. Since BEMF voltage measurement for zero-cross detection is not needed, *pretrig2* of the PDB0 is disabled. To control the torque/speed properly, Hall based application needs to measure DC bus current and DC bus voltage. This measurement is ensured by PDB0 and ADC0 with the same timing diagram as in Sensorless mode. CPU load is reduced due to the absence of the zero-crossing detection algorithm.

4.2. S32K116 Device initialization

To simplify and accelerate application development, embedded part of the BLDC Sensorless motor control application has been created using S32 Software Development Kit – S32 SDK. S32K116 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 S32K116 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
- McuLpitConfig() – LPIT 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*).

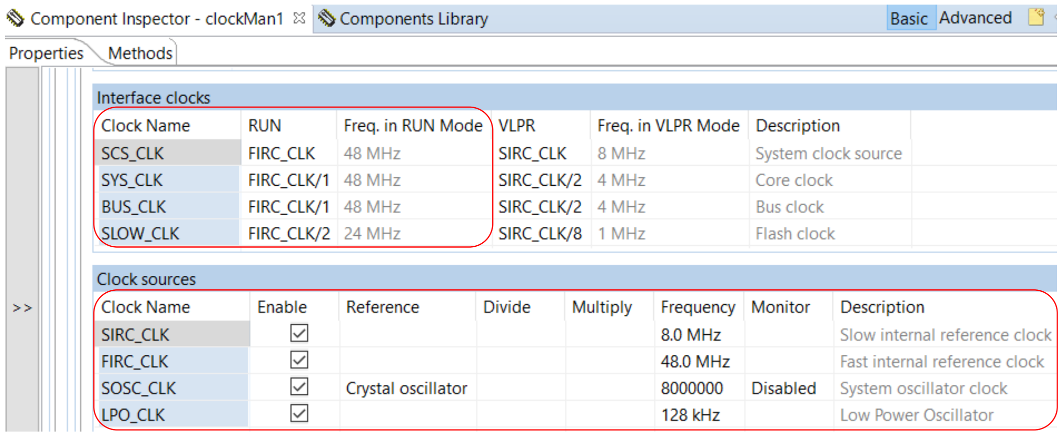
4.2.1. Clock configuration and power management

S32K116 features a complex clocking sourcing, distribution and power management. To run a core of the S32K116 as well as some MCU peripherals at maximum frequency 48 MHz in normal RUN mode, Fast Internal Reference Clock (FIRC) with frequency 48 MHz is selected to supply core, system and bus clock. This frequency is divided by two to supply flash with lower frequency (24 MHz). This clock configuration belongs to one of the typical and recommended. It is summarized in *Table 2.*

Table 2 Table S32K116 clock configuration in RUN mode

Clock	Frequency
CORE_CLOCK	48 MHz
SYS_CLK	48 MHz
BUS_CLK	48 MHz
FLASH_CLK	24 MHz

This clock configuration and power management can be setup easily by S32 Processor Expert. Preview of the S32K116 clock sourcing and distribution by means of Processor Expert is shown in *Figure 17.*



Interface clocks

Clock Name	RUN	Freq. in RUN Mode	VLPR	Freq. in VLPR Mode	Description
SCS_CLK	FIRC_CLK	48 MHz	SIRC_CLK	8 MHz	System clock source
SYS_CLK	FIRC_CLK/1	48 MHz	SIRC_CLK/2	4 MHz	Core clock
BUS_CLK	FIRC_CLK/1	48 MHz	SIRC_CLK/2	4 MHz	Bus clock
SLOW_CLK	FIRC_CLK/2	24 MHz	SIRC_CLK/8	1 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
LPO_CLK	☑				128 kHz		Low Power Oscillator

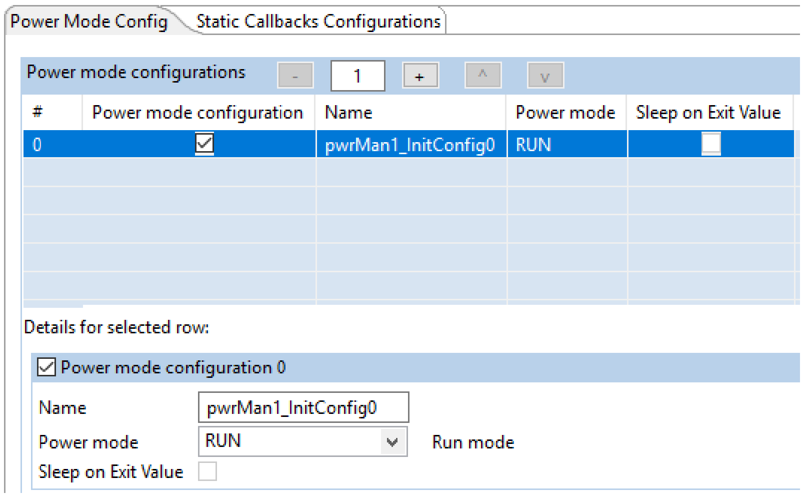
Figure 17. **S32K116 clock configuration in Processor Expert**

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. S32K116 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 it was discussed at begging of this chapter, power management of the S32K116 is configured for normal RUN mode. This power mode can be set in Processor Expert as well, *Figure 18*.



#	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 18. **S32K116 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 S32K116 device, *Example 2*.

Example 2. S32K116 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 S32K116 peripherals, which are discussed below.

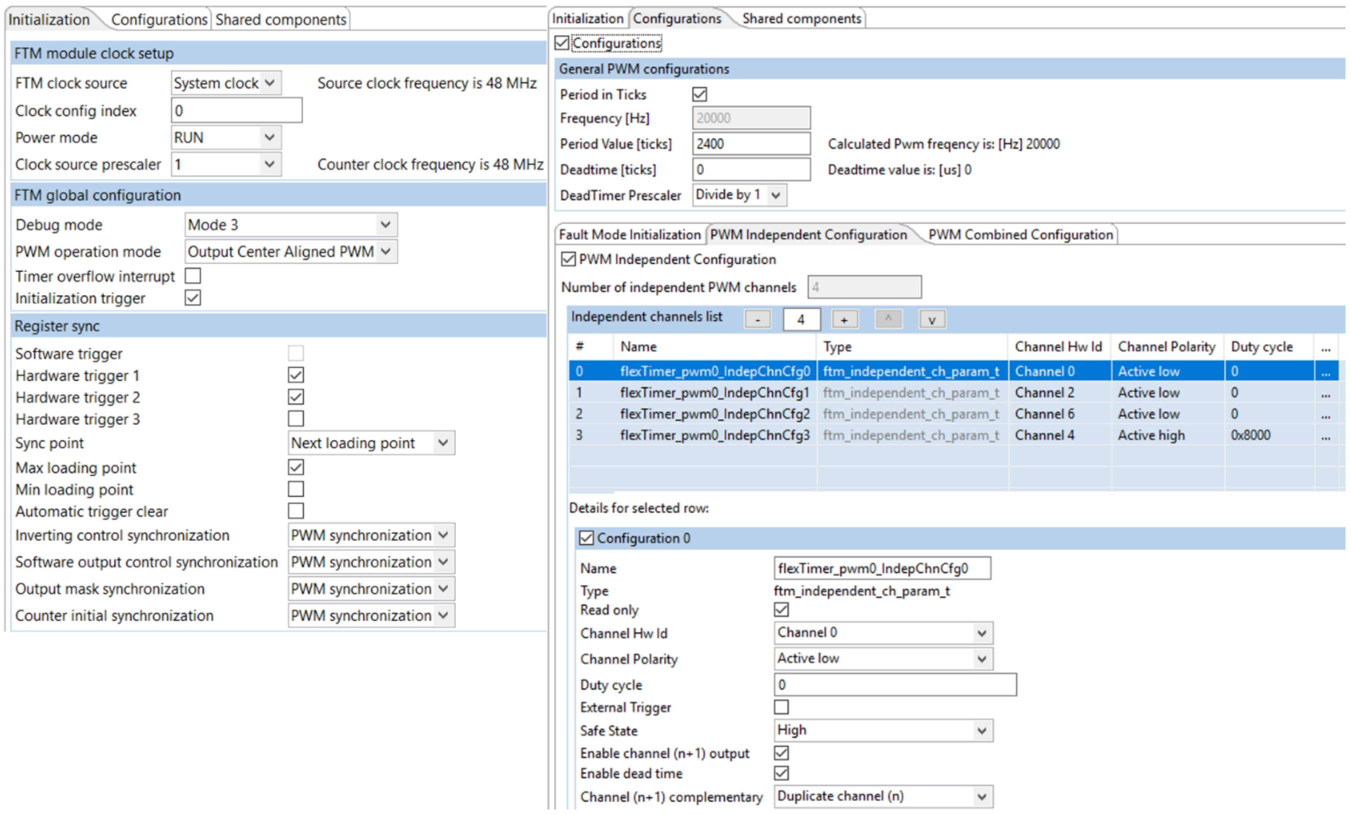
4.2.2. FlexTimer Module (FTM)

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

FTM0 instance is used in BLDC 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 16*, up 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 FTM0 must have duplicate polarity. 20 kHz PWM frequency is adjusted by FTM0 Modulo register (FTM0_MOD = 1200) taking 48MHz clock source frequency into account. To protect power MOSFETs against short circuit, the MC34GD3000 pre-driver will automatically insert during initializing MC34GD3000. This FTM0 configuration can be carried out by using Processor Expert, *Figure 19*.


Initialization | Configurations | Shared components

FTM module clock setup

FTM clock source	System clock	Source clock frequency is 48 MHz
Clock config index	0	
Power mode	RUN	
Clock source prescaler	1	Counter clock frequency is 48 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

Initialization | Configurations | Shared components

✓ Configurations

General PWM configurations

Period in Ticks	✓	
Frequency [Hz]	20000	
Period Value [ticks]	2400	Calculated Pwm freqency is: [Hz] 20000
Deadtime [ticks]	0	Deadtime value is: [us] 0
DeadTimer Prescaler	Divide by 1	

Fault Mode Initialization | PWM Independent Configuration | PWM Combined Configuration

✓ PWM Independent Configuration

Number of independent PWM channels 4

Independent channels list 4

#	Name	Type	Channel Hw Id	Channel Polarity	Duty cycle	...
0	flexTimer_pwm0_IndepChnCfg0	ftm_independent_ch_param_t	Channel 0	Active low	0	...
1	flexTimer_pwm0_IndepChnCfg1	ftm_independent_ch_param_t	Channel 2	Active low	0	...
2	flexTimer_pwm0_IndepChnCfg2	ftm_independent_ch_param_t	Channel 6	Active low	0	...
3	flexTimer_pwm0_IndepChnCfg3	ftm_independent_ch_param_t	Channel 4	Active high	0x8000	...

Details for selected row:

✓ Configuration 0

Name	flexTimer_pwm0_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 19. **S32K116 FTM0 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, deatime value, channels pairs setting are configured in *Configuration* tab on the right, *Figure 19*.

The double-buffered registers FTM0_SWOCTRL and FTM0_OUTMASK are used to control the unipolar PWM pattern as discussed in 3.2. The FTM0_SWCTRL register controls the PWM output by forcing selected channels into a defined state. The FTM0_OUTMASK register controls the PWM output by forcing selected channels into an inactive state. The double-buffered values are applied at each commutation event triggered by either by FTM1 *init_trig* in Sensorless mode or Hall sensor mode. To allow this triggering mechanism, *Hardware trigger 1* is enabled in *Initialization* tab, *Figure 19*. *Table 3* shows the SWOCTRL and OUTMASK values applied at a commutation event in a particular sector of the six-step commutation sequence.

Table 3 Software control and output mask definition in six-step commutation sequence

Sector	FTM0_SWOCTRL	FTM0_OUTMASK
0	0x0808	0xc4
1	0x8080	0x4C
2	0x8080	0x43
3	0x0202	0xc1
4	0x0202	0x0D
5	0x0808	0x07
Alignment[1]	0x0A0A	0x05
PWM off	0x0000	0xcF

[1] Alignment vector is set to allow a commutation sequence starting from sector 0

To allow the application of the double-buffered values outside the commutation event, *Hardware trigger 2* is enabled in *Initialization* tab as well, *Figure 19*. This hardware trigger is generated by writing 1 to the SIM_FTMOPT1[FTM0SYNCBIT] bit.

The duty cycle of the center-aligned PWM is controlled by the FTM0_CnV (n = 0, 2, 6) register values. In up counting mode, even channels define both, leading as well as trailing edges. Even channels are set according to *Equation 8*

$$FTM0\_CnV = duty\_cycle \times FTM0\_MOD, \qquad where \qquad duty\_cycle = [0,1]$$
$$FTM\_MOD = 1200$$

Equation 8

As discussed in chapter *4.1.3*, to initiate control loop at beginning of the PWM period, *Initialization trigger* is enabled in *Initialization* tab as well, *Figure 19*.

Once the FTM0 setting is completed, Processor Expert generates two configuration structures *flexTimer_pwm0_InitConfig* and *flexTimer_pwm0_PwmConfig* that access and set corresponding FTM0 registers executing *FTM_DRV_Init* and *FTM_DRV_InitPwm* functions, *Example 3*.

Example 3. S32K116 FTM0 configured by S32 SDK

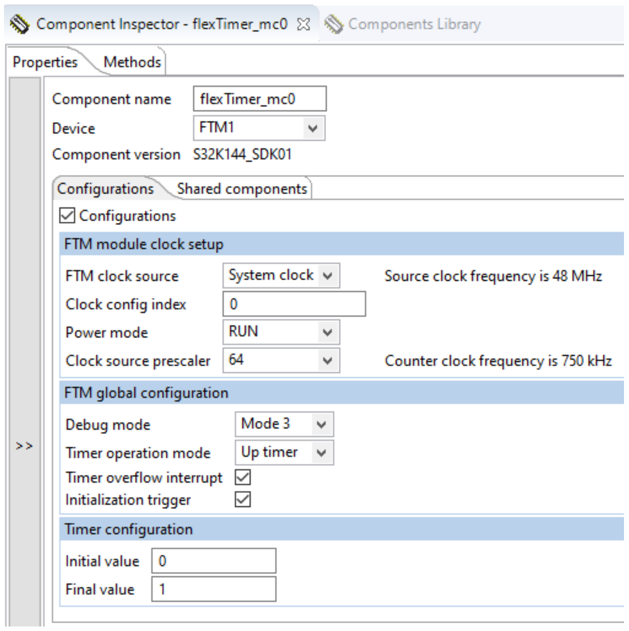
```
void McuFtmConfig(void)
{
    /* FTM0 module initialized as PWM signals generator */
    FTM_DRV_Init(INST_FLEXTIMER_PWM0, &flexTimer_pwm0_InitConfig, &statePwm);

    /* FTM0 module PWM initialization */
    FTM_DRV_InitPwm(INST_FLEXTIMER_PWM0, &flexTimer_pwm0_PwmConfig);

    /* Mask all FTM0 channels to disable PWM output */
    FTM_DRV_MaskOutputChannels(INST_FLEXTIMER_PWM0, PWM_CHANNEL_GROUP, false);

   /* Set FTM0SYNCBIT to trigger and update FTM0 registers */
    SIM->FTMOPT1 |= SIM_FTMOPT1_FTM0SYNCBIT_MASK;
}
```

FTM_DRV_MaskOutputChannels function disables PWM output masking all FTM channels.

4.2.2.2. Commutation timer for Sensorless mode

FTM1 is used in Sensorless mode to schedule and identify the commutation event. Initialization trigger signal *init_trig* is internally routed to the FTM0 module trigger 1 input in order to perform commutation of the PWM pairs. The commutation event is scheduled by changing the PWM period (counter module value FTM1_MOD). When the counter overflows, a rising edge is generated and an interrupt is invoked. The PWM generated by channel 0 has the duty cycle equal to 1 counter tick (FTM1_C0V = 1).

To be able to schedule long commutation periods at low speeds, the FTM1 counter is configured to run at 750 kHz frequency. This module settings can be configured by Processor expert *Figure 20* and executing SDK APIs shown in *Example 4*. HALL_SENSOR macro must be set to 0 to allow FTM1 configuration for Sensorless operation.



Component name: flexTimer_mc0
Device: FTM1
Component version: S32K144_SDK01

Configurations | Shared components

☑ Configurations

FTM module clock setup

FTM clock source	System clock	Source clock frequency is 48 MHz
Clock config index	0	
Power mode	RUN	
Clock source prescaler	64	Counter clock frequency is 750 kHz

FTM global configuration

Debug mode	Mode 3
Timer operation mode	Up timer
Timer overflow interrupt	☑
Initialization trigger	☑

Timer configuration

Initial value	0
Final value	1

Figure 20. **S32K116 FTM1 configuration in Processor Expert**

Example 4. S32K116 FTM1 configured by S32 SDK

```
void McuFtmConfig(void)
{
    /* FTM1 initialization */
    FTM_DRV_Init(INST_FLEXTIMER_MC0, &flexTimer_mc0_InitConfig, &stateMc0);

    /* FTM1 initialized as a simple up-counting timer with frequency 750 kHz */
    FTM_DRV_InitCounter(INST_FLEXTIMER_MC0, &flexTimer_mc0_TimerConfig);

}
```

4.2.2.3. Commutation timer for Hall sensor mode

Same configuration of the FTM1 is also used for Hall based operation. To detect commutation event, three pins (PTC7, PTC6, PTA13) are configured as GPIs with activated rising/falling edge interrupt. They share one PORT interrupt as shown in *Figure 21.* If rising or falling edge is detected on any of these three GPIs , PORT interrupt is generated and code shown in *Example 4* is executed. To be able to determine actual rotor speed, counter value of the FTM1 is captured by *FTM_DRV_CounterRead* and stored to variable *timeHallcommutation*. Consequently, FTM1 counter is reset to initial value and *init_trig* is generated to FTM0 input through flexible TRGMUX unit. This event updates PWM pattern through double-buffered registers FTM3_SWCTRL and FTM3_ OUTMASK. These are updated by values of the new commutation sector according to *Table 3*.

The free running counter is refreshed on every edge, so that the rotor speed can be established based on the captured commutation time T_{COM} every edge applying *Equation 2*. Rotor position is determined according to the Hall logic captured by GPIOs input logic.

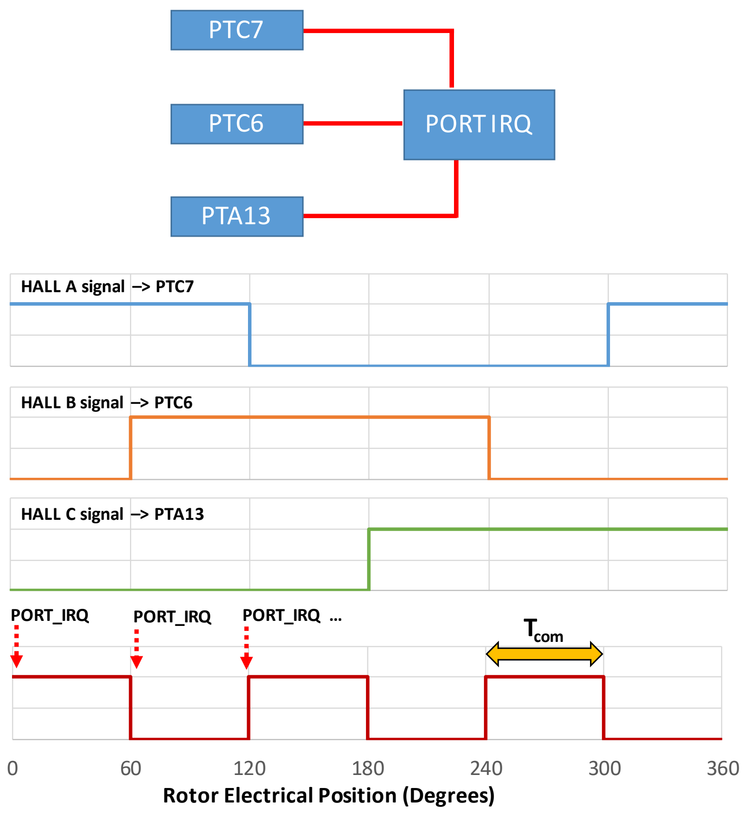


Figure 21. **Three GPI pins and FTM1 to capture commutation time**

Example 5. commutation event and rotor speed controlled by PORT_IRQHandler

```
void PORT_IRQHandler(void)
{
#if HALL_SENSOR
        uint16_t timeHallcommutation = 0u;

        /* PORTIRQ XOR visualization in FreeMASTER */
        PORTIRQXorSignal ^= 1;

        timeHallcommutation = FTM_DRV_CounterRead(INST_FLEXTIMER_MC0);

        /* Reset FTM1 counter */
        FTM1->CNT = 0;

        /* Get commutation sector based on the Hall logic */
        HALL_GetSector(&SensorHall);

        /* Commutation period is measured by modulus counter mode of the FTM1 */
        SensorHall.Period[SensorHall.Sector] = timeHallcommutation;

        if (driveStatus.B.EnableCMT)
        {
                        /* Prepare PWM settings for the next commutation sector */
                ACTUATE_SetPwmMask(ui8FTM0OutmaskVal[rotationDir][SensorHall.Sector],
                                ui16FTM0SwOctrlVal[rotationDir][SensorHall.Sector], HW_INPUT_TRIG0);
        }

        driveStatus.B.StallCheckReq = 1;
        driveStatus.B.HallEvent = 1;

        PINS_DRV_ClearPinIntFlagCmd(PORTA, 13u);
        PINS_DRV_ClearPinIntFlagCmd(PORTC, 6u);
        PINS_DRV_ClearPinIntFlagCmd(PORTC, 7u);
        …
}
```

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 module by FTM0 initialization trigger signal *init_trig*, selection bit field SEL0 of the TRGMUX_PDB0 registers must be specified to define trigger source.

Processor Expert generates configuration structure *trgmux1_InitConfig0* that sets all TRGMUX registers to assign trigger inputs with trigger outputs as demanded, *Example 6*. In particular, FTM0 initialization trigger is assigned to PDB0 trigger input as well as to TRGMUX output 0 PDB0 pre-triggers are routed to TRGMUX output 0, ADC1 conversion complete flag is assigned to TRGMUX output 1, FTM1 initialization trigger is assigned to FTM0 HW trigger 0.

Example 6. S32K116 TRGMUX module controlled by S32 SDK

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

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

};
const trgmux_inout_mapping_config_t trgmux1_InOutMappingConfig0[5] =
{
    {TRGMUX_TRIG_SOURCE_FTM1_INIT_TRIG, TRGMUX_TARGET_MODULE_FTM0_HWTRIG0, false},
    {TRGMUX_TRIG_SOURCE_FTM0_INIT_TRIG, TRGMUX_TARGET_MODULE_PDB0_TRG_IN, false},
    {TRGMUX_TRIG_SOURCE_FTM0_INIT_TRIG, TRGMUX_TARGET_MODULE_TRGMUX_OUT3, false},
    {TRGMUX_TRIG_SOURCE_PDB0_CH0_TRIG, TRGMUX_TARGET_MODULE_TRGMUX_OUT0, false},
    {TRGMUX_TRIG_SOURCE_ADC0_SC1A_COCO, TRGMUX_TARGET_MODULE_TRGMUX_OUT1, false},
};
```

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 FTM0 initialization trigger and schedules ADC channels sampling through PDB pre-triggers delays. When FTM0 initialization trigger is detected on the PDB0 trigger input, PDB0 generates hardware signal to trigger ADC0 channels in order defined by pre-trigger delays, *Figure 22*.

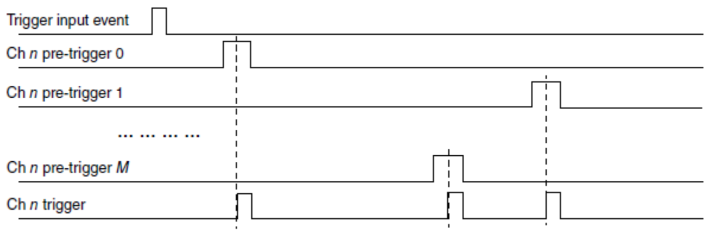


Figure 22. **PDB pre-triggers and trigger output**

PDB pre-trigger delays can be set independently using CHnDLYm registers. Since the PDB0 and FTM0 modules are synchronized and share the same source frequency 48MHz, values of the CHnDLYm registers are set using the same time base as for PWM. PDB pre-trigger can be also configured to work in back-to-back mode, when ADC conversion complete triggers next PDB channel pre-trigger and triggers output. This mode well suits to DC bus voltage and BEMF voltage, which should be sampled in one time instant ideally. Back to back operation ensures minimal delay between two consecutive ADC samples that is ADC conversion time and two peripherals clock cycles. *0* shows all PDB0 pre-triggers used in BLDC six-step motor control application.

Table 4 PDB0 and PDB1 pre-triggers

FOC state variable	PDB pre-trigger	CHnDLYm value [ticks]	Relation to PWM
DC bus current	pdb0_ch0_pretrig0	pdb_delay0	At 50% of the active PWM pulse
DC bus voltage	pdb0_ch0_pretrig1	pdb_delay1	At 80% of the active PWM pulse
Phase BEMF voltage	pdb0_ch0_pretrig2	back-to-back mode chained with pdb0_ch0_pretrig1 (delay is ignored)	At 80% of the active PWM pulse

DC bus current measurement is triggered every PWM cycle at beginning of the PWM period by *pdb0_ch0_pretrig0*. This delay is static value defined only once at the initialization phase. To measure DC bus voltage and BEMF voltage consecutively towards the end of the active PWM pulse, *pdb0_ch0_pretrig1* is dynamically modified according to actual duty cycle, *Equation 9*.

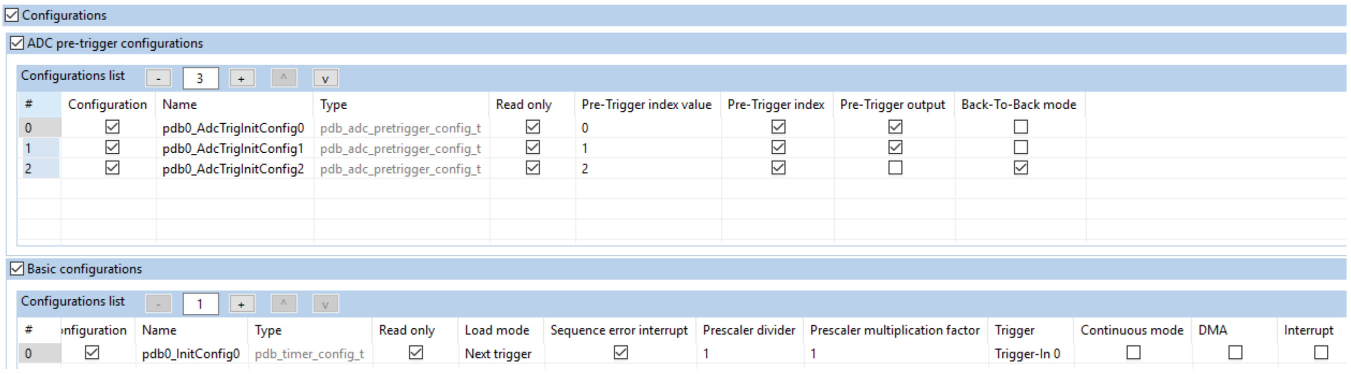
$$pdb\_delay = 0.8 \times duty\_cycle \times FTM0\_MOD, \qquad where \qquad pdb\_delay = [100, 1200]$$

Equation 9

A software routine limits *pdb_delay* to 100 ticks to prevent collision *pdb0_ch0_pretrig1*, at low duty cycles. This limit respects ADC conversion time that typically takes ~1.1µs considering short ADC sample time and 48MHz ADC input frequency. PDB Sequence Error Interrupt can be activated as well as hardware detector.

It should be noticed that CHnDLYx are double buffered registers meaning *pdb_delay* value is first latched into CHnDLYx buffers and then loaded from their buffers at beginning of the PWM period when 1 is written to SC[LDOK] bit and FTM0 *init_trig* signal is detected on PDB0 input.

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 23*.



☑ Configurations

☑ ADC pre-trigger configurations

Configurations list - 3 + ^ v

#	Configuration	Name	Type	Read only	Pre-Trigger index value	Pre-Trigger index	Pre-Trigger output	Back-To-Back mode
0	☑	pdb0_AdcTrigInitConfig0	pdb_adc_pretrigger_config_t	☑	0	☑	☑	☐
1	☑	pdb0_AdcTrigInitConfig1	pdb_adc_pretrigger_config_t	☑	1	☑	☑	☐
2	☑	pdb0_AdcTrigInitConfig2	pdb_adc_pretrigger_config_t	☑	2	☑	☐	☑

☑ Basic configurations

Configurations list - 1 + ^ v

#	nfiguration	Name	Type	Read only	Load mode	Sequence error interrupt	Prescaler divider	Prescaler multiplication factor	Trigger	Continuous mode	DMA	Interrupt
0	☑	pdb0_InitConfig0	pdb_timer_config_t	☑	Next trigger	☑	1	1	Trigger-In 0	☐	☐	☐

Figure 23. **S32K116 PDB0 module and pre-triggers configuration in Processor Expert**

Processor Expert generates configuration structures *pdbN_InitConfigX* and *pdbN_AdcTrigInitConfigX* that access appropriate PDB registers *Example 7*. To set PDB modulo and PDB pre-triggers delays, *PDB_DRV_SetTimerModulusValue* and *PDB_DRV_SetAdcPreTriggerDelayValue* are used and specified by values listed in *0*. Double-buffered registers of the PBD modules are loaded using *PDB_DRV_LoadValuesCmd* command.

Example 7. S32K116 PDB instances controlled by S32 SDK

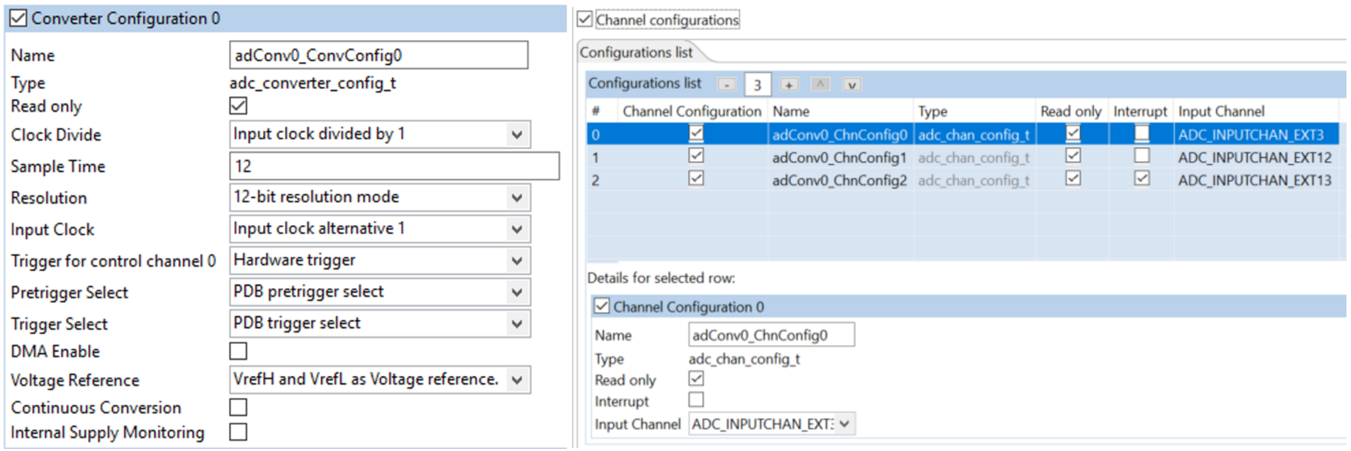
```
void McuPdbConfig(void)
{
    /* PDB0 module initialization */
    PDB_DRV_Init(INST_PDB0, &pdb0_InitConfig0);

    /* PDB0 CH0 pre-trigger0 initialization */
    PDB_DRV_ConfigAdcPreTrigger(INST_PDB0, 0, &pdb0_AdcTrigInitConfig0);
    /* PDB0 CH0 pre-trigger1 initialization */
    PDB_DRV_ConfigAdcPreTrigger(INST_PDB0, 0, &pdb0_AdcTrigInitConfig1);
    /* PDB0 CH0 pre-trigger1 initialization */
    PDB_DRV_ConfigAdcPreTrigger(INST_PDB0, 0, &pdb0_AdcTrigInitConfig2);

    /* Set PDB0 modulus value set to half of the PWM cycle */
    PDB_DRV_SetTimerModulusValue(INST_PDB0, PWM_MODULO);

    /* PDB0 CH0 pre-trigger0 delay set to sense DC bus current in the middle of the PWM cycle */
    PDB_DRV_SetAdcPreTriggerDelayValue(INST_PDB0, 0, 0, 0);
    /* PDB0 CH0 pre-trigger1 delay set to sense DC bus voltage towards the end of the active PWM pulse */
    /* Note: BEMF voltage will be automatically trigger after DC bus voltage convert completed */
    /* Initially set as for minimal duty cycle 9% - 10% of PWM period = 230 */
    PDB_DRV_SetAdcPreTriggerDelayValue(INST_PDB0, 0, 1, PDB_DELAY_MIN);

    /* Enable PDB0 prior to PDB0 load */
    PDB_DRV_Enable(INST_PDB0);

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

4.2.5. Analog-to-Digital Converter (ADC)

The S32K116 device has one 12-bit Analog-to-Digital Converter (ADC). This is 32-channel multiplexed input successive approximation ADC with 16 result registers.

The ADC instance is triggered by PDB. 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 1 resulting 48MHz 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 24*.


☑ Converter Configuration 0

Name	adConv0_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 configurations

Configurations list

Configurations list 3

#	Channel Configuration	Name	Type	Read only	Interrupt	Input Channel
0	☑	adConv0_ChnConfig0	adc_chan_config_t	☑	☐	ADC_INPUTCHAN_EXT3
1	☑	adConv0_ChnConfig1	adc_chan_config_t	☑	☐	ADC_INPUTCHAN_EXT12
2	☑	adConv0_ChnConfig2	adc_chan_config_t	☑	☑	ADC_INPUTCHAN_EXT13

Details for selected row:

☑ Channel Configuration 0

Name	adConv0_ChnConfig0
Type	adc_chan_config_t
Read only	☑
Interrupt	☐
Input Channel	ADC_INPUTCHAN_EXT3

Figure 24. **S32K116 ADC0 module and channels configuration in Processor Expert**

ADC0 measures BEMF voltage of the disconnected phase by *adc0_ch2*. ADC0 input channel is selected according to the actual commutation sector, *Table 5*.

Table 5 ADC0 input channel selection according to the actual sector

Sector	BEMF voltage	ADC0 input channel
0	Phase C	ADC_INPUTCHAN_EXT13
1	Phase B	ADC_INPUTCHAN_EXT14
2	Phase A	ADC_INPUTCHAN_EXT9
3	Phase C	ADC_INPUTCHAN_EXT13
4	Phase B	ADC_INPUTCHAN_EXT14
5	Phase A	ADC_INPUTCHAN_EXT9

DC bus current and DC bus voltage are measured by *adc0_ch0* and *adc0_ch1*, respectively. Conversion Complete Interrupt is activated for *adc0_ch2* to invoke interrupt as soon as last conversion is completed. To measure BEMF voltage, *ADC_INPUTCHAN_EXT13* is selected as an input channel, *Figure 24*.

Example 8 shows ADC0 module configuration. Processor Expert generates module configuration structure *adConv0_ConvConfig0* as well as channel configuration structures *adConv0_ChnConfigX*, which are present at the bottom of the example. These configuration structures take effect calling SDK APIs in *McuAdcConfig* function, *Example 8*.

Example 8. S32K116 ADC instances and channels controlled by S32 SDK

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

    ADC_DRV_AutoCalibration(INST_ADCONV0);

   /* ADC0_SE3 input channel is used for DC bus current sensing */
    ADC_DRV_ConfigChan(INST_ADCONV0, 0, &adConv0_ChnConfig0);

   /* ADC0_SE13 input channel is used for DC bus voltage sensing */
```

```
    ADC_DRV_ConfigChan(INST_ADCONV0, 1, &adConv0_ChnConfig1);

    /* ADC0_SE12 input channel is used for BEMF voltage sensing */
    ADC_DRV_ConfigChan(INST_ADCONV0, 2, &adConv0_ChnConfig2);
}

…

/*! adConv1 configuration structure */
const adc_converter_config_t adConv0_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 adConv0_ChnConfig0 = {
  .interruptEnable = false,
  .channel = ADC_INPUTCHAN_EXT3,
};

const adc_chan_config_t adConv0_ChnConfig1 = {
  .interruptEnable = false,
  .channel = ADC_INPUTCHAN_EXT12,

const adc_chan_config_t adConv0_ChnConfig2 = {
  .interruptEnable = true,
  .channel = ADC_INPUTCHAN_EXT13,

};
```

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

LPSPI is used as communication interface between S32K116 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 S32K116 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 9 represents initialization of the S32K116 LPSPI0, MC34GD3000 and some important S32K116 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 9*.

LPSPI0 communication frequency 2 MHz is derived from the LPSPI0 input frequency 48 MHz sourced from the Fast Internal Reference Clock (FIRC).

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 9. S32K116 LPSPI0 and MC34GD3000 controlled by TPP (S32 SDK)

```
void GD3000_Init(void)
{
    /* GD3000 pin configuration - EN1:PTA3 EN2:PTA3 & RST:PTA2 */
    tppDrvConfig.en1PinIndex     = 3U;
    tppDrvConfig.en1PinInstance = instanceA;
    tppDrvConfig.en2PinIndex     = 3U;
    tppDrvConfig.en2PinInstance = instanceA;
    tppDrvConfig.rstPinIndex     = 2U;
    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 =    48000000U;

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

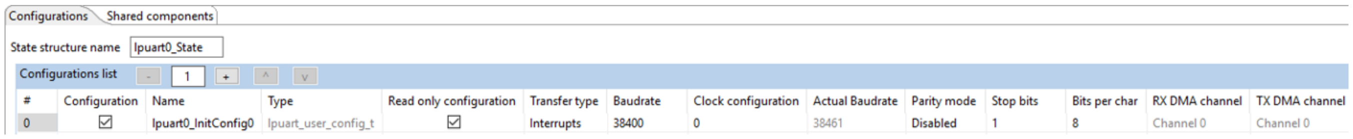
4.2.7. Low Power Universal Asynchronous Receiver/Transmitter (LPUART)

LPUART0 is used as a communication interface between S32K116 processor and FreeMASTER run-time debugging and visualization tool. Function *McuLpuartConfig* initializes LPUART0 module with baud rate 38400, 1 stop bit and 8 bits per channel. This configuration is carried out by SDK's LPUART driver, *Example 10.*

Example 10. S32K116 LPUART1 controlled by S32 SDK

```
void McuLpuartConfig(void)
{
    /* LPUART module initialization */
    LPUART_DRV_Init(INST_LPUART0, &lpuart0_State, &lpuart0_InitConfig0);
}

/*! Lpuart0 configuration structure */
const lpuart_user_config_t lpuart0_InitConfig0 = {
  .transferType = LPUART_USING_INTERRUPTS,
  .baudRate = 38400U,
  .parityMode = LPUART_PARITY_DISABLED,
  .stopBitCount = LPUART_ONE_STOP_BIT,
  .bitCountPerChar = LPUART_8_BITS_PER_CHAR,
  .rxDMAChannel = 0U,
  .txDMAChannel = 0U,
};
```

Configuration structure *lpuart0_InitConfig0* can be modified manually or configured by means of Processor Expert as shown in *Figure 25*.



Configurations | Shared components

State structure name: lpuart0_State

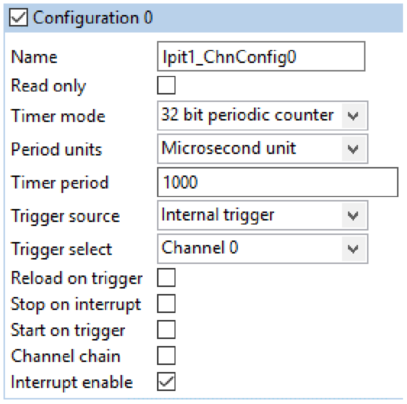
Configurations list: 1

#	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	☑	lpuart0_InitConfig0	lpuart_user_config_t	☑	Interrupts	38400	0	38461	Disabled	1	8	Channel 0	Channel 0

Figure 25. **S32K116 LPUART1 module configuration in Processor Expert**

4.2.8. Low Power Interrupt Timer (LPIT)

The LPIT channel 0 is employed to control the speed and motor current in a software task. LPIT channel 0 is configured to generate a periodic interrupt every 1 ms. This module setting can be configured by means of Processor expert and SDK commands as shown in *Figure 26* and *Example 11*.



☑ Configuration 0

Name	lpit1_ChnConfig0
Read only	☐
Timer mode	32 bit periodic counter
Period units	Microsecond unit
Timer period	1000
Trigger source	Internal trigger
Trigger select	Channel 0
Reload on trigger	☐
Stop on interrupt	☐
Start on trigger	☐
Channel chain	☐
Interrupt enable	☑

Figure 26. **S32K116 LPIT module configuration in Processor Expert**

Example 11. S32K116 LPIT module controlled by S32 SDK

```
void McuLpitConfig(void)
{
    /* LPIT module initialization */
    LPIT_DRV_Init(INST_LPIT1, &lpit1_InitConfig);

    /* LPIT channel0 initialization */
    LPIT_DRV_InitChannel(INST_LPIT1, 0, &lpit1_ChnConfig0);

}


/*! Global configuration of lpit0 */
const lpit_user_config_t lpit1_InitConfig =
{
    .enableRunInDebug = false,  /*!< true: LPIT run in debug mode; false: LPIT stop in debug mode */
    .enableRunInDoze = false    /*!< true: LPIT run in doze mode; false: LPIT stop in doze mode */
};

/*! User channel configuration 0 */
lpit_user_channel_config_t lpit1_ChnConfig0 =
{
    .timerMode = LPIT_PERIODIC_COUNTER,
    .periodUnits = LPIT_PERIOD_UNITS_MICROSECONDS,
    .period = 1000U,
    .triggerSource = LPIT_TRIGGER_SOURCE_INTERNAL,
    .triggerSelect = 0U,
    .enableReloadOnTrigger = false,
    .enableStopOnInterrupt = false,
    .enableStartOnTrigger = false,
    .chainChannel = false,
    .isInterruptEnabled = true
};
```

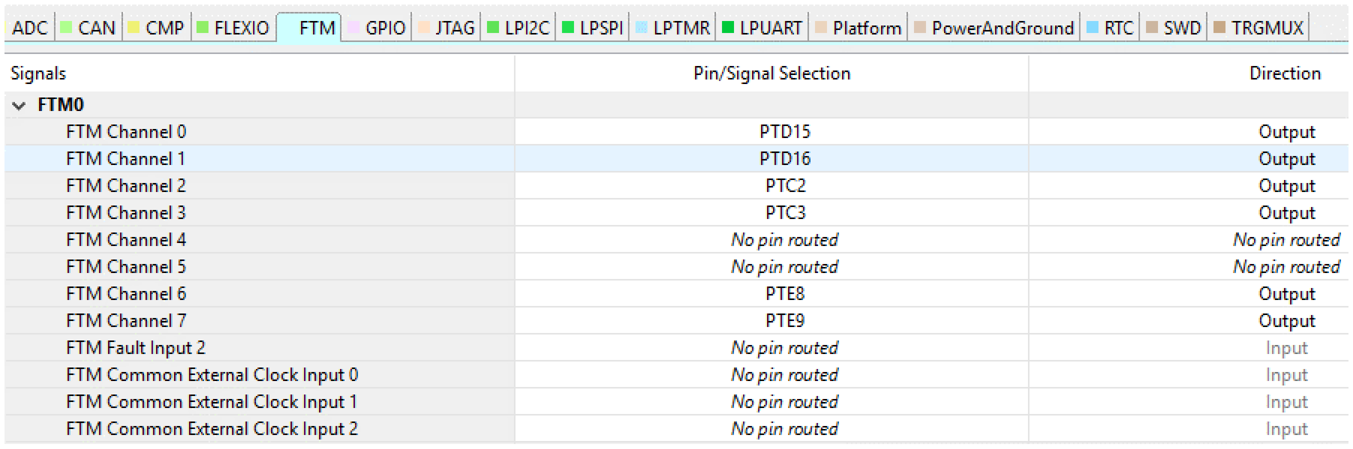
4.2.9. Port control and pin multiplexing

BLDC Six-Step motor control application requires following on chip pins assignment, ***Table** 6*.

Table 6 **Pins assignment for S32K116 BLDC Six-Step motor control**

Module	Signal name	Pin name / Functionality	Description
FTM0	PWMA_HS_B	PTD15 / FTM0_CH0	PWM signal for phase A high-side driver (inverted)
	PWMA_LS	PTD16 / FTM0_CH1	PWM signal for phase A low-side driver
	PWMB_HS_B	PTC2 / FTM0_CH2	PWM signal for phase B high-side driver (inverted)
	PWMB_LS	PTC3 / FTM0_CH3	PWM signal for phase B low-side driver
	PWMC_HS_B	PTE8 / FTM0_CH6	PWM signal for phase C high-side driver (inverted)
	PWMC_LS	PTE9 / FTM0_CH7	PWM signal for phase C low-side driver
PORT/GPI	HALL_A	PTC7	Hall sensor A signal
	HALL_B	PTC6	Hall sensor B signal
	HALL_C	PTA13	Hall sensor C signal
ADC0	BEMF_A	PTC1 / ADC0_SE9	BEMF voltage measurement of Phase A
	BEMF_B	PTC16 / ADC0_SE14	BEMF voltage measurement of Phase B
	BEMF_C	PTC15 / ADC0_SE13	BEMF voltage measurement of Phase C
	DCBI	PTA7 / ADC0_SE3	DC bus current measurement
	DCBV	PTC14 / ADC0_SE12	DC bus voltage measurement
LPSPI0	SPI_SCLK	PTB2 / LPSPI0_SCK	SPI clock (2 MHz)
	SPI_MISO	PTB3 / LPSPI0_SIN	SPI input data from GD3000
	SPI_MOSI	PTB4 / LPSPI0_SOUT	SPI output data for GD3000
LPUART0	SDA_SPI0_SOUT	PTB0 / LPUART0_RX	UART transmit data (FreeMASTER)
	SDA_SPI0_SIN	PTB1 / LPUART0_TX	UART receive data (FreeMASTER)
TRGMUX	PTA1	PTA1 / TRGMUX_OUT0	PDB0 channel 0 trigger output
	PTD0	PTD0 / TRGMUX_OUT1	ADC0 conversion complete flag
	PTA0	PTA0 / TRGMUX_OUT3	FTM0 initialization trigger
GPIO	GD_EN	PTA3 / PTA3	Enable signal for GD3000
	GD_RST_B	PTA2 / PTA2	Reset signal for GD3000
	SPI_CS_B	PTB5 / PTB5	Chip select signal for GD3000
	BTN0	PTD3 / PTD3	Application control via board button
	BTN1	PTD5 / PTD5	Application control via board button
	PTD1	PTD1 / PTD1	GPIO toggling to measure execution time
	BRAKE_PWM	PTD2 / PTD2	Connecting / disconnecting braking resistor
	GD_INT	PTA11 / PTA11	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 FTM0 module is shown in *Figure 27* as an example.


ADC | CAN | CMP | FLEXIO | FTM | GPIO | JTAG | LPI2C | LPSPI | LPTMR | LPUART | Platform | PowerAndGround | RTC | SWD | TRGMUX

Signals	Pin/Signal Selection	Direction
FTM0		
FTM Channel 0	PTD15	Output
FTM Channel 1	PTD16	Output
FTM Channel 2	PTC2	Output
FTM Channel 3	PTC3	Output
FTM Channel 4	*No pin routed*	*No pin routed*
FTM Channel 5	*No pin routed*	*No pin routed*
FTM Channel 6	PTE8	Output
FTM Channel 7	PTE9	Output
FTM Fault Input 2	*No pin routed*	Input
FTM Common External Clock Input 0	*No pin routed*	Input
FTM Common External Clock Input 1	*No pin routed*	Input
FTM Common External Clock Input 2	*No pin routed*	Input

Figure 27. **S32K116 Pins assignment for FTM0 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 12*. It defines that PTA11 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 S32K116 are configured calling *PINS_DRV_Init* function at the top of the *Example 12*.

Example 12. S32K116 pins configuration controlled by S32 SDK

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

    /* Enable interrupt when rising edge is detected on PTA11 to detect MC34GD3000 faults */
    PINS_DRV_SetPinIntSel(PORTA, 11u, PORT_INT_RISING_EDGE);

#if HALL_SENSOR
    /* Enable interrupt when rising or falling edge is detected on PTA13, PTC6 and PTC7
             * to detect Hall Sensor */
    PINS_DRV_SetPinIntSel(PORTA, 13u, PORT_INT_EITHER_EDGE);
    PINS_DRV_SetPinIntSel(PORTC, 7u,  PORT_INT_EITHER_EDGE);
    PINS_DRV_SetPinIntSel(PORTC, 6u,  PORT_INT_EITHER_EDGE);
#endif
}

pin_settings_config_t g_pin_mux_InitConfigArr[NUM_OF_CONFIGURED_PINS] =
{
...

    {
        .base          = PORTA,
        .pinPortIdx    = 11u,
        .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

Figure 28 presents the conceptual system block diagram of the BLDC Six-step control technique working either in sensorless or Hall sensor-based mode. This section is focused on the software design of the Sensorless algorithm based on the zero-crossing detection technique.

The application is optimized for S32K116 motor control peripherals to achieve the least possible core involvement in state variable acquisition and output action application. The motor control peripherals (FTM1, FTM0, PDB0, ADC0) are internally linked together to work independently from the core, and to achieve deterministic sampling of analog quantities and precise commutation of the stator field. The software part of the application consists of different blocks which are described below. The entire application behavior is controlled from a PC through the FreeMASTER run-time debugging tool.

The system block diagram is shown in *Figure 28*. The motor control algorithm blocks utilize the Automotive Math and Motor Control Library for ARM® Cortex®-M0+ (see section *References*).

The inputs of the control loop are the measured voltages and current on the power stage, in particular the phase voltages, the DC bus current, and DC bus voltage. The DC bus current is amplified by the current sense amplifier, which is part of the MC34GD3000 FET pre-driver, and then routed together with the DC bus voltage and phase voltages to the ADC for measurement acquisition.

From a control perspective, the block diagram can be divided into two logical parts:

- *Commutation control*, where the phase voltages and DC bus voltage are used to calculate the actual position of the shaft. According to the identified position, the next commutation event can be prepared.
- *Speed/torque control*, where the required shaft velocity is compared to the actual measured speed and regulated by the PI controller. The output of the speed PI controller is the duty cycle. The duty cycle is limited by the current PI controller and assigned to the PWM.

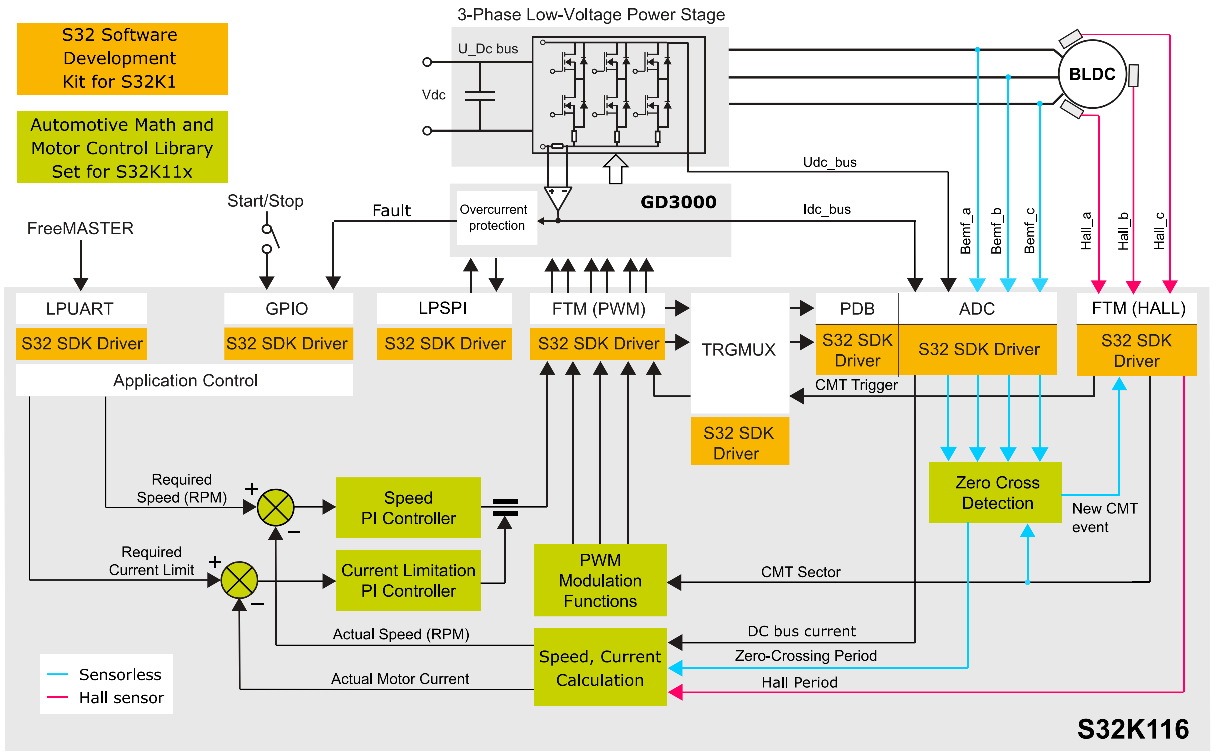


Figure 28. **System block diagram**

4.3.1. Introduction

This section describes the software design of the Sensorless BLDC Six Step 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 flow in Sensorless mode

Figure 29 explains the different application states. The figure consists of two interconnected parts:

- The speed over time characteristic
- The blocks in the lower part of the picture, which show the states of the application and the transitions between respective states

The application software has three main states: the alignment state, the open-loop start state, and the run state. In the run state, the BLDC motor is fully controlled in a closed-loop sensorless mode. After the initialization of the peripheral modules has completed, the software enters the alignment state. In alignment state, the rotor position is stabilized into a known position in order to create the same start-up torque in both directions of rotation. This is achieved by applying a PWM signal to phase C. Phases A and B are assigned with a duty cycle equal to zero; that is, they are connected to the negative pole of the DC bus. The value of the duty cycle on phase C depends on the motor inertia and load applied on the shaft. Such a technique aligns the shaft into position between phase A and B, which is perpendicular to both start-up flux vectors (vectors 0 and 3) generated by the stator winding, and therefore ensures the same start-up torque in both directions of rotation. The duration of the alignment state depends on the motor’s electrical and mechanical constants, the applied current (meaning duty cycle), and the mechanical load.

When the alignment time-out expires, the application software moves to the open-loop start state. At a very low shaft velocity, the BEMF voltage is too low to reliably detect the zero-crossing. Therefore, the motor has to be controlled in an open-loop mode for a certain time period. The very first vector generated by the stator windings needs to be set to a position 90° relative to the position of the flux vector generated by magnets mounted on the rotor. The alignment and first start-up vector are shown in *Figure 29*. The duration of the open-loop start state is defined by the number of open-loop commutations. The number of open-loop commutations depends on the mechanical time constant of the motor, including load, and also on the applied voltage (duty cycle). The shaft velocity after an open-loop start-up is approximately 5% of nominal velocity. At a velocity approximately 5% of nominal velocity, the BEMF voltage is high enough to reliably detect the zero-crossing.

After a defined number of commutation cycles, the state changes from the open-loop start state to the run state. From here on, the commutation process based on the BEMF zero-crossing measurement takes place, and the control enters the closed-loop mode.

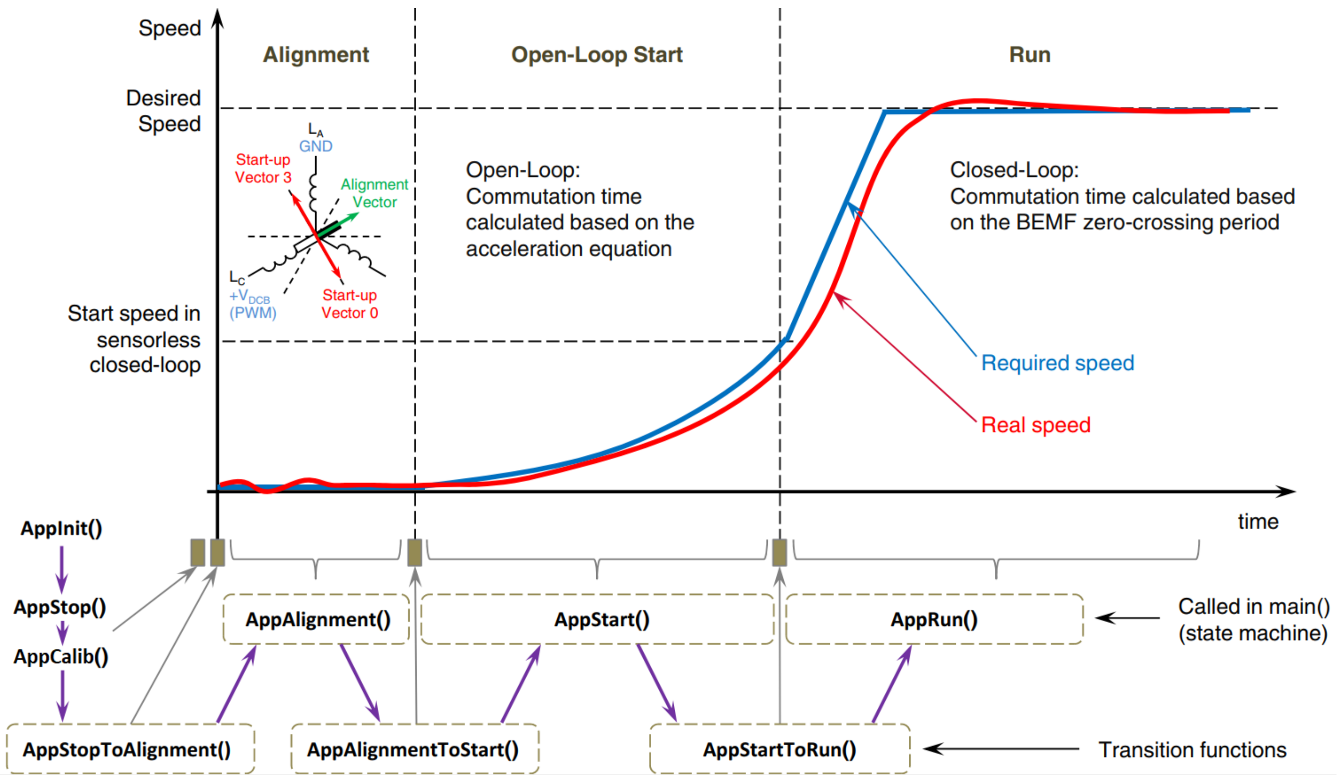


Figure 29. **Flow chart diagram of main function with background loop.**

4.3.3. State machine

The application state machine is implemented using a one-dimensional array of pointers to state functions, called AppStateMachine[]. The index of the array specifies the pointer to the related application state function. The application state machine consists of the following seven states selected using the index variable appState value. The application states are listed in the *Table 7*. Possible state transitions are shown in *Figure 30*.

Table 7 **Application states in Sensorless mode**

AppState	**Application state**	**Description**
0	INIT	The INIT state provides the initial configuration of the PWM duty cycle and DC bus current offset calibration. The state machine then transitions to the STOP state.
1	CALIB	The CALIB state provides the DC bus current calibration. The state machine then transitions to ALIGNMENT state.
2	ALIGNMENT	In the ALIGNMENT state, the alignment vector is applied to the stator to set the rotor to the defined position. The duration of the alignment state and the duty cycle applied during the state are defined by the ALIGN_DURATION and ALIGN_VOLTAGE macro values accessible in the BLDC appconfig.h header file. The state machine then transitions to the START state.
3	START	In the START state, the motor commutation is controlled in an open-loop without any rotor position feedback. The initial commutation period is controlled by the STARTUP_CMT_PER macro value. Motor acceleration (commutation period multiplier <1) is set by the START_CMT_ACCELER macro value. The number of commutations in the START state is defined by STARTUP_CMT_CNT macro value. All macro values are accessible in the BLDC_appconfig.h header file. The aim of the START state is to achieve an RPM where the zero-crossing event can be reliably detected (BEMF high enough). Once the defined number of commutations is performed, the state machine transitions to the RUN state.
4	RUN	In the RUN state, the BLDC motor is controlled in the closed-loop by the sensorless algorithm (BEMF zero-crossing detection). Speed control and current limitation are performed as described in *4.3.6, "Speed evaluation, motor current limitation and control"*. The transition to the INIT state is done by setting the appSwitchState variable to 0.
5	STOP	In the STOP state, the motor is stopped and prepared to start running. Transition to the ALIGNMENT state is performed once the appSwitchState variable is set to 1 and the freewheeling counter expires.
6	FAULT	The fault detection function is executed in the main endless loop, detecting DC bus undervoltage, DC bus overvoltage, DC bus overcurrent, and GD3000 faults. Once any of the faults are detected, the state machine automatically transitions to the FAULT state. The PWM outputs are set to the safe state. To exit the FAULT state, all fault sources must be removed and the faultSwitchClear variable has to be set to 1 to clear the fault latch. The state machine then automatically transitions to the INIT state.

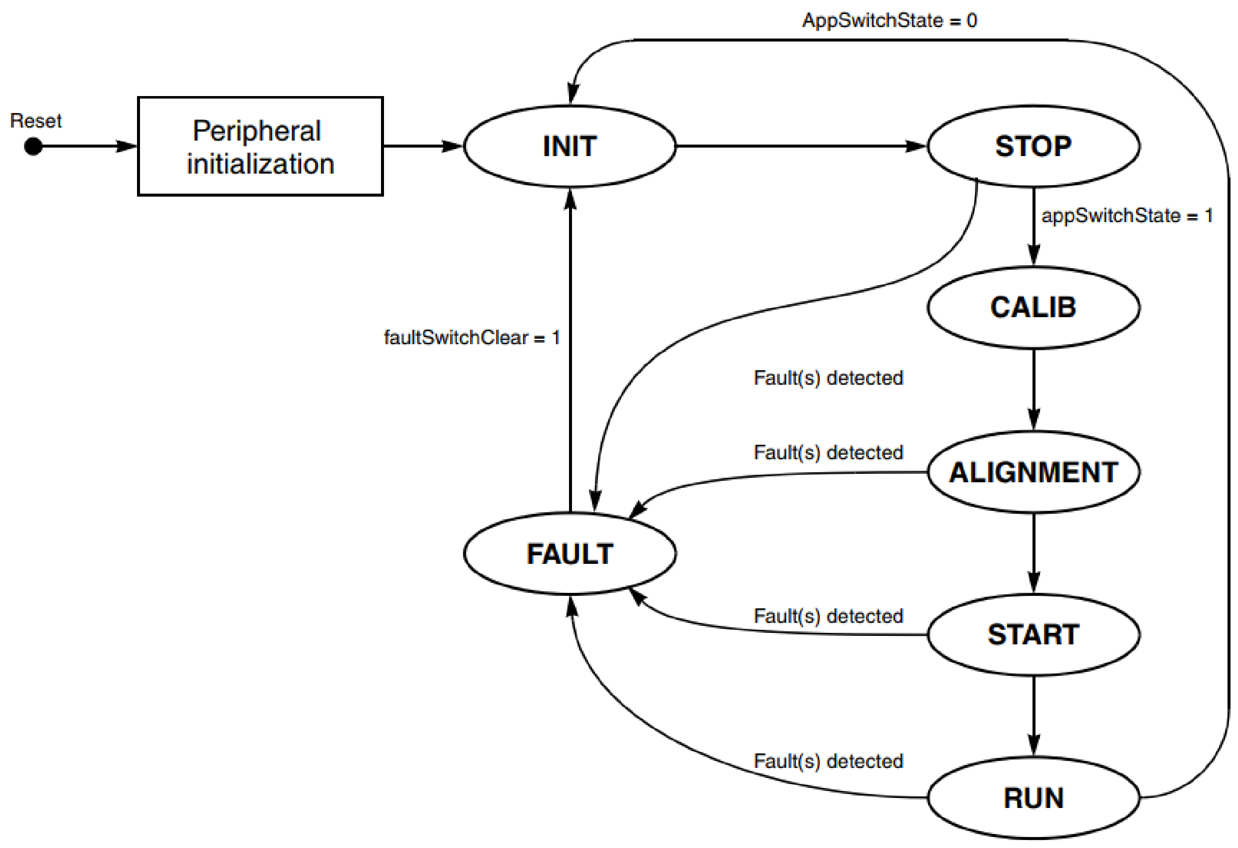


Figure 30. **Application state machine**

4.3.4. Application timing and interrupts

Figure 31 shows the application timing and the associated interrupts used for the commutation, zero-crossing and speed control. The grey boxes show the executed interrupt routines versus the phase voltage measurement.

The top row shows the interrupt that is activated when the ADC conversion sequence of BEMF voltage, DC bus current, and DC bus voltage has been completed. In this interrupt, the FTM1 timer counter value is saved as a BEMF measurement reference point. The zero-crossing detection algorithm is executed in each ADC0 conversion complete interrupt after a commutation event. Once the zero-crossing is found, the, detection algorithm is disabled until the new commutation event occurs.

The second row shows the FTM1 timer counter overflow interrupt generated at the time of the commutation event. The time between each FTM1 timer counter overflow interrupt is dependent on the actual speed of the motor. Channel of the ADC0 is reconfigured to reflect the change in the phase used for the BEMF voltage sensing.

The last row shows the LPIT channel 0 time-out interrupt generated every 1 ms. This interrupt is used for speed loop control and motor current limitation, executing PI controller functions.

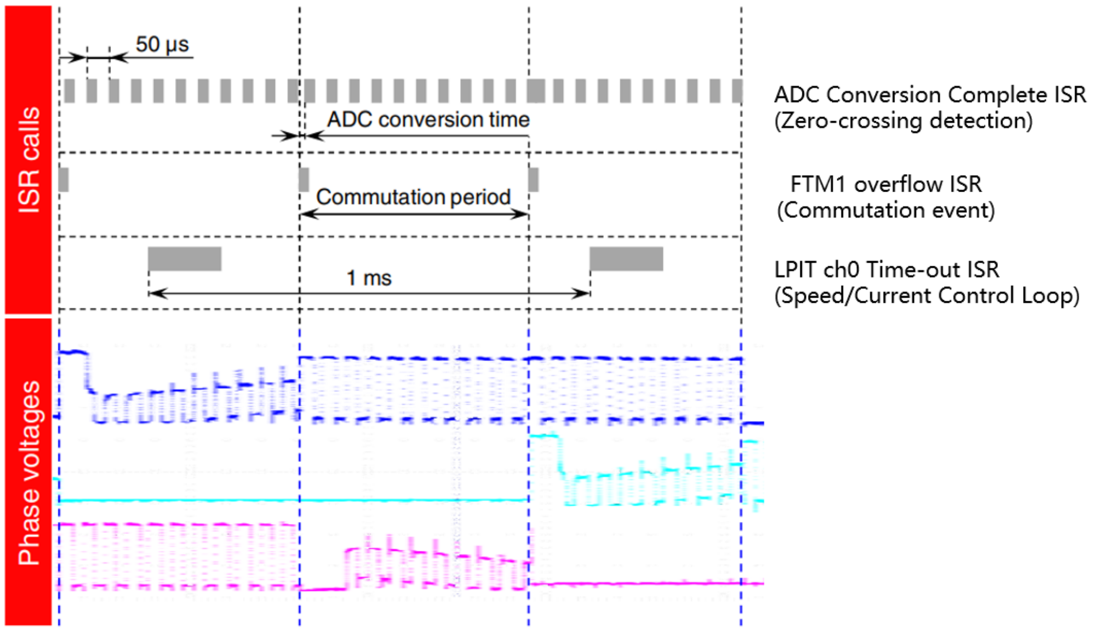


Figure 31. **Application timing and interrupts**

4.3.5. Zero-crossing detection processing

For state variable acquisition and zero-crossing detection processing, the ADC0 conversion sequence complete interrupt is used. The interrupt service routine is executed once the conversion sequence consisting of BEMF voltage, DC bus current, and DC bus current conversion is finished. The ADC0 conversion sequence complete interrupt service routine is shown in *Example 13*.

Before the ADC0 conversion complete ISR is executed, the ADC0 store the results in the ADC0 results registers; BEMF voltage into ADC0_R2, DC bus current into ADC0_R0, and DC bus voltage into ADC0_R1. These measurements are saved then into the result structure.
The value of the current sense amplifier bias voltage offset is subtracted from the measured DC bus current value to obtain the bidirectional DC bus current.

A filtering of the DC bus voltage and DC bus current is provided using the moving average filter functions. The BEMF voltage is then calculated as the difference between the phase voltage and the half of the DC bus voltage. The BEMF voltage value is a signed number.

The software checks whether the current decay period has already passed (see *3.3.3.1*) to initiate the zero-crossing detection. The current decay period is called T_{OFF} (variable *timeZCToff*) in the application implementation, *Example 13*.

Example 13. Processing measurements in the ADC0 conversion complete ISR

```
void ADC0_IRQHandler()
{
#if (!HALL_SENSOR)
    timeOldBackEmf = timeBackEmf;
        timeBackEmf = FTM_DRV_CounterRead(INST_FLEXTIMER_MC0);
#endif
```

```
    /* DC BUS measurement of the disconnected phase */
    ADCResults.DCBVVoltage = (tFrac16)(ADC0->R[1u] << 3);
    /* DC Bus current  measurement */
    ADCResults.DCBIVoltageRaw = (tFrac16)(ADC0->R[0u]);
    /* Bemf Voltage measurement of the disconnected phase */
    ADCResults.BEMFVoltage = (tFrac16)(ADC0->R[2u] << 3);

    /* Hall counter measurement */
    SensorHall.Ftm1HallCnt = FTM1->CNT;

    /* Real DC Bus current = Raw value - DC bus current offset */
    ADCResults.DCBIVoltage = ADCResults.DCBIVoltageRaw - ADCResults.DCBIOffset;

    u_dc_bus_filt = (tFrac16)(GDFLIB_FilterMA_F16(ADCResults.DCBVVoltage, &Udcb_filt));

    /* bemfVoltage = Voltage of the disconnected phase - DC Bus voltage/2 */
    bemfVoltage = ADCResults.BEMFVoltage - (u_dc_bus_filt >> 1);

    if(duty_cycle > DC_THRESHOLD)
    {
        torque_filt = (tFrac16)(GDFLIB_FilterMA_F16(ADCResults.DCBIVoltage, &Idcb_filt));
    }
    else
    {
        /* Ignore DC bus current measurement at low duty cycles */
        torque_filt = (tFrac16)(GDFLIB_FilterMA_F16((tFrac16)0, &Idcb_filt));
    }

/* ZC detection algorithm is ignored in Sensorbased mode */
#if (!HALL_SENSOR)
    if(driveStatus.B.AfterCMT == 1)
    {
        if(timeBackEmf > timeZCToff)
        {
            driveStatus.B.AfterCMT = 0;
        }
    }
....
```

Where the commutation transient time T_{OFF} has not yet expired (*driveStatus.B.AfterCMT = 1*), the zero-crossing calculation will not be performed. The calculation will also not be performed if the zero-crossing point has already been identified in the current commutation period (*driveStatus.B.NewZC= 1*), or if the application is running in open-loop mode (*driveStatus.B.Sensorless = 0*).

If the above mentioned conditions are not met, the zero-crossing detection routine will be executed. Based on the current commutation sector, the BEMF slope direction is checked. If the BEMF slope is negative, the sign of the calculated value is changed. This operation allows usage of a single BEMF zero-crossing detection function for a positive slope BEMF in all commutation sectors.

When the zero-crossing position calculation is finished, the BEMF voltage value is stored as the old value as it will be referenced again in the next PWM cycle.

Code listing in *Example 15* describes the zero-crossing detection routine that was called in the interrupt shown before.

Example 14. S32K116 BEMF Zero-crossing detection control

```
....
        if((driveStatus.B.AfterCMT == 0) && (driveStatus.B.NewZC == 0) && (driveStatus.B.Sensorless == 1))
        {
            /* If the BEMF voltage is falling, invert BEMF voltage value */
            if((ActualCmtSector & 0x01) == 0)
            {
                bemfVoltage = -bemfVoltage;
            }

            /* Rising BEMF zero-crossing detection */
            if(bemfVoltage >= 0)
            {
                ....
            }

            bemfVoltageOld = bemfVoltage; /* Save actual BEMF voltage (for ADC samples interpolation) */

            driveStatus.B.AdcSaved = 1;
        }

        /* S32K116 only one ADC. If set BEMF phase in FTM1, ADC sample will be disorganized and will trigger PDB
        error */
        if(TRUE == CommutationFlag)
        {
            CommutationFlag = FALSE;
            /* Measure back-EMF voltage of the disconnected stator phase */
            MEAS_SetBEMFPhase(bemfPhaseList[rotationDir][ActualCmtSector]);
        }
#endif

        /* Timer for Rotor alignment */
        if(driveStatus.B.Alignment)
        {
            if(alignmentTimer > 0)
            {
                alignmentTimer--;
            }
            driveStatus.B.AdcSaved = 0;
        }

        // Application variables record
        FMSTR_Recorder();

}
```

In the case of a negative BEMF voltage ($V_{BEMF} < V_{DCB} / 2$), the zero-crossing point has not been passed and the zero-crossing is not detectable. The software exits the zero-crossing detecting routine and leaves the zero-crossing status bit unchanged (*driveStatus.B.NewZC* = *0*). In the case of a zero or a positive BEMF voltage ($V_{BEMF} \geq V_{DCB} / 2$), the zero-crossing point was reached or passed and *Equation 7* is calculated, meaning that the BEMF voltage is divided by the delta of the two measured points and multiplied by the measured PWM period (BEMF measurement period). After this calculation, the old zero-crossing time and the new one are saved into the appropriate variables. The zero-crossing period is then calculated based on the calculated time of zero-crossing and the time of the zero-crossing in the previous commutation cycle. The zero-crossing period is also filtered to improve reliability

At the end of the routine, the new commutation time is calculated. Here, some motor characteristics have to be taken into account. Instead of just adding half of a zero-crossing period to the actual zero-crossing time, a so-called advance angle factor is taken into account, which actually activates the commutation a bit earlier than calculated. This is usually a constant and depends on the motor characteristics.

Finally, the zero-crossing status bit is set (*driveStatus.B.NewZC* = *1*), so the zero-crossing detection does not take place anymore in the current commutation cycle.

Example 15. S32K116 BEMF Zero-crossing detection algorithm

```
/* Rising BEMF zero-crossing detection */
if(bemfVoltage >= 0)
{
     /* Rising interpolation */
    delta = bemfVoltage - bemfVoltageOld;

    /* calculating the time of BEMF zero-crossing */
    if((driveStatus.B.AdcSaved == 1) && (delta > bemfVoltage))
    {
        timeBackEmf -= MLIB_Mul(MLIB_Div(bemfVoltage, delta), (timeBackEmf - timeOldBackEmf));
    }
    /* calculating the time just for open loop control */
    else
    {
        timeBackEmf -= ((timeBackEmf - timeOldBackEmf) >> 1);
    }

    lastTimeZC = timeZC;
    timeZC = timeBackEmf;

    /* periodZC = (timeZC - lasTimeZC) + ftm_mod_old(no timer reset) */
    periodZC[ActualCmtSector] = (ftm_mod_old - lastTimeZC) + timeZC;
    /* Average of the previous and current ZC period */
    actualPeriodZC = (actualPeriodZC + periodZC[ActualCmtSector]) >> 1;
    /* advancedAngle(0.3815) = 0.5 * Advanced Angle(0.763) */
    NextCmtPeriod = MLIB_Mul(actualPeriodZC, advanceAngle);

    /* Update commutation period -> FTM1_MOD = timeZC + nextCmtPeriod */
    FTM1_UPDATE_MOD(timeZC + NextCmtPeriod);

    driveStatus.B.NewZC = 1;
}
```

4.3.6. Speed evaluation, motor current limitation and control

The speed controller in *Example 16* is executed in a timer interrupt every 1 ms. First of all, the actual speed is calculated from all of the last six zero-crossing periods, and this is stored as the actual speed. The required speed is fed into the ramp function controlling the motor speed slope. The difference between the speed ramp function output and actual speed defines the speed error.

In the closed-loop mode, the actual speed error is fed into the PI controller function. Inputs to the PI controller function include the speed error and the PI controller's parameters such as the proportional and integral gain constants. The output of the PI controller is the duty cycle, which is scaled to the PWM resolution.

At the end of the speed control function, the duty cycle is loaded into the FTM0 PWM module.

Example 16. S32K116 Speed evaluation software flow

```
void LPIT0_Ch0_IRQHandler()
{
        uint8_t i;
        PTD->PSOR |= 1<<2;

        if(driveStatus.B.CloseLoop == 1)
        {
                torqueErr = MLIB_SubSat(I_DCB_LIMIT, torque_filt);
                currentPIOut = GFLIB_ControllerPIpAW_F16(torqueErr, &currentPIPrms);
                /* Speed control */
        #if HALL_SENSOR
                period6ZC = SensorHall.Period[0];
                for(i=1; i<6; i++)
                {
                        period6ZC += SensorHall.Period[i];
                }
        #else
                period6ZC = periodZC[0];
                for(i=1; i<6; i++)
                {
                        period6ZC += periodZC[i];
                }
        #endif

                /* Actual rotor speed is calculated based on ZC period or period measured by FTM1 Modulus Counter
                mode */
                actualSpeed = (uint16_t)(((uint32_t)(SPEED_SCALE_CONST << 15)) / period6ZC);

                /* Upper speed limit due to the limited DC bus voltage 12V */
                if(requiredSpeed >= N_NOM)
                requiredSpeed = N_NOM;

                /* Lower speed limit keeping reliable sensor less operation */
                if(requiredSpeed < mcat_NMin)
                requiredSpeed = mcat_NMin;

                requiredSpeedRamp = MLIB_ConvertPU_F16F32(GFLIB_Ramp_F32(MLIB_ConvertPU_F32F16(requiredSpeed),
                &speedRampPrms));
                speedErr = requiredSpeedRamp - actualSpeed;
                speedPIOut = GFLIB_ControllerPIpAW(speedErr, &speedPIPrms);
        ....
```

The current limit controller is located in the same 1 ms timer interrupt (*LPIT0_Ch0_IRQHandler()*) as the speed controller because the inputs and outputs of both controllers are linked together.

When the actual speed has been calculated, the current limit PI controller can be called by feeding it with the difference between the actual current and the maximum allowed current of the motor. The output of the PI controller is scaled to the number proportional to the PWM period. After the current PI controller has calculated its duty cycle, both duty cycle output values are compared to each other.

If the speed PI controller duty cycle output is higher than the current limit PI controller output, then the speed PI Controller duty cycle output value is limited to the output value of the current limit PI controller. Otherwise, the speed PI duty cycle output will be taken as the duty cycle update value. The value of the duty cycle will be used to update the FTM0 PWM module. At the end, the integral portion values of both the PI controllers need to be synchronized to avoid one of the controllers increasing its internal value as far as the upper limit. If the duty cycle was limited to the current PI duty cycle output, then the integral portion of the current PI controller will be copied into the integral portion of the speed controller, and vice versa. The above described procedure is also described in *Example 17*.

At the end of *LPIT0_Ch0_IRQHandler()* PDB0 pre-trigger delays are calculated based on the actual duty cycle to measure DC Bus current and BEMF voltage towards the half and end of the active PWM pulse as discussed in chapter *4.2.4*. Double-buffered registers PDB_CHnDLYx are updated when PDB_DRV_LoadValuesCmd is called and FTM0 *init_trig* is detected on PDB0 trigger input.

Example 17. S32K116 Speed evaluation and current limitation

```
....
        if(currentPIOut >= speedPIOut)
        {
            /* If max torque not achieved, use speed PI output */
            currentPIPrms.f32IntegPartK_1 = MLIB_ConvertPU_F32F16(speedPIOut);
            currentPIPrms.f16InK_1 = 0;
            /* PWM duty cycle update <- speed PI */
            duty_cycle = MLIB_Mul(speedPIOut, PWM_MODULO);

            driveStatus.B.CurrentLimiting = 0;
        }
        else
        {
            /* Limit speed PI output by current PI if max. torque achieved */
            speedPIPrms.f32IntegPartK_1 = MLIB_ConvertPU_F32F16(currentPIOut);
            speedPIPrms.f16InK_1 = 0;
            /* PWM duty cycle update <- current PI */
            duty_cycle = MLIB_Mul(currentPIOut, PWM_MODULO);

            driveStatus.B.CurrentLimiting = 1;
        }

        /* Update PWM duty cycle */
        ACTUATE_SetDutycycle(duty_cycle, HW_INPUT_TRIG0);
    }
    else
    {
        actualSpeed = 0u;
    }

/* Free wheeling is ignored in Sensor based mode */
#if (!HALL_SENSOR)

    if(driveStatus.B.Freewheeling)
    {
        if(freewheelTimer > 0)
        {
            freewheelTimer--;
        }
        else
        {
            driveStatus.B.Freewheeling = 0;
        }
    }
#endif

    /* pdb_delay calculated based on the actual duty_cycle
     * to measure DC bus current, DC bus voltage and Back EMF voltage
     * towards the end of the active PWM pulse
     */
    pdb_delay1 = (uint16_t)((duty_cycle << 2) / 10);

    /* Saturate, if pdb_delay is lower than PDB_DELAY_MIN */
    if(pdb_delay1 < PDB_DELAY_MIN)
    {
        pdb_delay1 = PDB_DELAY_MIN;
    }

    /* Update PDBs delays */
    PDB_DRV_SetAdcPreTriggerDelayValue(INST_PDB0, 0, 1, pdb_delay1);

    PDB_DRV_LoadValuesCmd(INST_PDB0);

    CheckSwitchState();

    LPIT_DRV_ClearInterruptFlagTimerChannels(INST_LPIT1, 0b1);
}
```

4.3.7. Automotive Math and Motor Control Library

The application source code uses the NXP Automotive Math and Motor Control Library Set for ARM® Cortex®-M0+ (available at www.nxp.com) which consists of the following sub-libraries:

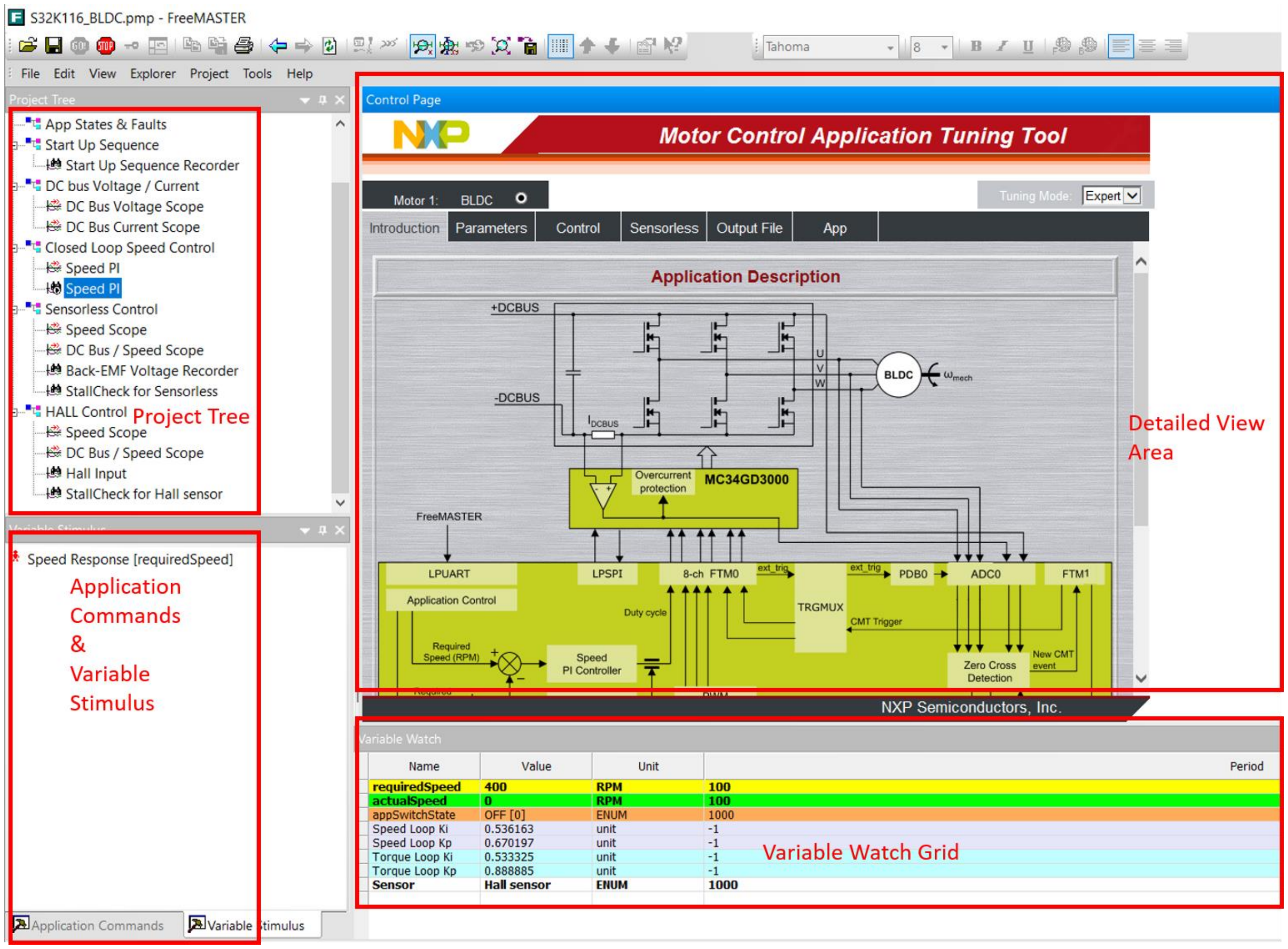
- Mathematical Library (MLIB) – includes basic mathematical functions such as addition, multiplication, etc.
- General Function Library (GFLIB) – includes basic trigonometric and general mathematical functions such as sine, cosine, ramp, PI controller, etc.
- General Digital Filters Library (GDFLIB) – includes digital FIR and IIR filters
- General Motor Control Library (GMLIB) – includes standard algorithms used for motor control such as Clarke/Park transformations, Space Vector Modulation, etc.
- Advanced Motor Control Function Library (AMCLIB) – comprising advanced algorithms used for motor control purposes.

5. Application Control

5.1. FreeMASTER graphical user interface

The FreeMASTER run-time debugging tool is used to control the application and monitor application variables during run-time. The FreeMASTER window with an opened application project comprises several panes:

- **Project Tree** – Provides a logical project tree structure containing the main page, several oscilloscopes and BEMF voltage recorder.
- **Variable Stimulus** – Allows you to enable automatic motor speed stimulus for motor speed response observation.
- **Variable Watch Grid** – Contains the list of watched variables and provides a simple interface to start/stop the motor and to set the rotation speed of the motor.
- **Detailed View Area** – Displays the Motor Control Application Tuning (MCAT) tool GUI by default. Contents of the detailed view area change based on the selected item in the project tree.



Name	Value	Unit	Period
requiredSpeed	**400**	**RPM**	**100**
actualSpeed	**0**	**RPM**	**100**
appSwitchState	OFF [0]	ENUM	1000
Speed Loop Ki	0.536163	unit	-1
Speed Loop Kp	0.670197	unit	-1
Torque Loop Ki	0.533325	unit	-1
Torque Loop Kp	0.888885	unit	-1
Sensor	**Hall sensor**	**ENUM**	**1000**

Figure 32. **FreeMASTER window with an application project opened**

5.1.1. Project tree

Allows selecting the content of the detailed view area, as follows:

- **S32K_BLDC_Sensorless** – displays the MCAT GUI
- **DC Bus Current Calib** – displays the variable recorder (DCBI Calib) which allows recording of DC Bus current offset as well as real value.
- **StallCheck Detection** – displays the variable recorder (StallCheck) triggered by variable *stallCheckCounter*.
- **App States & Faults** – displays application states and faults in Variable Watch Grid.
- **Start Up Sequence** – displays the variable recorder (Start Up Sequence) which allows recording of BEMF voltage, drive status, actual commutation sector and next commutation period.
- **Closed Loop Control** – displays scope and recorder of speed control variables

- **Sensorless Control** – displays scope of speed variables, DC Bus voltage variables, DC Bus current variable, mixed scope with speed variables and DC Bus variables and displays also recorder of BEMF and commutation variables.

5.2. Motor Control Application Tuning Tool

The MCAT is a graphical tool with a friendly environment and intuitive control. As shown in *Figure 33Figure 33* the tool consists of a motor selector bar, tab menu, and workspace. The MCAT tool represents a modular concept that consists of several sub-modules. Each sub-module represents one tab in the tab menu. The arrangement of the submodules is flexible according to the needs of the embedded application.

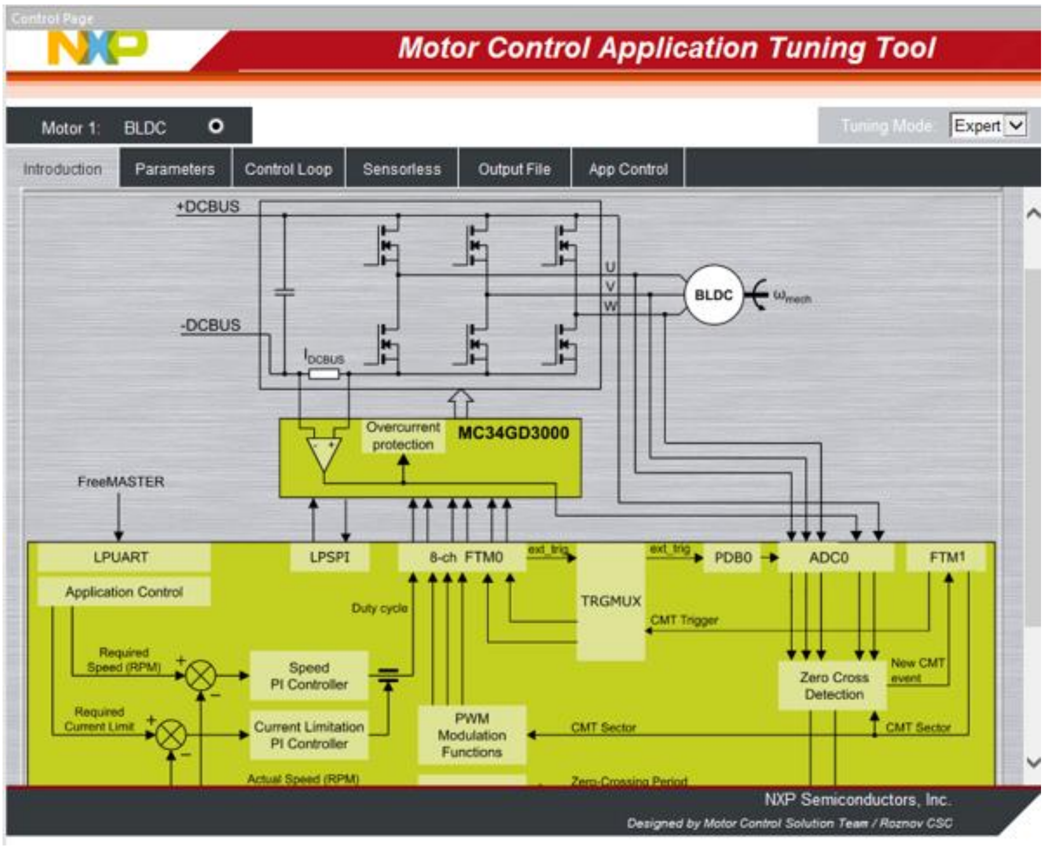


Figure 33. **MCAT – project page**

The MCAT tool is part of reference software for a dedicated MCU. Since the tuning tool cannot be used as a standalone, it is included in the FreeMASTER project by default.

The tool supports output header file generation with the calculated constants required for control algorithms, and also enables on-line updates of those application control variables selected for tuning, for example, the control loop, speed ramp, and so forth. The variables are updated by clicking the “Update Target” button on each control tab.

The set motor parameters can be stored in an internal MCAT file by clicking the “Store Data” button or the data can be reloaded by clicking the “Reload Data” button.

Each parameter and constant contains a short hint that can be activated on a parameter name mouse focus; see *Figure 34* for an example of this hint information.

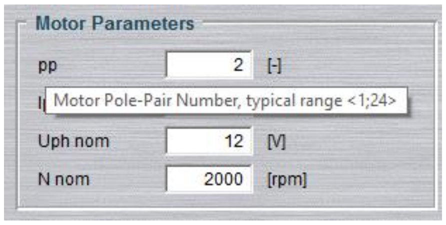


Figure 34. **Parameters hint information**

The MCAT tool workspace is unique for each tab and a detailed overview of each available tab is provided in the subsequent sections.

5.2.1. Introduction tab

The introduction tab can be considered as a voluntary tab. It provides a room for describing or introducing the targeted motor control application, as shown in *Figure 33*.

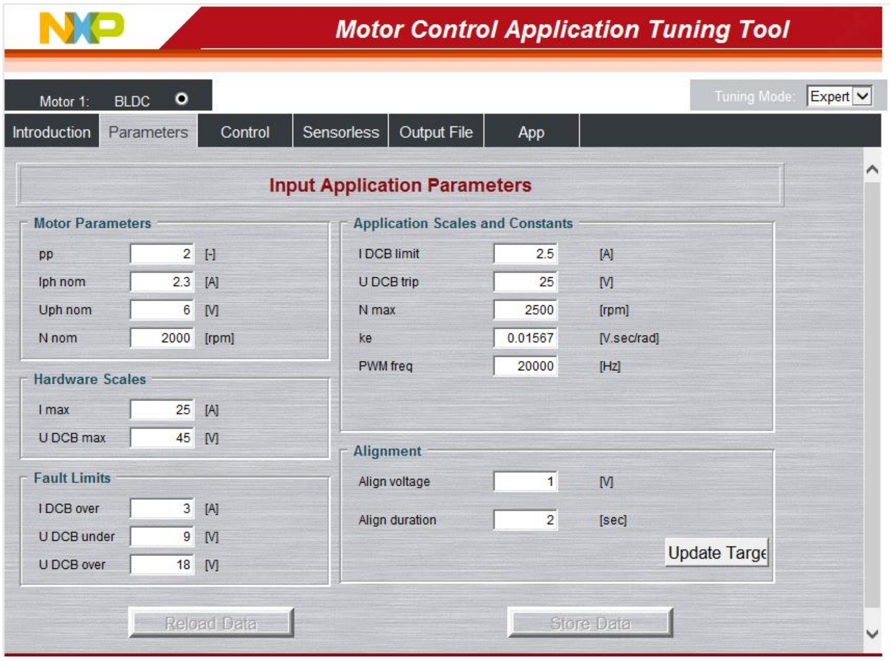
5.2.2. Parameters tab

The parameters tab is dedicated for entering the input application parameters, as shown in *Figure 35* a mandatory tab due to its high-level dependency with other tabs. Please take care while filling in an filled value in the cells can cause unexpected behavior in an application running on the target. The impact of each required input is described in *Table 8*. The number of input parameters needed to be filled in depends on the selected application tuning mode:

- Basic – highly recommended for users who are not experienced enough in motor control theory. The number of required input parameters is reduced. The mandatory cells are with a white background while the rest of the input parameters are calculated automatically by the MCAT tool engine. These parameters are read-only and shadowed.
- Expert – all input parameters are accessible and freely editable by the user. However, their settings require a certain level of expertise in motor control theory.

NOTE

When switching from the Expert to Basic mode, some parameters are overridden by the automatically calculated parameter values. Values previously set in the Expert mode are not retained and need to be reloaded by changing any editable parameter value and clicking the “Reload Data” button after switching back to Expert mode.



Motor 1: BLDC

Tuning Mode: Expert

Input Application Parameters

Motor Parameters

pp	2	[-]
Iph nom	2.3	[A]
Uph nom	6	[V]
N nom	2000	[rpm]

Hardware Scales

I max	25	[A]
U DCB max	45	[V]

Fault Limits

I DCB over	3	[A]
U DCB under	9	[V]
U DCB over	18	[V]

Application Scales and Constants

I DCB limit	2.5	[A]
U DCB trip	25	[V]
N max	2500	[rpm]
ke	0.01567	[V.sec/rad]
PWM freq	20000	[Hz]

Alignment

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

Figure 35. **Parameters tab – Expert mode**

Table 8 shows the list of the MCAT tool input parameters with their units, a brief description, typical range, and their accessibility status in basic mode.

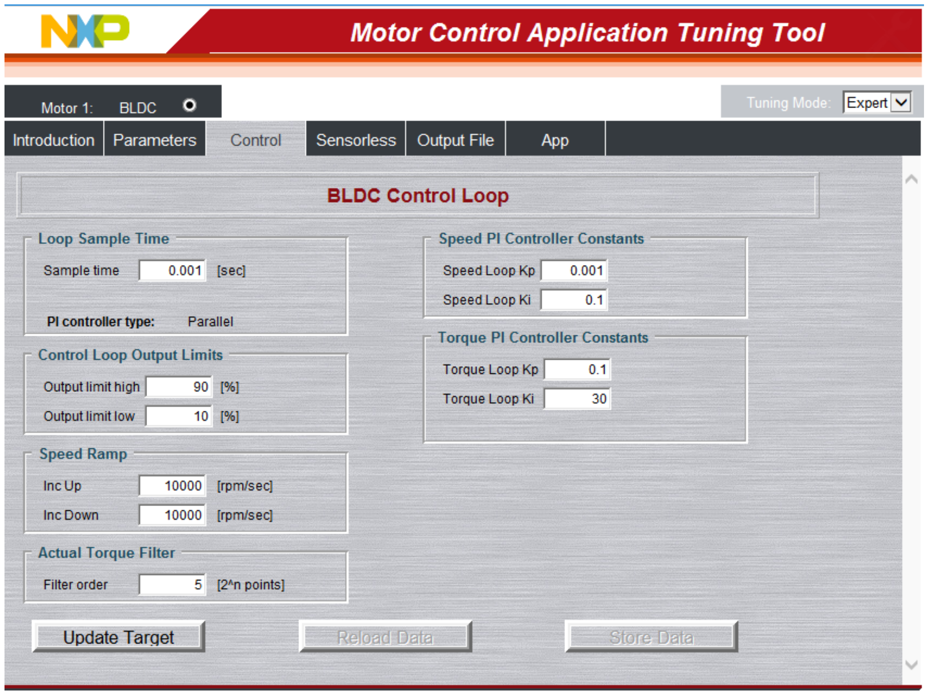
Table 8 **Parameters tab – parameter list**

Parameter name	**Unit**	**Description**	**Basic mode accesibility**
pp	[-]	Motor pole-pair number	Yes
Iph nom	[A]	Motor nominal phase current	Yes
Uph nom	[V]	Motor nominal phase voltage	Yes
N nom	[rpm]	Motor nominal mechanical speed	Yes
I max	[A]	HW board current scale	Yes
U DCB max	[V]	HW board DC bus voltage scale	Yes
I DCB over	[A]	DC bus overcurrent fault threshold current	No
U DCB under	[V]	DC bus undervoltage fault threshold voltage	No
U DCB over	[V]	DC bus overvoltage fault threshold voltage	No
I DCB limit	[A]	DC bus current limit of control loop	No
U DCB trip	[V]	Resistor braking DC bus voltage threshold	No
N max	[rpm]	Mechanical speed limit	No
ke	[V.sec/rad]	Back-EMF constant	No
PWM freq	[Hz]	Frequency of PWM output signal	No
Align voltage	[V]	Voltage for mechanical rotor alignment	No
Align duration	[sec]	Duration of motor alignment	No

The parameters of the controlled motor can be acquired from the motor data sheet provided by the motor manufacturer, or by laboratory measurement.

5.2.3. Control loop tab

The control loop tab is designed for speed and torque loop tuning. The torque and speed PI controllers run in parallel with a common output limitation. The tab contains input parameters for the torque and speed control loops that are used for the PI controller, the speed ramp, and speed filter constant calculations, as shown in *Figure 36*.



BLDC Control Loop

Loop Sample Time

Sample time 0.001 [sec]

PI controller type: Parallel

Control Loop Output Limits

Output limit high 90 [%]

Output limit low 10 [%]

Speed Ramp

Inc Up 10000 [rpm/sec]

Inc Down 10000 [rpm/sec]

Actual Torque Filter

Filter order 5 [2^n points]

Speed PI Controller Constants

Speed Loop Kp 0.001

Speed Loop Ki 0.1

Torque PI Controller Constants

Torque Loop Kp 0.1

Torque Loop Ki 30

Figure 36. **Control loop tab – Expert mode**

Table 9 shows the list of the speed/torque loop input parameters with their physical units, a brief description, typical range, and their accessibility status in basic mode.

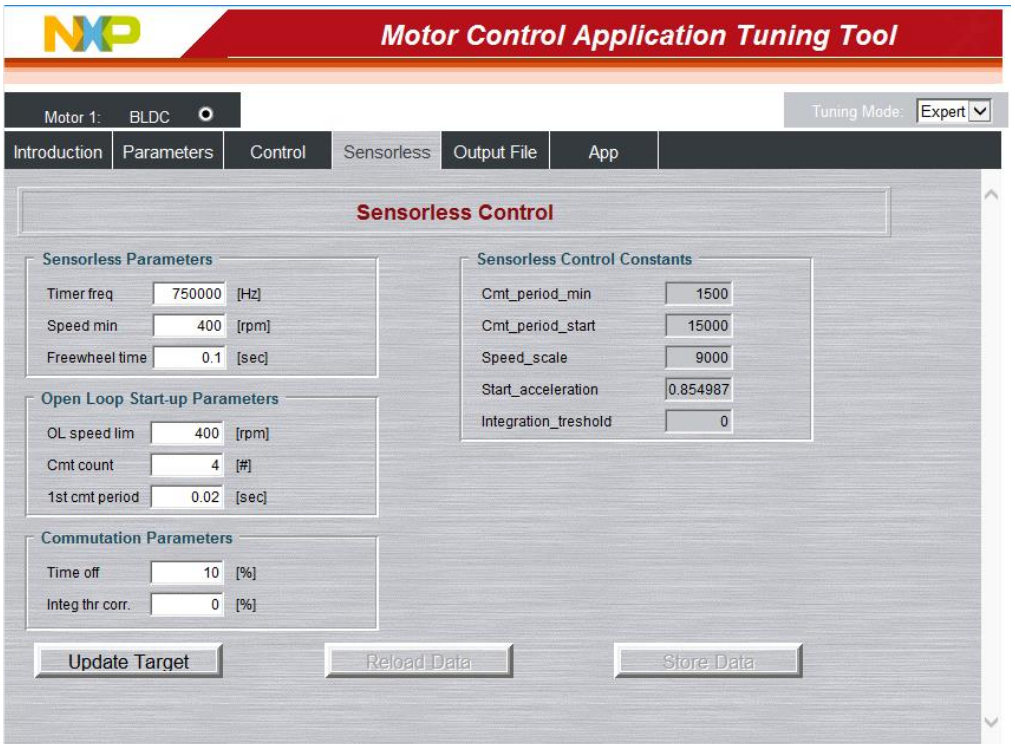
Table 9 **Control loop tab – parameter list**

Parameter name	Unit	Description	Bacis mode accesibility
Sample time	[sec]	Control loop period	No
Output limit high	[%]	Control loop output high limit	No
Output limit low	[%]	Control loop output low limit	No
Inc Up	[rpm/sec]	Speed ramp increment up	Yes
Inc Down	[rpm/sec]	Speed ramp increment down	Yes
MA Filter	[lambda]	Number of 2^n points of MA Torque filter	No
Speed Loop Kp	-	Proportional gain of speed PI controller in time domain	No
Speed Loop Ki	-	Integral gain of speed PI controller in time domain	No
Torque Loop Kp	-	Proportional gain of torque PI controller in time domain	No
Torque Loop Ki	-	Integral gain of torque PI controller in time domain	No

Clicking the “Update Target” button effects an update of the control loop and speed ramp dedicated variables in the target using the actual inputs from the tab.

5.2.4. Sensorless tab

The sensorless tab enables parameter settings for the BLDC sensorless control algorithm. The tab is divided into two parts, the left-side fields represent those input parameters required for sensorless algorithm constant calculation and the right-side represents the read-only calculated constants, as shown in *Figure 37*.



Sensorless Parameters

Timer freq	750000	[Hz]
Speed min	400	[rpm]
Freewheel time	0.1	[sec]

Open Loop Start-up Parameters

OL speed lim	400	[rpm]
Cmt count	4	[#]
1st cmt period	0.02	[sec]

Commutation Parameters

Time off	10	[%]
Integ thr corr.	0	[%]

Sensorless Control Constants

Cmt_period_min	1500
Cmt_period_start	15000
Speed_scale	9000
Start_acceleration	0.854987
Integration_treshold	0

Figure 37. **Sensorless tab – Expert mode**

*Table 10**Error! Reference source not found.*** shows the list of the speed loop input parameters with their physical units, a brief description,
typical range, and their accessibility status in basic mode.

Table 10 **Sensorless tab – parameter list**

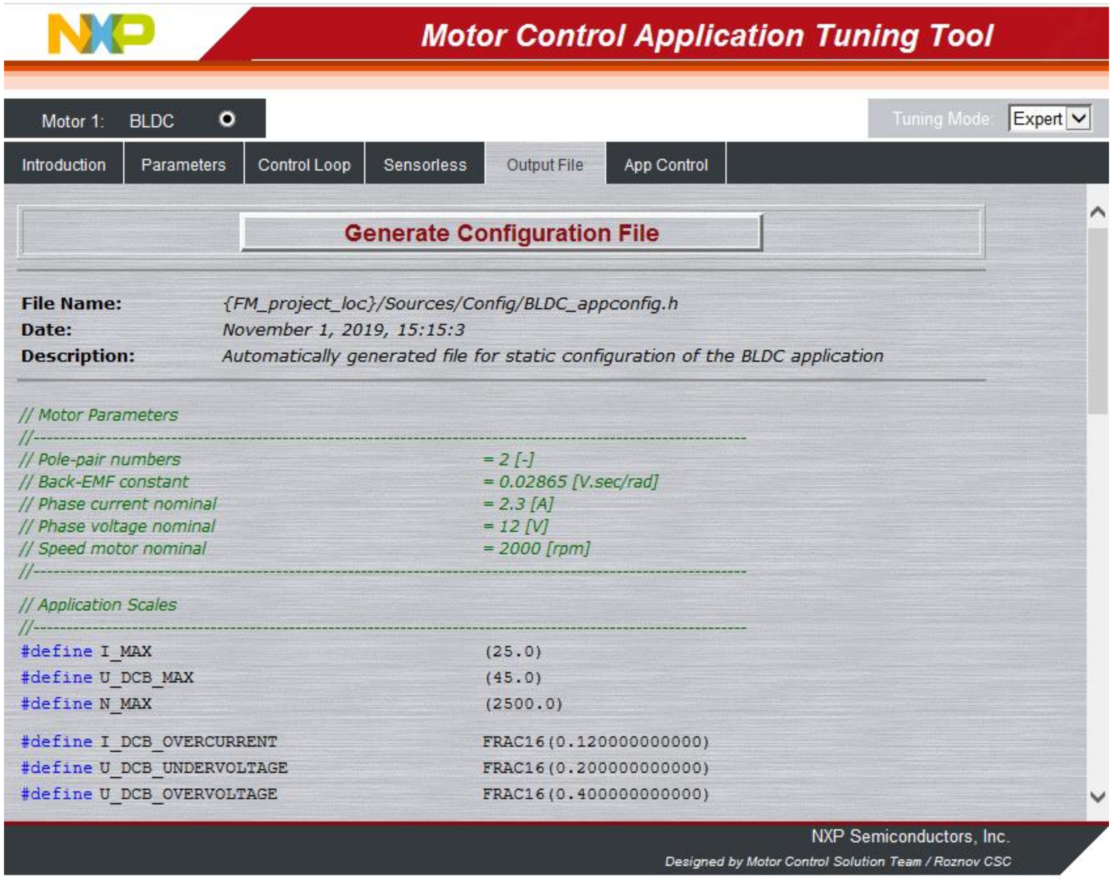
Parameter name	Unit	Description	Bacis mode accesibility
Timer freq	[Hz]	Frequency of timer used for commutation timing and period measurement	No
Speed min	[rpm]	Minimal speed threshold for sensorless speed control	No
Freewheel time	[sec]	Freewheel counter value	No
OL speed lim	[rpm]	Target open-loop speed; threshold to switch to closed-loop operation	No
Cmt count	[#]	Commutation number for open-loop start-up	No
1st cmt period	[sec]	First commutation period duration	No
Time off	[%]	Current decay period in percentage of actual commutation period	No

Integ thr corr.	[%]	Back-EMF integration threshold correction constant	Yes

[1] This parameter value is ignored as the BEMF voltage integration method is not used by the application.

5.2.5. Output file tab

The output file tab serves as a preview of the application constants corresponding to the tuned motor control application, as shown in *Figure 38*.



File Name: {FM_project_loc}/Sources/Config/BLDC_appconfig.h
Date: November 1, 2019, 15:15:3
Description: Automatically generated file for static configuration of the BLDC application

```
// Motor Parameters
//----------------------------------------------------------------------
// Pole-pair numbers                      = 2 [-]
// Back-EMF constant                      = 0.02865 [V.sec/rad]
// Phase current nominal                  = 2.3 [A]
// Phase voltage nominal                  = 12 [V]
// Speed motor nominal                    = 2000 [rpm]
//----------------------------------------------------------------------

// Application Scales
//----------------------------------------------------------------------
#define I_MAX                             (25.0)
#define U_DCB_MAX                         (45.0)
#define N_MAX                             (2500.0)

#define I_DCB_OVERCURRENT                 FRAC16(0.120000000000)
#define U_DCB_UNDERVOLTAGE                FRAC16(0.200000000000)
#define U_DCB_OVERVOLTAGE                 FRAC16(0.400000000000)
```

Figure 38. **Output file tab – Expert mode**

The constants are thematically divided into the groups according to selected control tabs as follows:

- Application scales
- Mechanical alignment
- BLDC control loop
- BLDC sensorless module
- FreeMASTER scale variables

Application tuning modes are not available for this tab.

Click the “Generate Configuration File” button to generate the content of the output file tab. The header file BLDC_appconfig.h is generated and saved to the default path MCSPTE1AK116_BLDC_6Step\Sources\Config.

5.2.6. Application control tab

The last tab available from the tab menu is the application control tab. The application control page is based on the graphical components to provide a user friendly control interface.

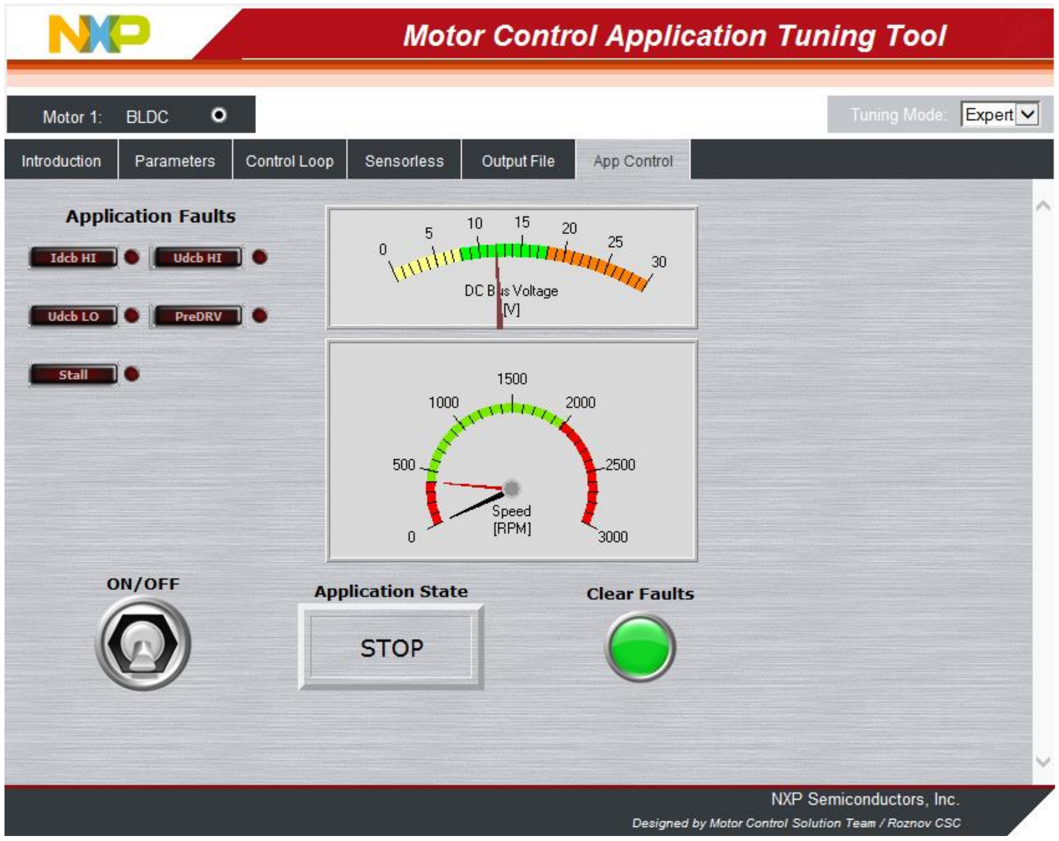


Figure 39. **Application control tab**

In this view, the most important variables and settings are displayed using a graphical representation. The fan can be switched on or off by using the “ON/OFF” switch or by changing the *appSwitchState* variable value in the Variable Watch Grid. The required speed can be selected either by clicking the speed gauge or by manually changing the *requiredSpeed* variable value in the Variable Watch Grid.

Where any fault is detected, it has to be cleared manually by clicking the green “Fault Clear” button or by setting the *faultSwitchClear* variable value to 0 in the Variable Watch Grid. Then, the application can be switched on again. Faults present in the system are signalized by the red fault indicators. Pending faults are signalized by small red circle indicators next to respective fault indicator.

6. Conclusion

The design described shows the simplicity and efficiency in using the S32K116 microcontroller for Sensorless BLDC 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 BLDC drive application tuning friendly and intuitive.

7. References

- MCSPTE1AK116, S32K116 Development Kit for BLDC and PMSM Motor Control
- S32 Design Studio IDE for ARM® based MCUs
- FreeMASTER Run-Time Debugging Tool
- S32K11XMCLUG , Automotive Math and Motor Control Library Set for S32K11x User Manual
- S32K1XXRM, S32K1xx Series Reference Manual
- S32K116EVB: S32K116 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

Document Number:AN13000
Rev. n
09/2020