NXP Semiconductors
Application Notes

Document Number: AN13767
Rev. 0, 11/2022

3-phase Sensorless PMSM Motor Control Kit with S32K344 using RTD Low Level API

Featuring Motor Control Application Tuning (MCAT) Tool

by: NXP Semiconductors

Contents

1. Introduction

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

This design serves as an example of motor control design using NXP S32K3 automotive family with MCUs based on a 32-bit Arm® Cortex-M7® core with IEEE-754 compliant single precision floating point unit optimized for a full range of automotive applications. An innovative drivers set, Real-Time Drivers (RTD), are used to configure and control the MCU. It complies with Automotive-SPICE, ISO 26262, ISO 9001 and IATF 16949. Low-level drivers of RTD and S32 Design Studio Configuration Tools (S32CT) are used to demonstrate non-AUTOSAR approach.

Following are the supported features:

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

2. System concept

The system is designed to drive a 3-phase PMSM. The application meets the following specifications:

- Based on the S32K3x4-Q172 development board for general-purpose industrial and automotive applications. See [1] for more information.
- DEVKIT-MOTORGD containing GD3000 MOSFETs pre-driver with extensive set of functions and condition monitoring (see [9] [10])
- Real-Time Drivers (RTD) and S32CT (non-AUTOSAR) used as S32K44 device configuration and control tool being a part of the S32 Design Studio for S32 Platform (S32DS) a NXP's complimentary integrated development environment (IDE) (see PMSM field oriented control)
- Control technique incorporating:
    - Field Oriented Control of 3-phase PM synchronous motor without position sensor
    - Closed-loop speed control with action period of 1ms
    - Closed-loop current control with action period of 100µs
    - Bi-directional rotation
    - Flux and torque independent control
    - Field weakening control extending speed range of the PMSM beyond the base speed
    - Position and speed is estimated by Extended Back Electromotive Force (eBEMF) observer or obtained by Encoder sensor
    - Open-loop start up with two stage alignment
    - Reconstruction of three-phase motor currents from two shunt resistors
    - FOC state variables sampled with 100 µs period
- Automotive Math and Motor Control Library (AMMCLIB) - FOC algorithm built on blocks of precompiled SW library (see [5])
- FreeMASTER software control interface (motor start/stop, speed setup) (see [4])
- FreeMASTER software monitor (monitoring/visualization of application variables)
- FreeMASTER embedded Motor Control Application Tuning (MCAT) tool (motor parameters, current loop, sensorless parameters, speed loop) (see [13])
- FreeMASTER software MCAT graphical control page (required speed, actual motor speed, start/stop status, DC-Bus voltage level, motor current and system status)
- FreeMASTER software speed scope (observes actual and desired speeds, DC-Bus voltage and motor current)
- FreeMASTER software high-speed recorder (reconstructed motor currents, vector control and algorithm quantities)
- DC-Bus over-voltage and under-voltage, over-current, overload and start-up fail protection

3. PMSM field oriented control

3.1. Fundamental principle of PMSM FOC

High-performance motor control is characterized by smooth rotation over the entire speed range of the motor, full torque control at zero speed, and fast acceleration/deceleration. To achieve such control, Field Oriented Control is used for PM synchronous motors.

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

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

Equation 1

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

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

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

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

Setting the reference frame speed synchronously with the rotor flux axis further results into d and q axis current components appearing as DC values. This implies utilization of simple current controllers to control the demanded torque and magnetizing flux of the machine, thus simplifying the control structure design.

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

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

Clarke/Park transformations discussed above are part of the Automotive Math and Motor Control Library set (see [5]).

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

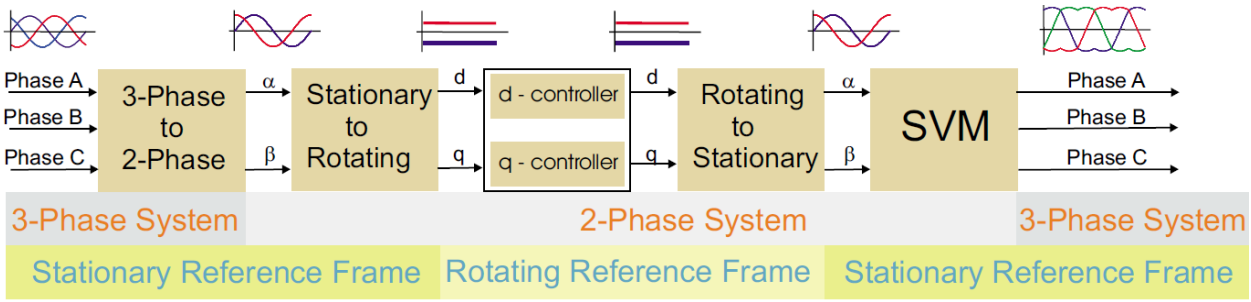

Figure 1. **Field oriented control transformations**

3.2. PMSM model in quadrature phase synchronous reference frame

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

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

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

Equation 2

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

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

Equation 3

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

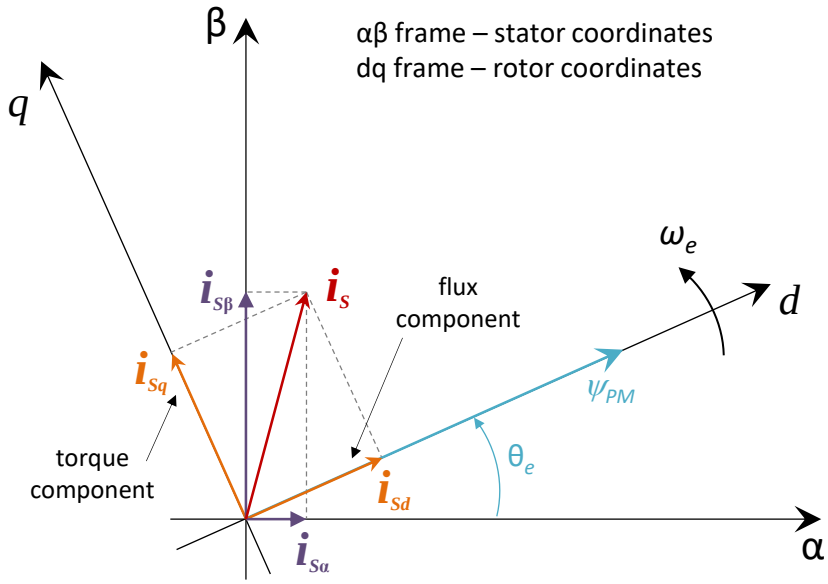

Figure 2. **Orientation of stator (stationary) and rotor (rotational) reference frames, with current components transformed into both frames**

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

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

Equation 4

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

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

Equation 5

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

Equation 6

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

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

Equation 7

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

Equation 8

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

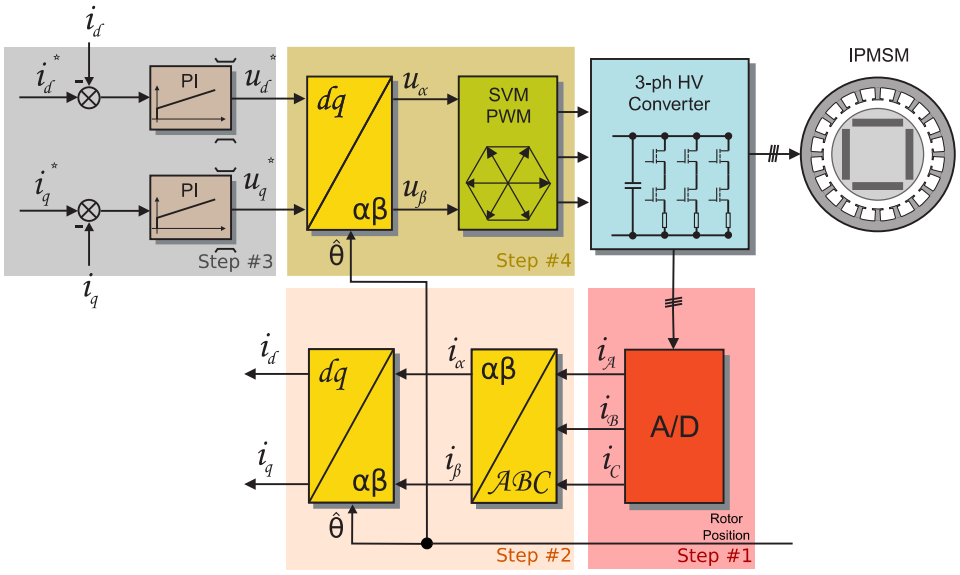

Figure 3. **FOC Control Structure**

3.3. Phase current measurement and output voltage actuation

The 3-phase voltage source inverter shown in *Figure 4* uses three shunt resistors (R56, R57, R58) placed in three legs of the inverter as phase current sensors. Stator phase current which flows through the shunt resistor produces a voltage drop which is interfaced to the Analog-to-Digital Converter (ADC) of microcontroller through conditional circuitry. Shunt resistor R60 is used as DC current sensor. Voltage drop is interfaced to GD3000 pre-driver internal operational amplifier and pre-driver is using it to detect an overcurrent event. (refer to DEVKIT-MOTORGD Schematic available at [9]).

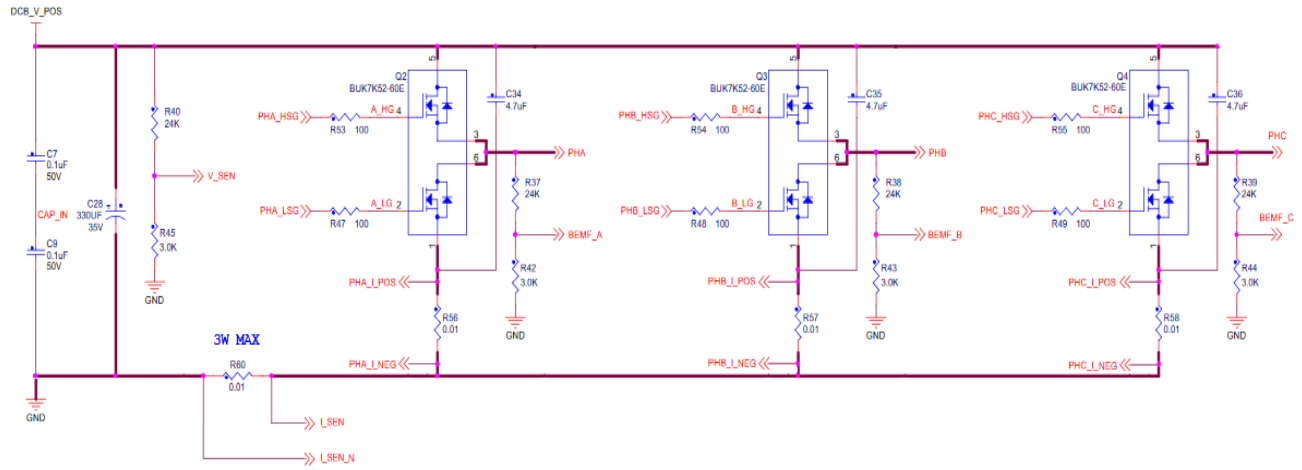

Figure 4. **3-phase DC/AC inverter with shunt resistors for current measurement**

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

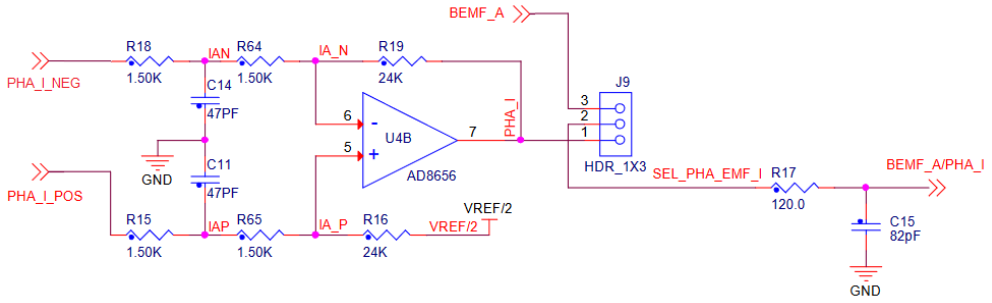

Figure 5. **Phase current measurement conditional circuitry**

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

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

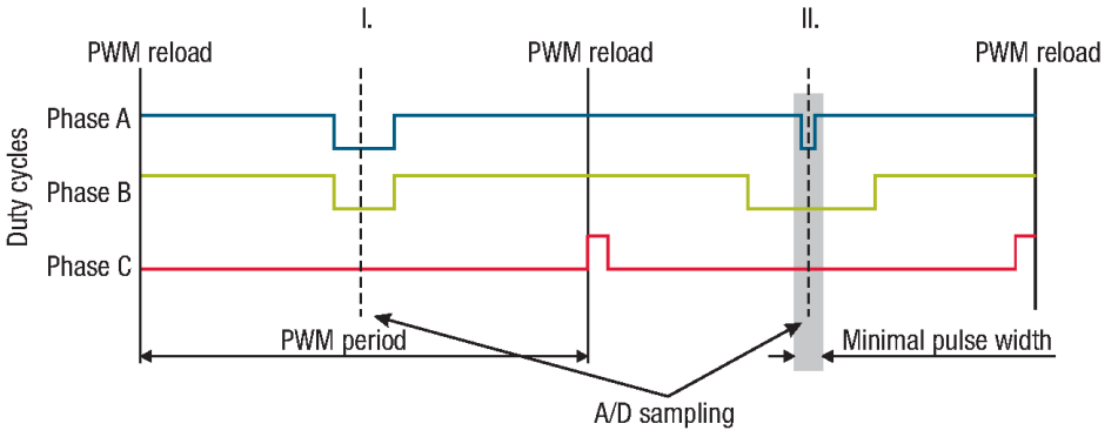

Figure 6. **Generated phase duty cycles in different PWM periods**

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

$$i_A + i_B + i_C = 0$$

Equation 9

NOTE

Although, there are three shunt resistors available on the power stage board (R56, R57, R58) and S32K344 has three AD converters, only two currents are measured simultaneously in this application in order to demonstrate ADC Single-shot mode and BCTU control mode in parallel. Third stator current is calculated based on *Equation 9*. To measure two stator currents in two inverter legs correctly, minimum ON times for the low-side switches are ensured by appropriate duty cycle limit.

3.4. Rotor position/speed estimation

In this application, rotor position and speed are either estimated sensorless by eBEMF observer or obtained by Encoder sensor. eBEMF observer as well as incremental Encoder sensor provide only relative position. To get absolute position, initial position must be known. This application uses mechanical rotor alignment when the rotor is moved from unknown to known position. The two stage alignment process is described in details in the section *State – ALIGN*.

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

Application in encoder mode can start from zero speed because speed and position are provided by sensor.

Structure and implementation details are discussed in section *AMMCLib Integration*.

3.5. Field weakening

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

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

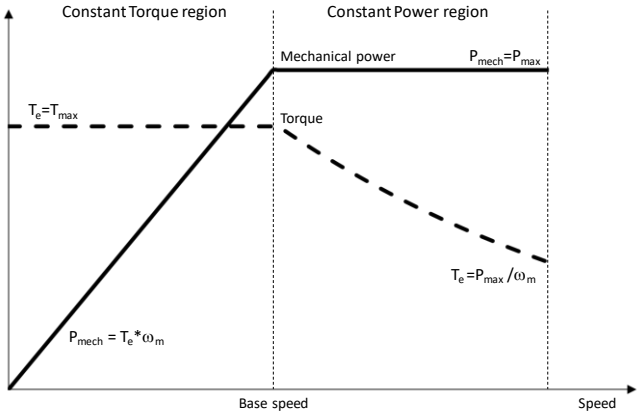

Figure 7. **Constant torque/power operating regions**

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

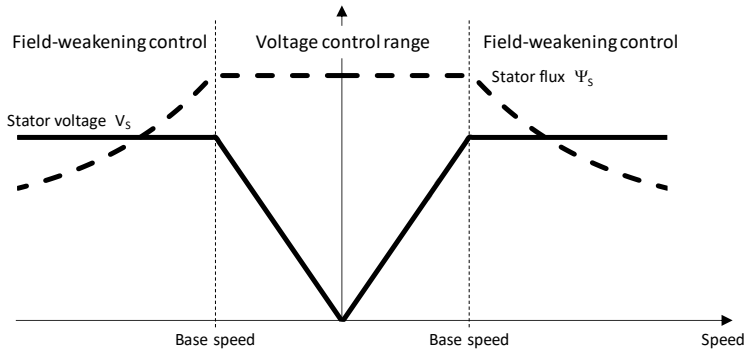

Figure 8. **Constant flux/voltage operational regions**

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

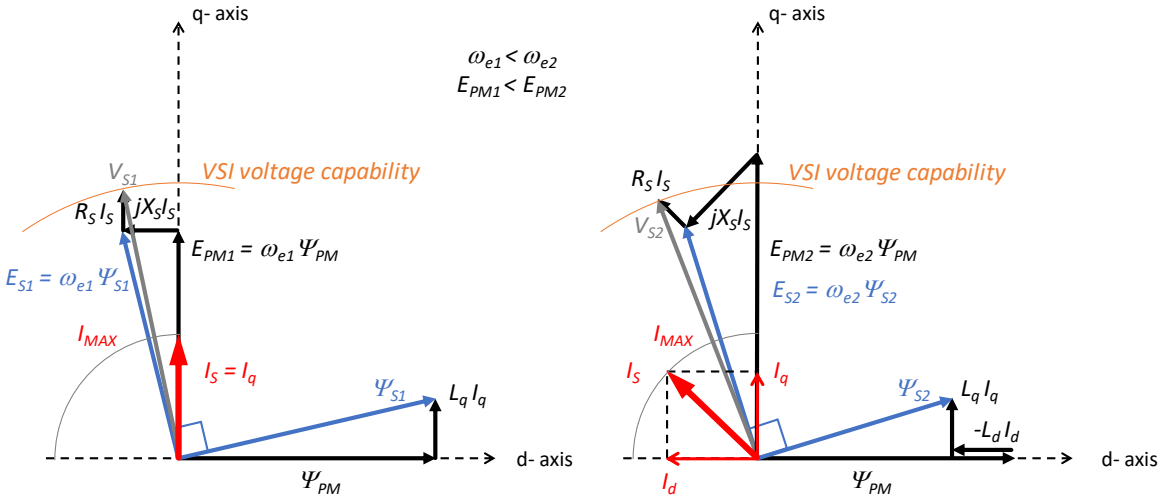

Figure 9. **Steady-state phasor diagram of PMSM operation up to base speed (left) and above speed (right)**

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

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

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

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

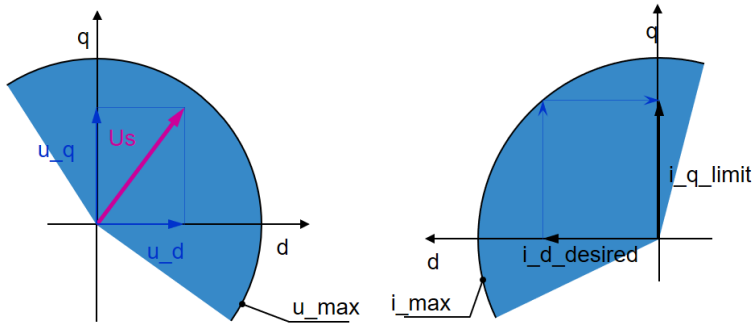

Figure 10. **Voltage (left) and current (right) limits for PMSM drive operation**

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

4. Software implementation on the S32K344

4.1. S32K344 – Key modules for PMSM FOC control

The S32K344 device includes modules such as the Enhanced Modular IO Subsystem (eMIOS), Logic Control Unit (LCU), Trigger MUX (TRGMUX), Body Cross-triggering Unit (BCTU) and Analogue-to-Digital Converter (ADC) suitable for real-time control applications, in particular, motor control applications. These modules are directly interconnected and can be configured to meet various motor control application requirements. *Figure 11* shows a module interconnection for a typical PMSM FOC application working in sensorless or sensor-based mode using dual shunt current sensing and encoder position sensor. The modules are described below and the detailed description can be found in the S32K3xx Reference Manual (see [7]).

4.1.1. Module interconnection

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

eMIOS plays a role of the real time timer/counter. Within the control loop it is responsible for generation of PWM signal (period, duty cycle), generation of the trigger for analogue data capturing in the precise moment or counting edges of encoder signal. LCU enriches this modular concept with advance features. In PWM generation it is responsible for creation of PWM complementary pairs, dead time insertion, disabling/enabling PWM outputs or it preprocess signals from an encoder sensor to get quadrature decoder functionality.

BCTU and ADC modules are responsible for analog data capturing. BCTU answers question “what is going to be measured?” by a predefined list of ADC channels. The ADC answers question “How it is going to be measured? ” by setting a conversion resolution, sampling duration etc.

eMIOS and LCU are connected through TRGMUX unit which is responsible for a configurable signal interconnection within the microcontroller. The eMIOS channels CH1-CH3 create 3-phase center aligned PWM signal and share PWM time base CH0. The center aligned PWM is formed using flexible Output Pulse Width Modulation Buffered (OPWMB) eMIOS mode where each channel uses two compare registers (A, B) to control rising and falling edge independently. LCU OUT0-OUT5 create commentary PWM pairs to control particular MOSFET transistors. The LCU uses true tables, output polarity control and configurable digital filters to generate control signals for transistors with inserted deadtime. The eMIOS CH4 is dedicated for trigger functionality. Same as in case of PWM signals OPWMB mode is also used for trigger. The CH4 is linked with trigger time base CH23. Time bases CH0 and CH23 are synchronized, what offers possibility of an independent configuration of sampling and PWM frequency.

BCTU is linked with eMIOS channels through the channel flag. When the flag is set, BCTU starts to execute conversions according to the list of conversions and clears the flag back. BCTU is capable of controlling all three ADCs, so list of single or parallel conversions can be invoked. In this example a list of parallel conversions of ADC0 and ADC1 is used to obtain phase currents and DC-bus voltage. Conversion results are stored to BCTU FIFO. ADC2 is used for microcontroller temperature measurement to demonstrate non real time background measurement.

Quadrature decoder functionality is achieved by cooperation of eMIOS, LCU and TRGMUX. LCU decodes encoder signals PHA and PHB into digital signals, which carry captured edges per particular rotor direction. The eMIOS module works as a counter and holds captured edges for clockwise CH5 and counter clockwise CH6 direction. Absolute position is obtained by subtracting counters values.

Detailed description can be found in the S32K3xx Reference Manual (see [7]).

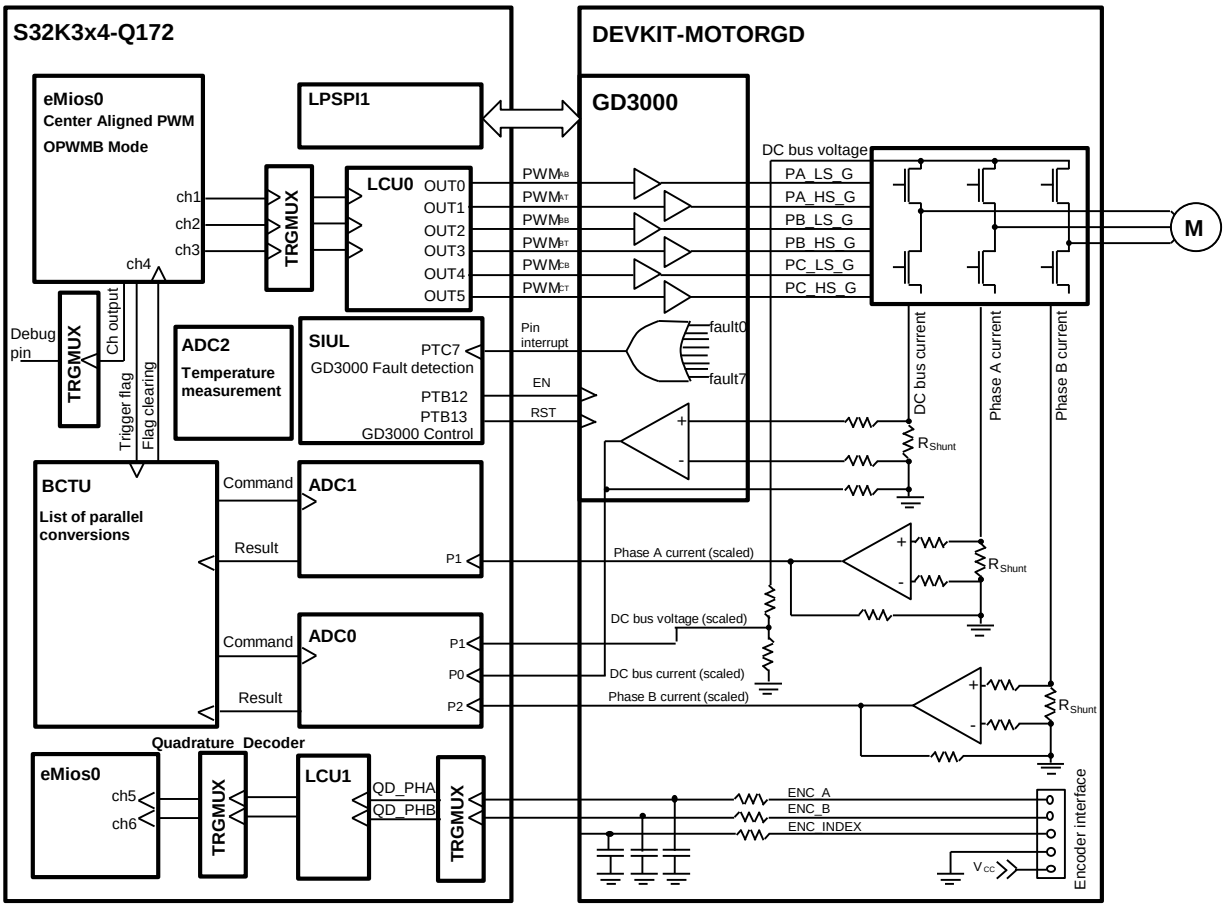

Figure 11. **S32K344 module interconnection**

4.1.2. S32K344 and FETs pre-driver interconnection

Excitation of power FETs is ensured by NXP GD3000 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 [9] [10]). NXP's Three-Phase Brushless Motor Pre-Driver Software Driver (TPP) is used to control and to configure GD3000.

Configuration of GD3000 pre-driver is realized via LPSPI1 module. The GD3000 allows different operating modes to be set and locked by SPI commands. SPI commands also report condition of the GD3000 based on the internal monitoring circuits and fault detection logic. S32K344 detects fault state of the GD3000 by means of interrupt signal on PTC7 pin. Integrated current sensing amplifier with analog comparator allow to measure DC bus current and detect overcurrent. Interconnection between S32K344 and GD3000 is briefly depicted in *Figure 11*.

4.1.3. Module involvement in digital PMSM FOC control loop

This section will discuss timing and modules synchronization to accomplish PMSM FOC on the S32K344 and the internal hardware features.

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

The PMSM FOC control with dual-shunt current measurement is based on static timing. It means the trigger point of the ADC conversions is located at same place within every control loop cycle. This trigger point is also configurable during runtime.

eMIOS timer uses the concept of time bases for signal synchronization. There are 5 channels (CH0, CH8, CH16, CH22 and CH23) which can act as the time base what means that other channels can see value of their counter through the bus. CH0,CH8,CH16 can create local time bases for 7 channels and CH22 and CH23 can create a global time bases for any channel. In the example CH0 creates the PWM time base for channels CH1,CH2 and CH3 which are responsible for PWM signal generation. The CH23 creates a TRIGGER time base for CH4 which is responsible for triggering BCTU. Both time bases operate in Modulus Counter Buffered (MCB) up counting mode, where period is set by register A. It is possible to start time bases synchronously by enabling eMIOS global prescaler. Offset between time bases is given by time base channel initial counter value. In this example time bases are synchronous with no offset.

PWM frequency is 20 kHz and sampling frequency is 10 kHz. PWM channels and trigger channels operates in OPWMB mode. Channel output signal is formed by comparing channel registers A and B with time base counter. For example PWM signal for phase A is generated by output of the CH1. Center aligned PWM is achieved by proper setting of registers A and B. PWM A signal is routed to LCU where complementary signals for particular MOSFETs are created (LCU0 OUT0 and OUT1) respecting pre-driver input polarity and the dead time is inserted.

Trigger signal CH4 is formed in the same way as PWM signals. An important point here is that the connection between BCTU and CH4 is through the CH4 flag and not through the CH4 output. Flag can be generated on both compares or on compare with register B only. In this example, the flag is set on register B only it means on falling edge of the CH4 output signal. CH4 output signal can be routed using the TRGMUX to microcontroller pin for trigger debugging.

When flag of eMIOS CH4 is set, the BCTU starts list of conversions controlling ADC0 and ADC1 and also clears back the CH4 flag. I_{PHA} and I_{PHB} stator currents are measured simultaneously at the beginning of PWM cycle, which is in the middle of non-active vector, where bottom MOSFETs of both inverter legs are open, and currents flow through shunt resistors. DC-bus voltage U_{DCbus} is measured in the following sample. The ADC results are stored into BCTU FIFO result registers and interrupt is raised on watermark event. FOC control algorithm calculates new duty-cycle values based on measured currents and DC-bus voltage and updates eMIOS channels CH1, CH2, CH3. Register A and B are double-buffered so change will be coherently propagated on channels time base reload.

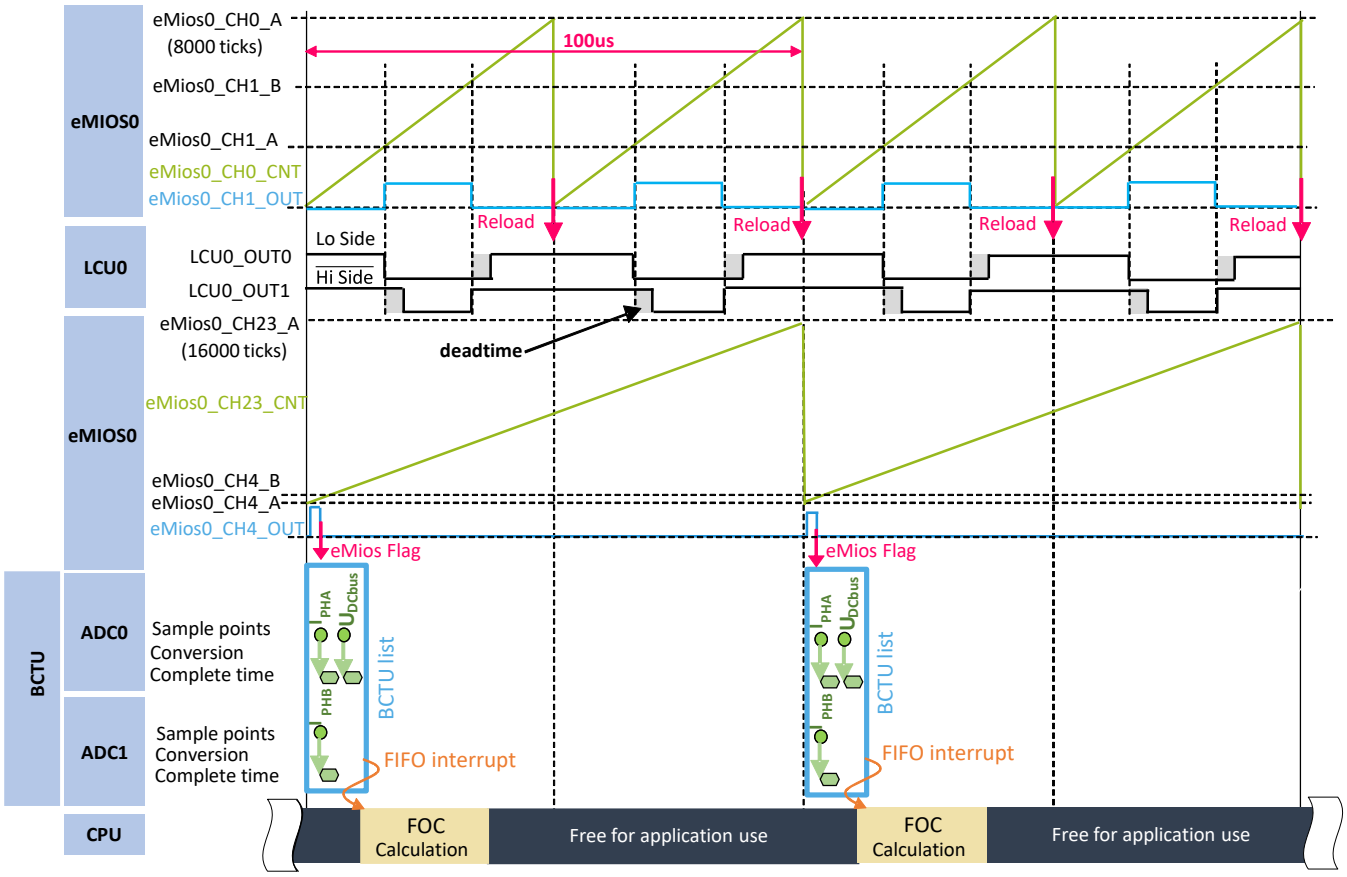

Figure 12. **Time Diagram of PWM and ADC Synchronization**

4.2. S32K344 device initialization

To simplify and accelerate an application development, embedded part of the PMSM FOC motor control application has been created using S32 Design studio, RTD drivers (low level part) and S32K344 is configured using S32 Configuration Tools, see the following figure.

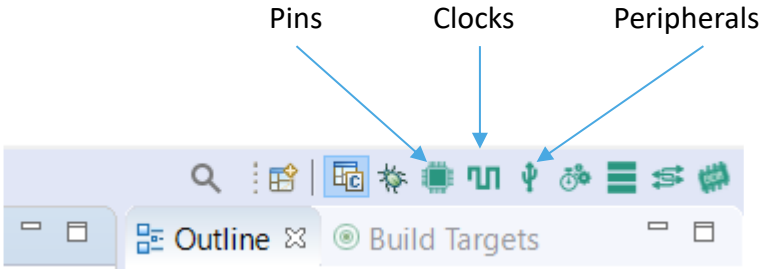

Figure 13. **Config tools**

Figure 14 describes the example project structure in the S32 Design Studio. Current settings of Config tools are stored in MCSPTE1AK344_PMSM_FOC_2Sh_ll.mex file and generated files by config tools (all configuration structures) can be found in folders board and generate. When a component is added using the config tool, its SW driver is copied into folder RTD so only used drivers are part of the project.

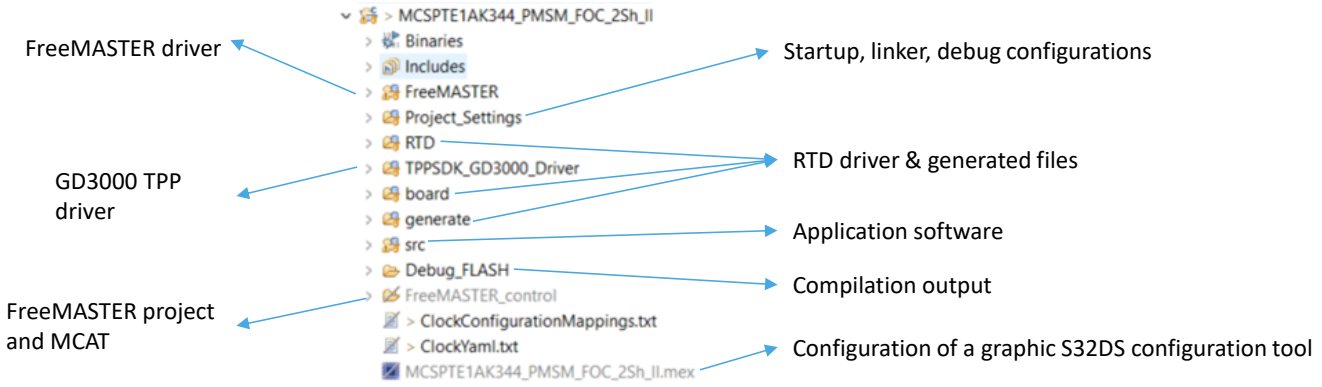

Figure 14. **Example project structure**

Peripherals are initialized at beginning of the main() function. For each S32K344 module, there is a specific initialization function, that uses configuration structures generated by Config tools to configure the MCU. XXX_Init functions must be called before any other Application Programming Interface (API) from the module. It is important to initialize Clock and OsIf at first. OsIf initializes systic timer which can be used for timeout measurements in other modules. The last function to call during the initialization process is Emios_Mcl_Ip_Init. It initializes time bases and enables their counters what initiate control cycle.

List of the initialization APIs:

- Clock_Ip_Init() - Initializes MCU clock configuration
- OsIf_Init() - Initializes the OS interface (basic timing/Os services for drivers)
- IntCtrl_Ip_Init() - Initializes the configured interrupts
- IntCtrl_Ip_ConfigIrqRouting() - Initializes interrupt handlers
- Siul2_Port_Ip_Init() - Initializes PINs and PORT configuration
- Trgmux_Ip_Init() - Initializes TRGMUX module configuration
- Lpuart_Uart_Ip_Init() - Initializes LPUART module configuration
- Adc_Sar_Ip_Init() - Initializes ADC modules configuration
- Lcu_Ip_Init() - Initializes LCU module configuration
- Lpspi_Ip_Init() - Initializes LPSPI module configuration
- Siul2_Icu_Ip_Init() - Initializes input capture configuration for External Interrupt Request (EIRQ)
- Emios_Pwm_Ip_InitChannel() - Initializes emios PWM and Trigger channels configuration
- Emios_Icu_Ip_Init() - Initializes eMios input capture configuration
- Bctu_Ip_Init() - Initializes BCTU module configuration
- Emios_Mcl_Ip_Init() - Initializes eMios time-bases configuration

RTD documentation can be found in the folder created in the S32 Design Studio installation path:

"c:\NXP\S32DS\software\PlatformSDK_S32K3_2022_03\SW32K3_RTD_4_4_2_0_0_D2203"

4.2.1. Port control and pin configuration

PMSM FOC sensorless motor control application requires following on chip pins assignment:

Table 1. **Pins assignment for S32K344 PMSM Sensorless FOC control**

Module	Signal name	Pin name / Functionality	Description
LCU0	PWMA_HS	PTD2 / LCU0_OUT1	PWM signal for phase A high-side driver (inverted)
	PWMA_LS	PTD3 / LCU0_OUT0	PWM signal for phase A low-side driver
	PWMB_HS	PTA2 / LCU0_OUT3	PWM signal for phase B high-side driver (inverted)
	PWMB_LS	PTA3 / LCU0_OUT2	PWM signal for phase B low-side driver
	PWMC_HS	PTA1 / LCU0_OUT5	PWM signal for phase C high-side driver (inverted)
	PWMC_LS	PTA0 / LCU0_OUT4	PWM signal for phase C low-side driver
ADC0	DCB_V	PTD0 / ADC0_P1	DC bus voltage measurement
	PHB_I	PTA8 / ADC0_P2	Phase B stator current measurement
ADC1	PHA_I	PTA13 / ADC1_P1	Phase A stator current measurement
LPSPI1	GD3000_CLK	PTB14 / LPSPI1_SCK	SPI clock (1MHz)
	GD3000_SIN	PTB15 / LPSPI1_SIN	SPI input data from GD3000
	GD3000_SOUT	PTB16 / LPSPI1_SOUT	SPI output data for GD3000
LPUART6	FMSTR_TX	PTA16 / LPUART6_RX	UART transmit data (FreeMASTER)
	FMSTR_RX	PTA15 / LPUART6_TX	UART receive data (FreeMASTER)
TRGMUX	TST_TGMX_O12_B21	PTB21 / TRGMUX_OUT12	Pin for debugging microcontroller internal signals
	TST_TGMX_O9_B18	PTB18 / TRGMUX_OUT9	Pin for debugging microcontroller internal signals
	ENC_PHA	PTA19 / TRGMUX_IN13	Phase A signal of the Encoder sensor
	ENC_PHB	PTA20 / TRGMUX_IN14	Phase B signal of the Encoder sensor
SIUL2	GD3000_EN	PTB12 / GPIO	Enable signal for GD3000
	GD3000_RST	PTB13 / GPIO	Reset signal for GD3000
	GD3000_CS	PTB17 / GPIO	Chip select signal for GD3000
	GD3000_INT	PTC7 / EIRQ7	Interrupt signal indicating GD3000 fault
	TST_GPIO_C24	PTC24 / GPIO	GPIO toggling to measure execution time
	TST_GPIO_B20	PTB20 / GPIO	GPIO toggling to measure execution time
	BTN_INC_SW5	PTB26 / GPIO	Application control via board button SW5
	BTN_DEC_SW6	PTB19 / GPIO	Application control via board button SW6
	LED_RED	PTA29 / GPIO	RGB_RED indicating fault state
	LED_GREEN	PTA30 / GPIO	RGB_GREEN indicating ready/calib state
	LED_BLUE	PTA31 / GPIO	RGB_BLUE indicating run state

Pin tool and Peripherals tool simplify configuration and particular RTD drivers offers an API to control the ports during the runtime.

4.2.1.1. SIUL2

System Integration Unit Lite2 (SIUL2) is a peripheral which provides control over all electrical pin controls and ports. It enables selection of the functions and electrical characteristics that appear on external chip pins. The pins assignment can be carried out by means of Pins tool. The pin assignment of the example is shown in *Figure 15* . Electrical characteristics as well as functionality are set in "*Routing Details"* tab. Tool also offers visualization of the pinout placement in the selected package.

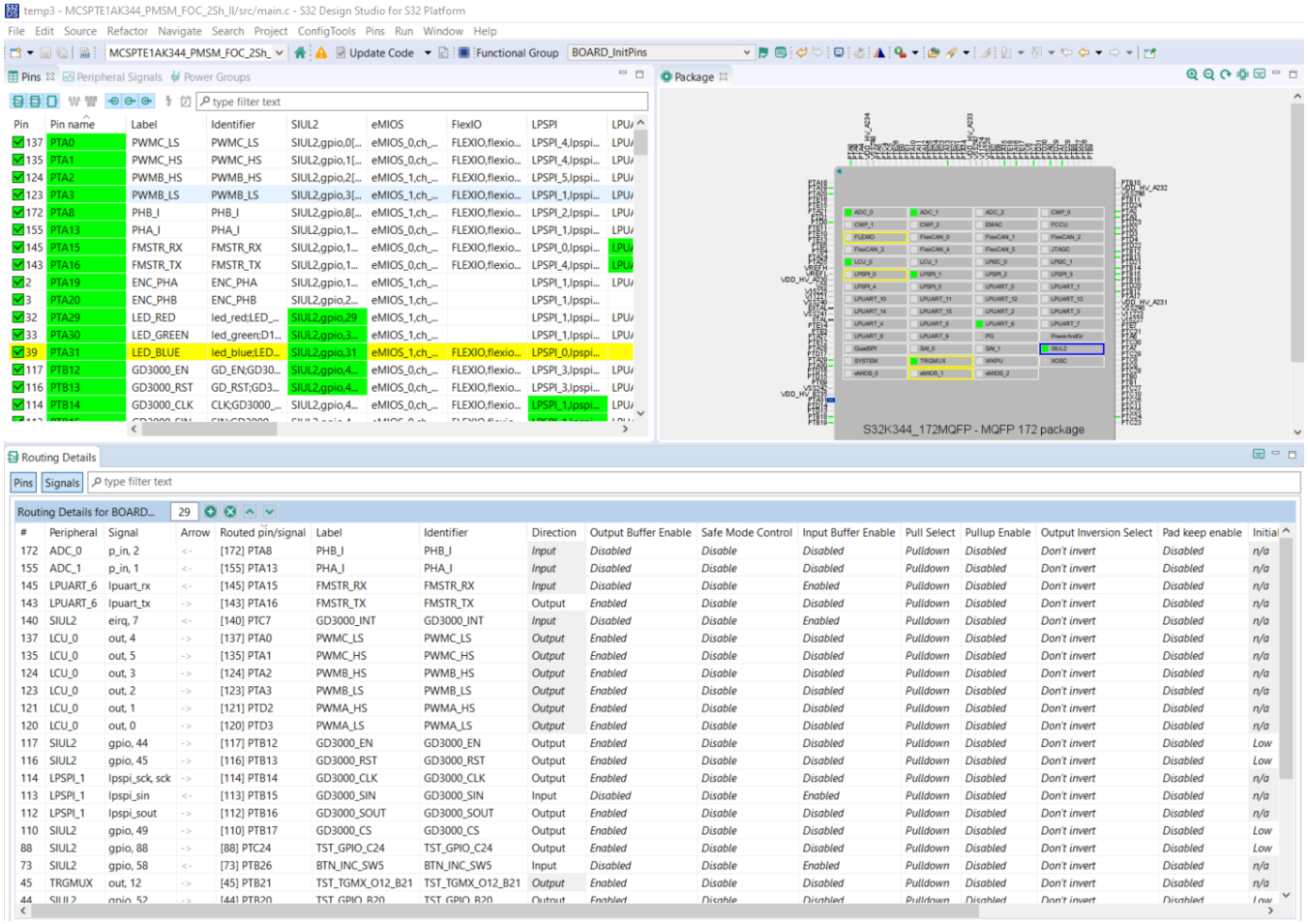

Figure 15. **Pins**

In order to control SIUL2, following drivers are used and configured using Peripherals tool.

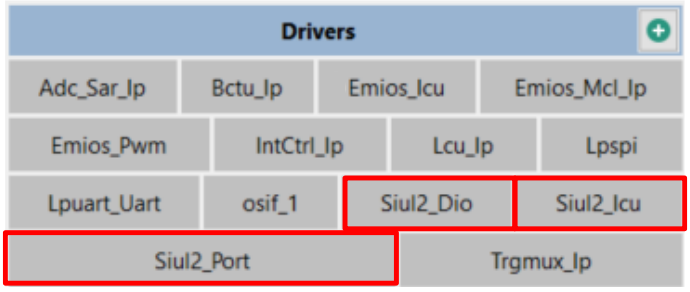

Figure 16. **Pins SW drivers**

Siul2_Dio and Siul_Port drivers uses configuration generated by Pins tool. Siul_Port initializes all pins and Siul2_Dio is used to control GPIO functionality as is shown in *Example 1* .

Example 1. Pin control API

```
void main (void)
{
...

  Siul2_Port_Ip_Init(NUM_OF_CONFIGURED_PINS0, g_pin_mux_InitConfigArr0);
...
}
```

```
void BctuFifoNotif(void)
{
    cntrState.usrControl.btSpeedUp = Siul2_Dio_Ip_ReadPin(BTN_INC_SW5_PORT, BTN_INC_SW5_PIN);
    cntrState.usrControl.btSpeedDown = Siul2_Dio_Ip_ReadPin(BTN_DEC_SW6_PORT, BTN_DEC_SW6_PIN);
...
    Siul2_Dio_Ip_SetPins(TST_GPIO_C24_PORT, (1 << TST_GPIO_C24_PIN));
...
    Siul2_Dio_Ip_ClearPins(TST_GPIO_C24_PORT, (1 << TST_GPIO_C24_PIN));
...
}
```

Pin PTC7 is used for GD3000 fault state detection so it is configured to external IRQ functionality by Pins tool. Siul2_Icu driver is responsible for configuration of an external pin input capture event. In "*IcuHwInterruptConfigList"* tab the driver is informed whether interrupt is going to be used for IRQ signal and proper interrupt handler is enabled and can be used in interrupt configuration as is described in *Interrupts*. In "*IcuSiul2"* tab a prescaler and an interrupt filter for specific EIRQ signal is set to eliminate interrupt on random glitches on the pin.

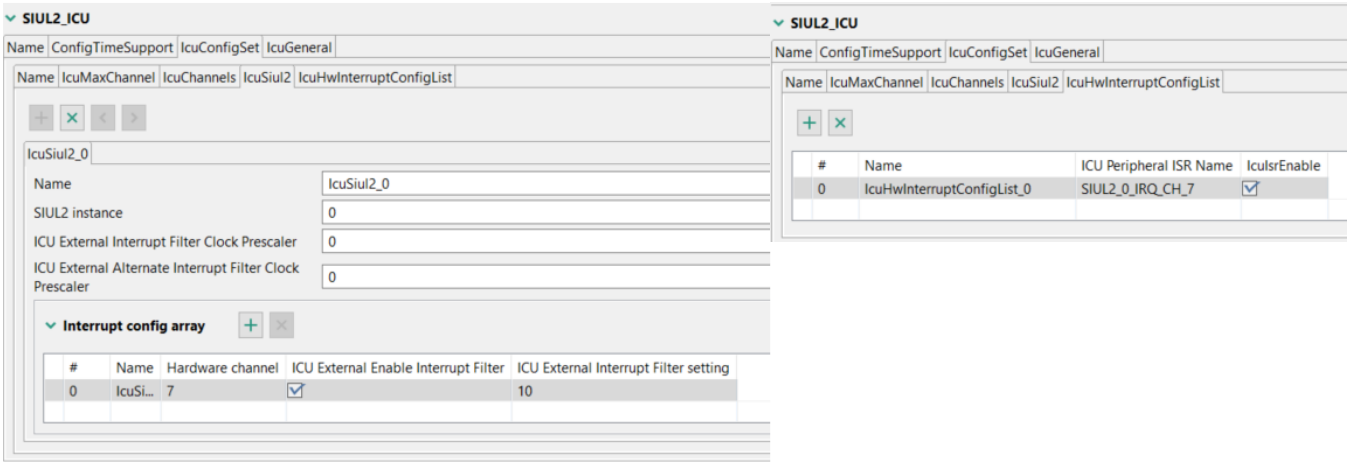

Figure 17. **SIUL EIRQ configuration**

"IcuChannels" tab configures more general settings like mode, which edge of the signal should be detected and which notification function should be called on this event. Notification function is part of custom code. Previous settings are referenced through "*IcuChannelRef* " parameter. API functions must be called as is shown in the *Example 2* in order to apply config settings, to enable interrupt and to enable notification function at SIUL level.

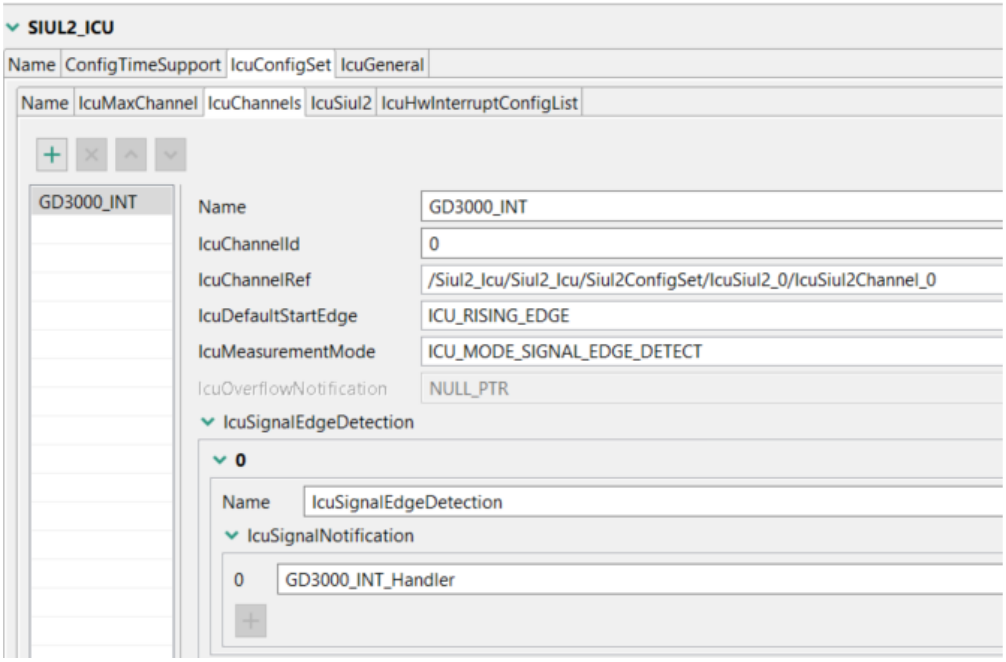

Figure 18. **ICU channel configuration**

Example 2. Pin input capture API

```
void main (void)
{
...

    /* Initialize ICU channel for GD3000 interrupt. */
    Siul2_Icu_Ip_Init(SIUL2_ICU_IP_INSTANCE, &Siul2_Icu_Ip_0_Config_PB_BOARD_InitPeripherals);
    /* Enable ICU edge detect for GD3000 interrupt. */
    Siul2_Icu_Ip_EnableInterrupt(SIUL2_ICU_IP_INSTANCE, 7U);
    Siul2_Icu_Ip_EnableNotification(SIUL2_ICU_IP_INSTANCE, 7U);
...
}
```

Example 3. SIUL ICU notification function

```
void GD3000_INT_Handler (void)
{
    /* Set GD3000 INT flag. */
    gd3000Status.B.gd3000IntFlag = true;
}
```

4.2.1.2. TRIGGER MUX

The TRGMUX peripheral provides an extremely flexible mechanism for interconnection of various trigger sources to multiple pins/peripherals. It is a very useful feature for debugging. This is configured using Trgmux_Ip driver.

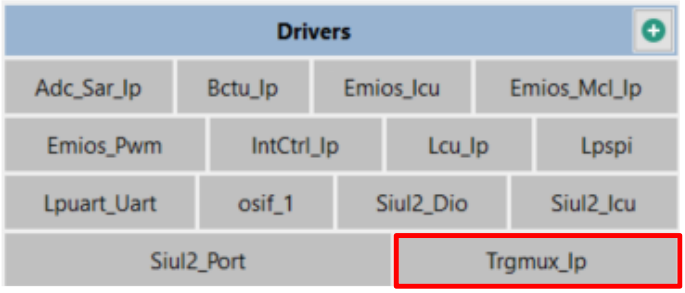

Figure 19. **TRGMUX SW driver**

TRGMUX implements configurable connection between peripherals, which offers flexible triggering scheme in S32K3 device. This device has 16 pads (SIUL2) mapped to TRGMUX inputs and TRGMUX outputs, so internal signals can be visualized to output pin. In the example pins PTB18 (TRGMUX out 9) and PTB21(TRGMUX out 12) are selected as pins for internal signal monitoring. Connection is created within TRGMUX hardware group. For example hardware group TRGMUX_IP_SIUL_12_15 gathers TRGMUX SIUL outputs 12-15. The connection is made by selecting specific hardware output and input. PTB18 visualizes output of eMIOS0 CH4 (which is a trigger signal for analogue capturing) and PTB21 visualizes eMIOS0 CH1 which is PWM signal for phase A. Other signals like reload can be visualized by changing the "*Hardware Input"* configuration. Setting is applied by calling Trgmux_Ip_Init function. Full list of all possible interconnections can be found in S32K3XX_TRGMUX_connectivity.xls attached to S32K3xx Reference Manual [7] .

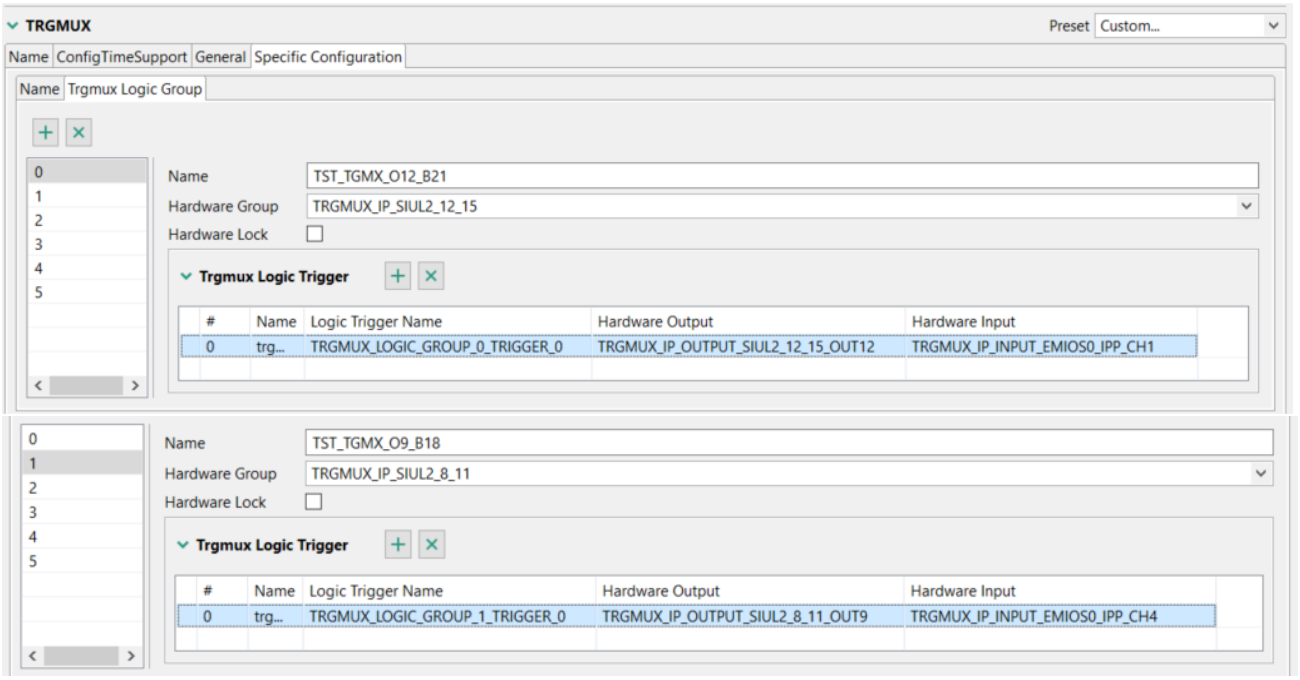

Figure 20. **TRGMUX groups for debugging purposes**

4.2.2. Clock and Interrupt configuration

In order to configure S32K3 clocks and interrupts RTD offers Clocks Configuration tool companioned by Clock_Ip driver and Peripherals tool for OSIF and IntCrlt_Ip driver configuration.

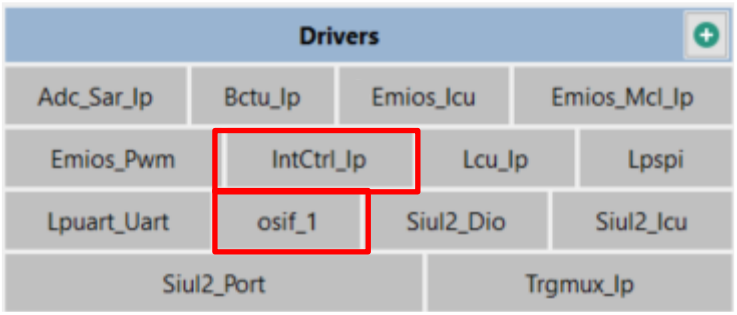

Figure 21. **OS Interface and interrupts**

4.2.2.1. Clocking

S32K344 features a complex clocking sourcing by Fast internal RC oscillator (FIRC), Slow internal RC oscillator (SIRC), Fast external crystal oscillator (FXOSC), Slow external crystal oscillator (SXOSC), Phase-locked loop (PLL), Clock Generation Module (MC_CGM), Mode Entry module's (MC_ME).

To run the core of the S32K344 at maximum frequency 160MHz, S32K344 is supplied externally by 16 MHz crystal. This clock source supplies Phase-lock-loop (PLL) and its output is adjusted to 160 MHz frequency. PLL output PHI0 is then used to supply the core CORE_CLK. All real-time control peripherals are supplied by CORE_CLK , what eliminates unwanted wait states on the bus when peripherals are controlled by core during runtime.

This clock configuration can be setup by S32 Clock Configuration tool which offers visual graphical user interface (GUI) to change the settings. Clock settings are applied by calling Clock_Ip_Init() function, where generated configuration by Clocks tool is an argument.

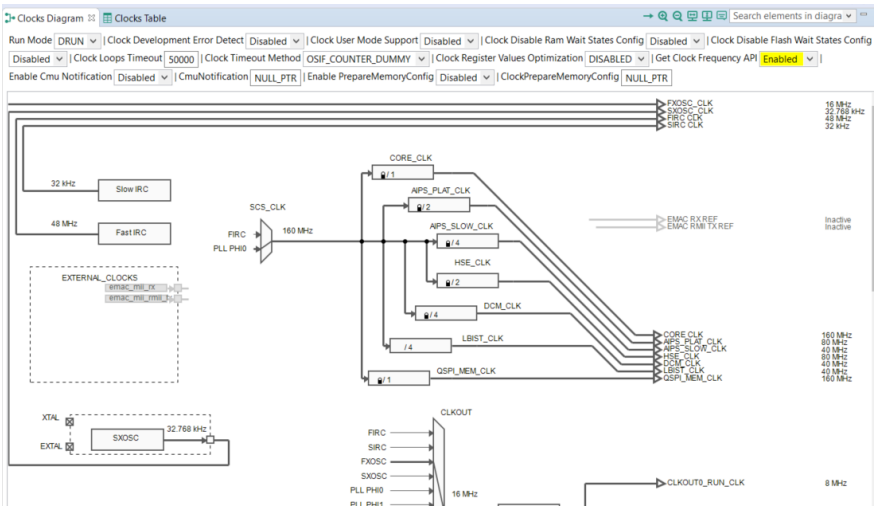

Figure 22. **Clocks tool**

Clock setting is summarized in the following table.

Table 2. **S32K144 clock configuration**

Clock	Frequency	Peripheral
CORE_CLK	160 MHz	ADC0-2,BCTU,LCU0-1,eMIOS0-2
AIPS_SLOW_CLK	40 MHz	LPSPI1, LPUART6,TRGMUX

Operating System Interface (OSIF) driver provides basic timing/OS services for drivers, allowing for OS independent implementations. This example is baremetal without operating system, but other drivers can use OSIF for timeouts detection. OSIF settings are applied by calling OsIf_Init() function.

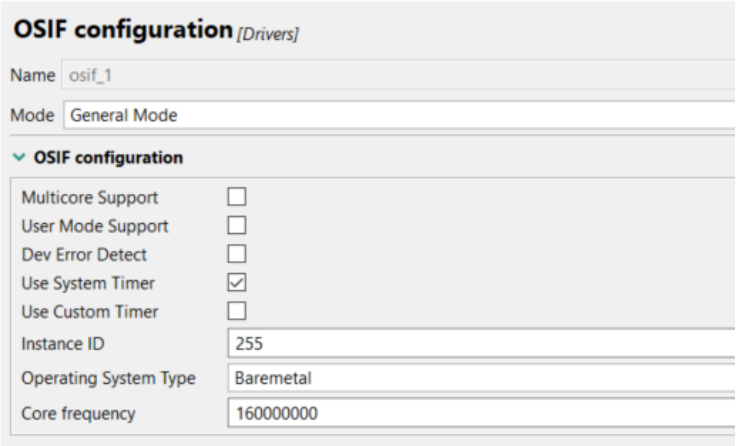

Figure 23. **Clocks tool**

4.2.2.2. Interrupts

IntCrtl_Ip driver is responsible for an interrupt configuration on S32K3 platform. Settings impact Miscellaneous System Control Module (MSCM), Nested vectored interrupt controller (NVIC), and

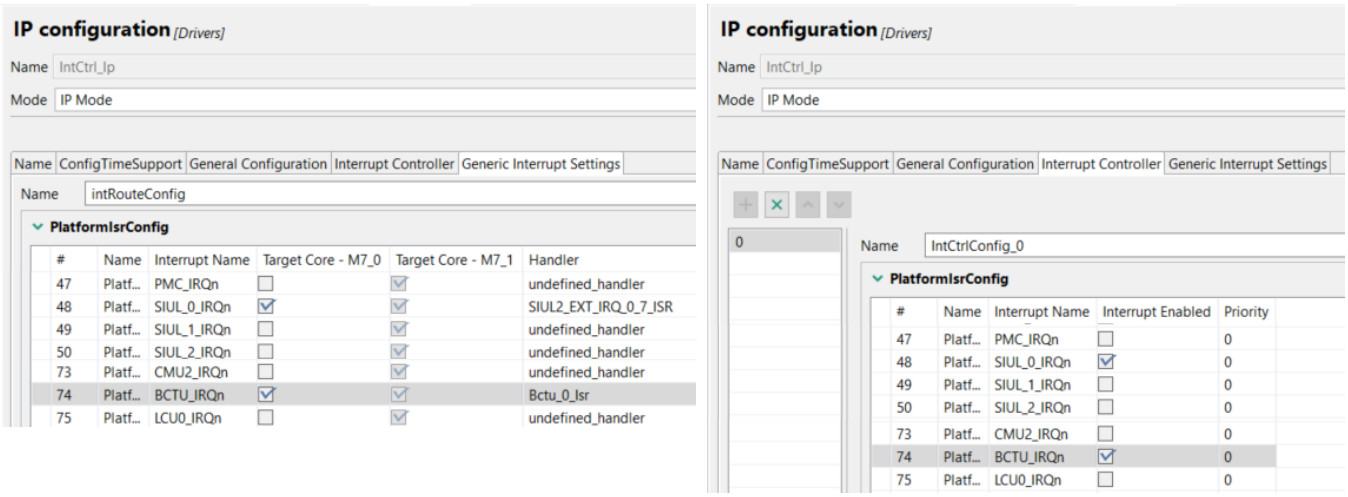

Figure 24. **Interrupt controller**

interrupt vector table. The example uses two interrupts: External IRQ from pin and Interrupt from BCTU. There are three options to set in "*Generic Interrupt settings"* column "*Handler"*: undefined handler user can set also own custom handler (but this interrupt service routine must be defined in custom code) or interrupt service routine from RTD driver. Naming of RTD interrupt service routines can be found in integration manual of particular RTD driver. Bctu_0_Isr and SIUL2_EXT_IRQ_0_7_ISR handle their interrupt and call notification functions on specific event defined by Siul2_Icu and Bctu_Ip component settings in peripheral tool (GD3000_INT_Handler, BctuFifoNotif).

Interrupt setting and handlers installation to vector table are realized by calling IntCtrl_Ip_Init(), IntCtrl_Ip_ConfigIrqRouting().

Example 4. Clock API

```
void main (void)
{
...

    /*****************************************************************************************************
    *Configure and enable interrupts
    *****************************************************************************************************/
    IntCtrl_Ip_Init(&IntCtrlConfig_0);
    IntCtrl_Ip_ConfigIrqRouting(&intRouteConfig);
...
}
```

4.2.3. Center-aligned PWM

Generation of the center aligned PWM functionality is realized by modules eMIOS, TRGMUX and LCU. In order to configure and control those peripherals following RTD drivers are used: Emios_Mcl_Ip to configure eMIOS timebase, Emios_Pwm to configure and control eMIOS PWM channels, Lcu_Ip to configure and control LCU and Trgmux_Ip to interconnect eMIOS and LCU.

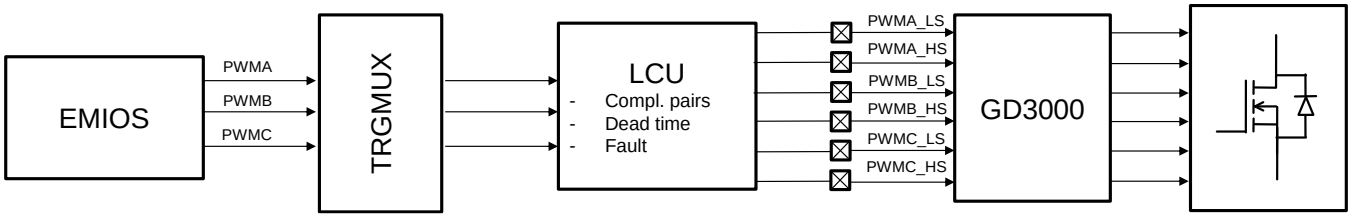

Figure 25. **PWM signal forming**

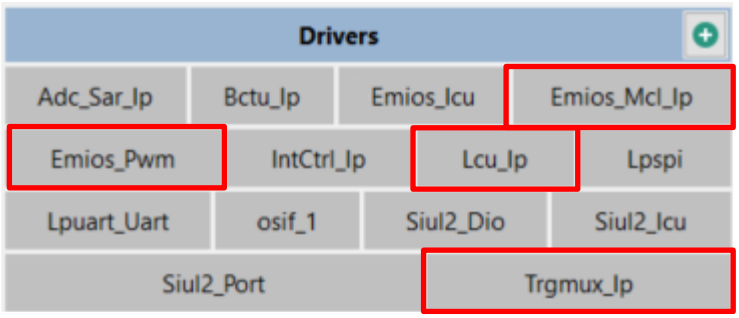

Figure 26. **Drivers for PWM generation**

4.2.3.1. eMIOS

eMIOS CH0 is configured as a time base for PWM signals. This channel can create local time base for CH 1-7. Chanel operates in a Modulus Counter Buffered (MCB) mode where there is just up counting. When the internal counter matches a value defined by field period (channel register A of the eMIOS channel) and a clock tick occurs, the internal counter is reset to 1 and reload is generated. Considering 160MHz and bus prescalers DIV_1, the "*Default period"* 8000 ticks means 50µs/20 kHz. "*Offset at*

start" gives the opportunity to initialize counter value before the counting is started what allows to configure delay between multiple synchronized time-bases.

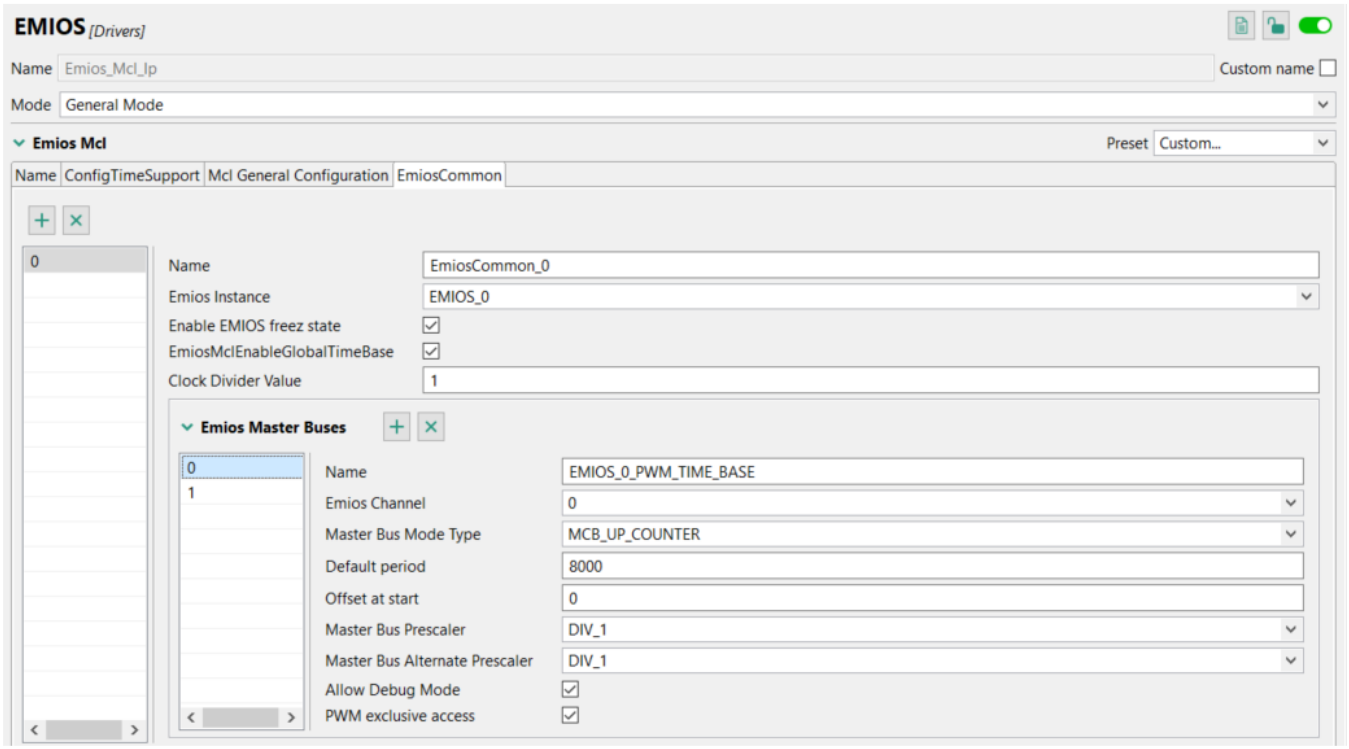

Figure 27. **PWM time base configuration**

eMIOS CH 1-3 are configured to generate the PWM signal for motor phases PHA-C. Channels operates in Output PWM Buffered (OPWMB) mode. This is the most flexible eMIOS PWM mode, which offers independent setting of both PWM signal edges (by channels register A and B) and can form the most common types of PWM signal. Channels select local timebase BCDE as a counter bus and timebase settings are also referenced through "*PwmEmiosBusRef"* field. Channel is able to see timebase counter value through the BCDE bus and compare it with its registers A and B. "*Polarity"* defines output state on specific compare. Complete timing diagram can be found in *Figure 12*. Driver offers an abstraction where "*duty cycle"* is an active pulse (space between compare A and B) and "*Phase shift"* defines placement of this active pulse within the PWM period. Proper values for register A and B are calculated by driver. Init values of the "*Phase shift"* and "*duty cycle"* are set in Peripherals tool. Settings are applied by calling Emios_Pwm_Ip_InitChannel() and Emios_Mcl_Ip_Init() where after calling the Emios_Mcl_Ip_Init() time base counting is started. PWM signal is modified during the runtime by disabling PWM update, updating the dutycycle and the phase shift and enabling the update. Registers A and B are double buffered in OPWMB mode so new values of registers A and B are propagated on nearest reload generated by time base.

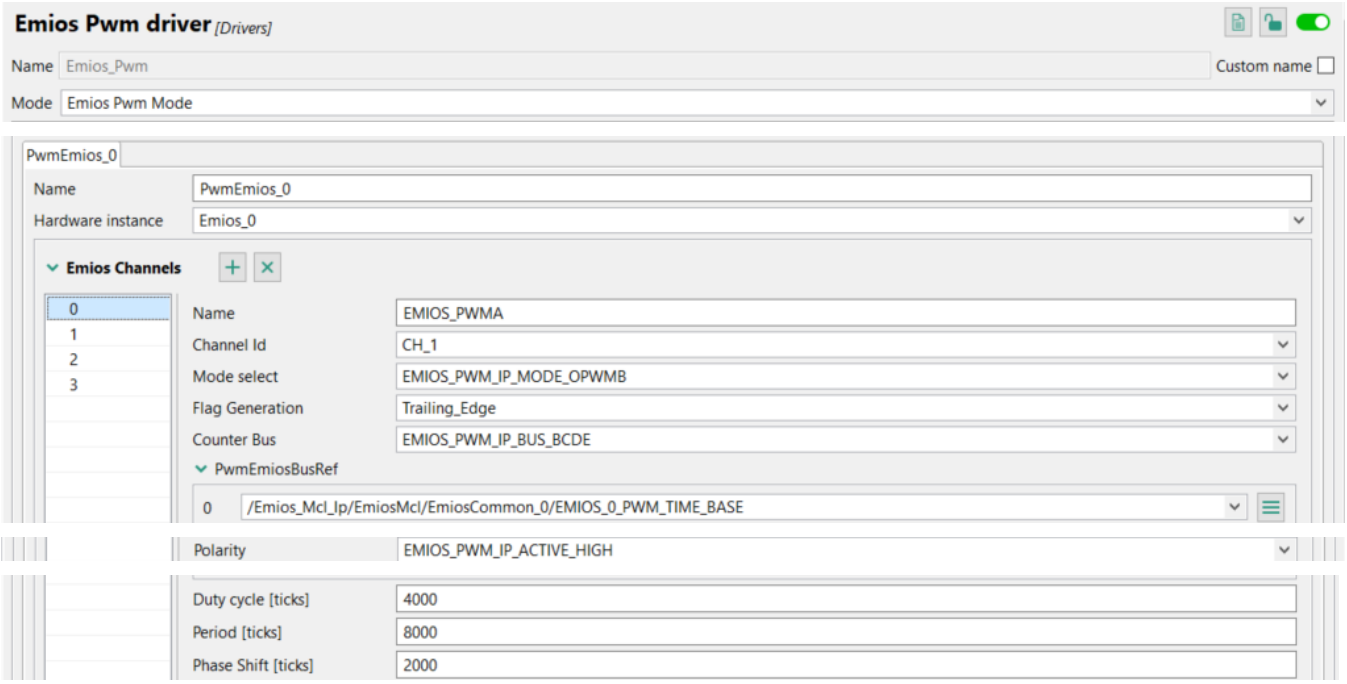

Figure 28. **PWM channel configuration**

Example 5. eMIOS API for PWM

```
void main (void)
{
...

    /*******************************************************************************************
    * eMios Driver
    *******************************************************************************************/
    Emios_Pwm_Ip_InitChannel(0U, &Emios_Pwm_Ip_BOARD_InitPeripherals_I0_Ch1);
    Emios_Pwm_Ip_InitChannel(0U, &Emios_Pwm_Ip_BOARD_InitPeripherals_I0_Ch2);
    Emios_Pwm_Ip_InitChannel(0U, &Emios_Pwm_Ip_BOARD_InitPeripherals_I0_Ch3);
...
    /*Enable eMIOS clock at last to ensure the correct trigger order*/
    Emios_Mcl_Ip_Init(0U, &Emios_Mcl_Ip_0_Config_BOARD_INITPERIPHERALS);
...

}

tBool ACTUATE_SetDutycycle(SWLIBS_3Syst_FLT *fltpwm)
{
...

    Emios_Pwm_Ip_ComparatorTransferDisable(0U,(uint32_t)0b1110U);
...
    Emios_Pwm_Ip_SetPhaseShift(0U, 1U, pwmShiftA);
    Emios_Pwm_Ip_SetDutyCycle(0U, 1U, pwmDutyA);
    Emios_Pwm_Ip_SetPhaseShift(0U, 2U, pwmShiftB);
    Emios_Pwm_Ip_SetDutyCycle(0U, 2U, pwmDutyB);
    Emios_Pwm_Ip_SetPhaseShift(0U, 3U, pwmShiftC);
    Emios_Pwm_Ip_SetDutyCycle(0U, 3U, pwmDutyC);
    Emios_Pwm_Ip_ComparatorTransferEnable(0U,(uint32_t)0b1110U);
}
```

4.2.3.2. TRIGGER MUX

TRGMUX ensures a connection between eMIOS and LCU. Settings within the "*Hardware group"* TRGMUX_IP_LCU0_0 connects outputs of eMIOS0 channels 1-3 to LCU0 inputs 0-2. Setting is applied by calling Trgmux_Ip_Init() function.

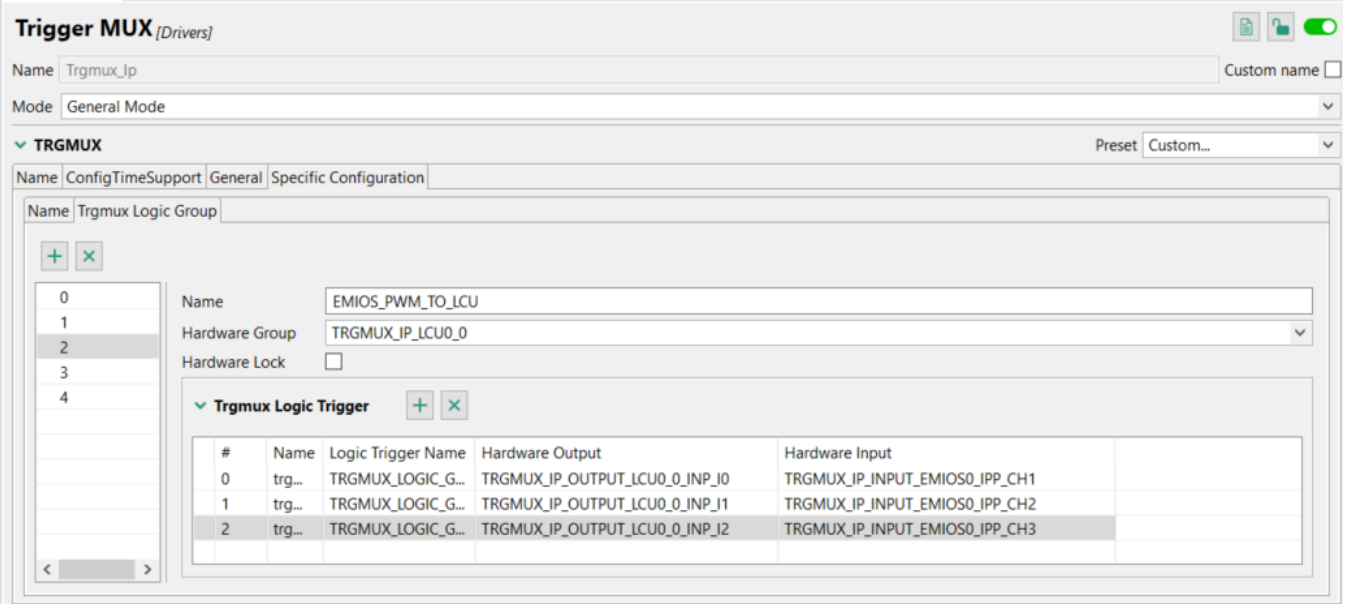

Figure 29. **TRGMUX settings for PWM signals**

4.2.3.3. LCU

Logic control unit (LCU) is a peripheral for a real time control, which offers a programmable logic function to create output waveforms or to process digital signals. LCU contains three Logic cells (LC) embedded each with four inputs and outputs with configurable true table for each output and more other features like digital filters, force inputs, sync inputs, SW override logic. In order to generate the PWM complementary signal following functionality is needed: Input multiplexing, Look Up Table (LUT), Digital filters, output polarity settings as is shown in *Figure 30*. Full featured LCU diagram can be found in S32K3xx Reference Manual [7] . Lcu_Ip driver is used to configure and to control LCU. In this example LCU0 instance is selected to generate PWM complementary pairs. LC0 generate signals for phases A and B and LC1 generates signals for Phase C. First configuration relates to inputs multiplexing. Configurations 0-2 in *"Lcu Logic Input"* tab create a connection between LCU instance inputs and LC inputs. Multiplexor inputs 0,1(eMIOS0 CH 1,2) are connected to LC0 input 0,1 and multiplexor input 2 (eMIOS0 Ch3) is connected to LC1 input 0. Output configuration for complementary pairs is in a tab *"Lcu Logic Output"* configurations 0-5. The first important thing to configure is an output polarity. High side inputs of GD3000 have inverted polarity so also related outputs have inverted polarity. It ensures that during the time when LCU outputs are disabled all MOSFETs are in an inactive state (also different strategies like for example all bottom MOSFETSs on can be used by changing the output polarity settings). Next setting is Look-up Table (LUT) for every output. LUT defines output state of the LUT Block for every combination of four inputs (combination 0000 is least significant bit of the LUT register). For example O0 mirrors I0 and O1 (as complementary channel) negates I0 as is shown in *Table 3*. Last thing to configure is a dead time. It is generated using digital filters where rising edges of the LUT block output are delayed. Complete waveform composition

of complementary channels can be seen in the *Figure 30*. GD3000 is able to automatically turn off MOSFETs when fault occurs. In case of simpler drivers or external fault logic, LCU offers asynchronous Force logic which can automatically disable LCU outputs on external pin event. For more details about this feature see S32K3xx Reference Manual [7] . In order to configure and control LCU, a Lcu_Ip RTD driver is used. Settings are applied by calling Lcu_Ip_Init() function and outputs can be enabled/disabled by calling Lcu_Ip_SetSyncOutputEnable().

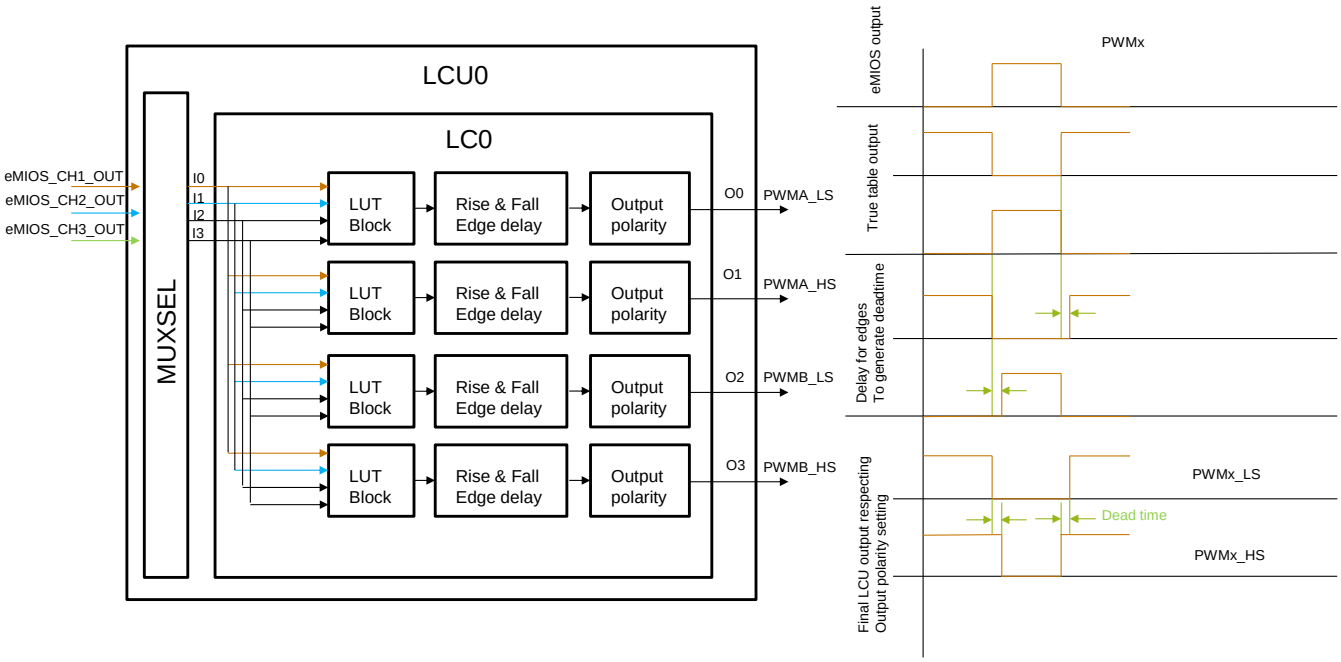

Figure 30. **Simplified LCU features block diagram for PWM**

Table 3. **LUT configurations for LCU0 LC0**

LC0_I3	LC0_I2	LC0_I1	LC0_I0		LC0_O0	LC0_O1	LC0_O2	LC0_O3
x	x	PWM_PHB	PWM_PHA		PWMA_LS	PWMA_HS	PWMB_LS	PWMB_HS
0	0	0	0		1	0	1	0
0	0	0	1		0	1	1	0
0	0	1	0		1	0	0	1
0	0	1	1		0	1	0	1
0	1	0	0		1	0	1	0
0	1	0	1		0	1	1	0
0	1	1	0		1	0	0	1
0	1	1	1		0	1	0	1
1	0	0	0		1	0	1	0
1	0	0	1		0	1	1	0
1	0	1	0		1	0	0	1
1	0	1	1		0	1	0	1
1	1	0	0		1	0	1	0
1	1	0	1		0	1	1	0
1	1	1	0		1	0	0	1
1	1	1	1		0	1	0	1
LUT					**0x5555**	**0xAAAA**	**0x3333**	**0xCCCC**

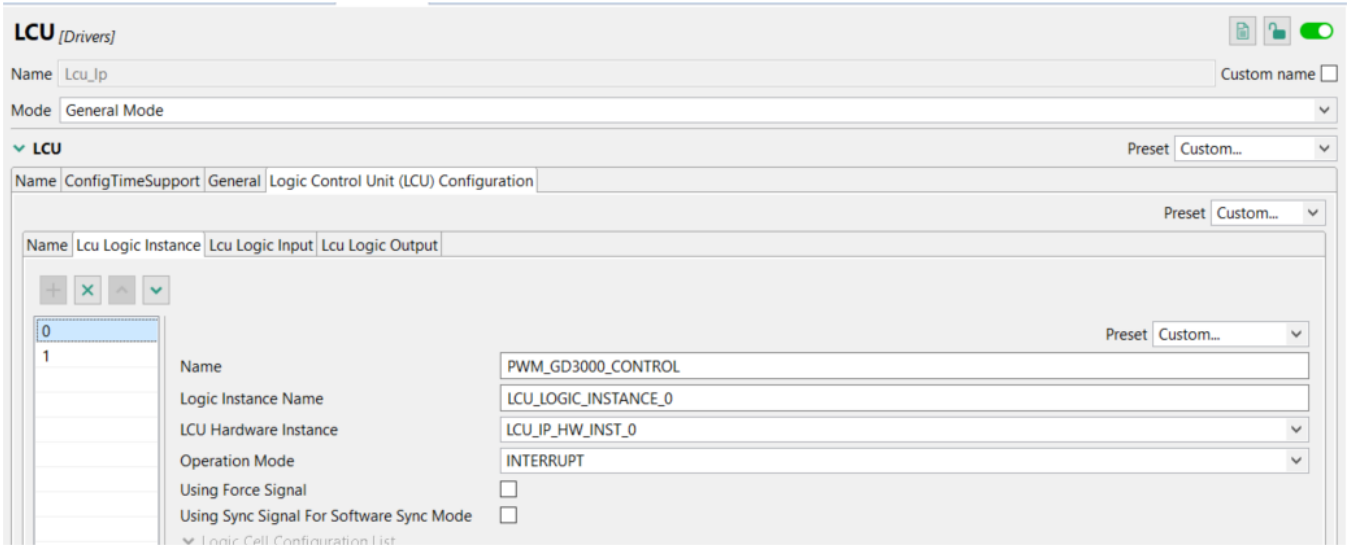

Figure 31. **LCU instance configuration for PWM**

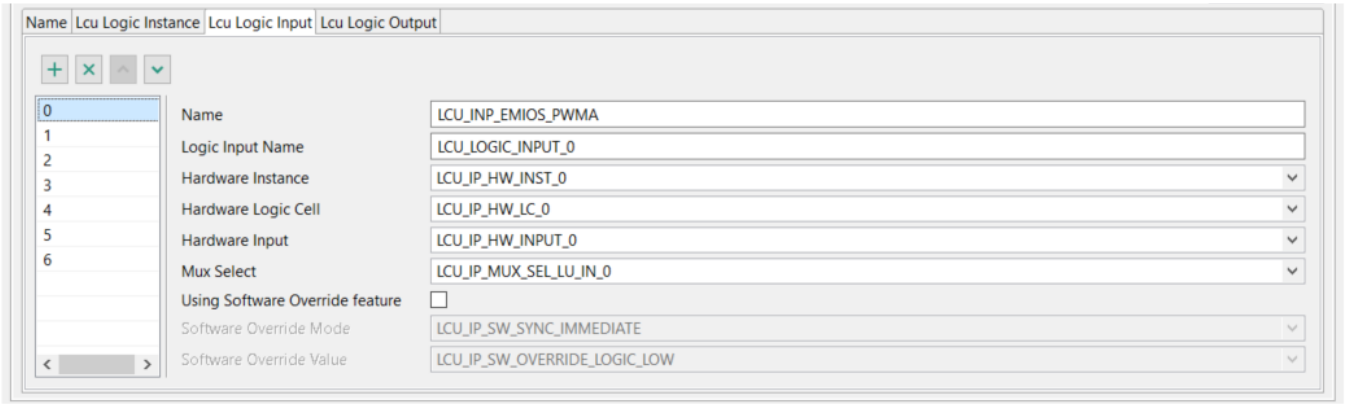

Figure 32. **LC inputs configuration for PWM**

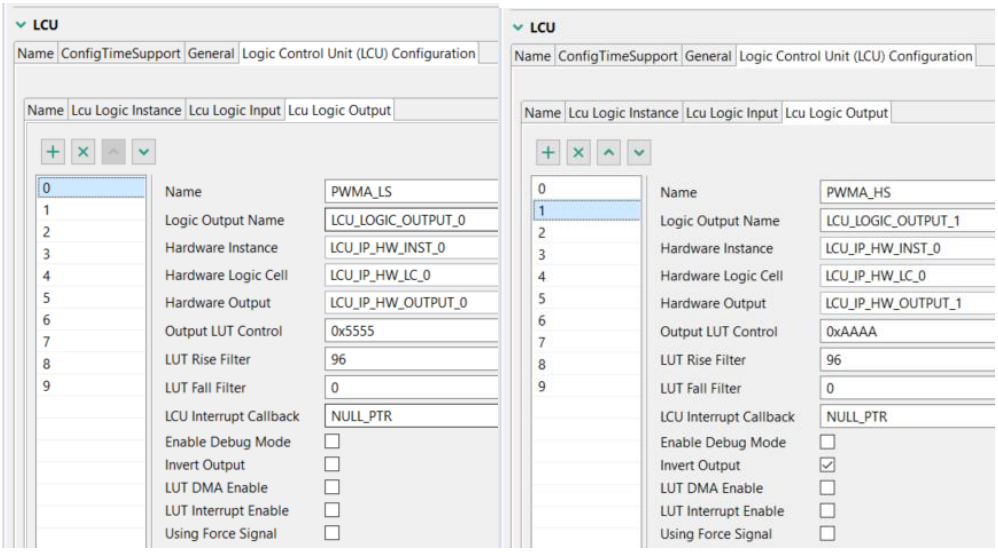

Figure 33. **LC outputs for PWM**

4.2.4. Analogue data capturing

Motor control analogue feedback capturing is realized by ADC0, ADC1, BCTU and eMIOS peripherals. BCTU controls parallel conversion of ADC0 and ADC1. eMIOS defines the trigger point when the conversion should start. ADC2 was reserved for a MCU temperature measurement and is not controlled by BCTU in order to demonstrate a non-real time measurement in parallel to real time control. In order to configure and to control those peripherals, following RTD sw drivers are used: Adc_Sar_Ip, Bctu_Ip, Emios_Mcl_Ip, Emios_Pwm.

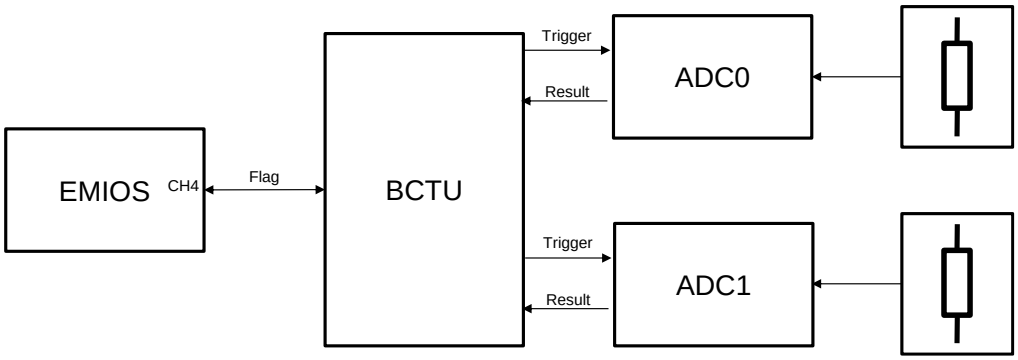

Figure 34. **MC analog feedback capturing**

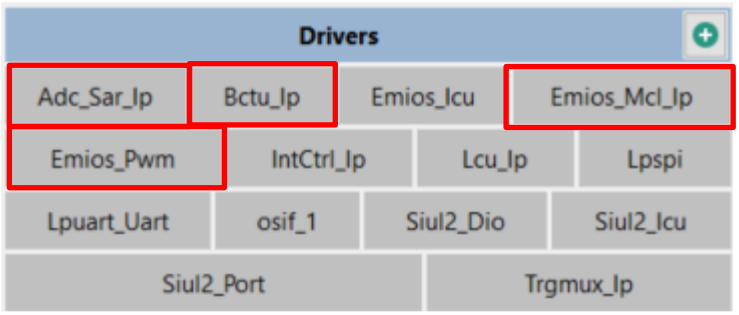

Figure 35. **Drivers for analogue feedback capturing**

4.2.4.1. ADC

The S32K344 device has three Analog-to-Digital Converters (ADCs) with the SAR algorithm. The ADC channels are divided into three groups - Precision, Standard and External (each allows independent configuration settings and different accuracy/performance level). Each channel has selectable resolution (8-, 10-, 12-, 14-bit). Conversion can be started by Normal conversion trigger, Injected conversion trigger or BCTU conversion trigger. There is also special mode , BCTU control mode, where it is explicitly set that only the BCTU can start a conversion of ADC instance. All other trigger sources are ignored. This mode is used for MC measurement ADC0 and ADC1 whereas ADC2 executes normal conversion invoked by SW. The most important setting can be seen in the Peripherals tools settings. Settings are applied calling Adc_Sar_Ip_Init() function and after the configuration ADCs are calibrated by Adc_Sar_Ip_DoCalibration(). Temperature measurement is invoked by Adc_Sar_Ip_TempSenseGetTemp() function as a non-real time control background task.

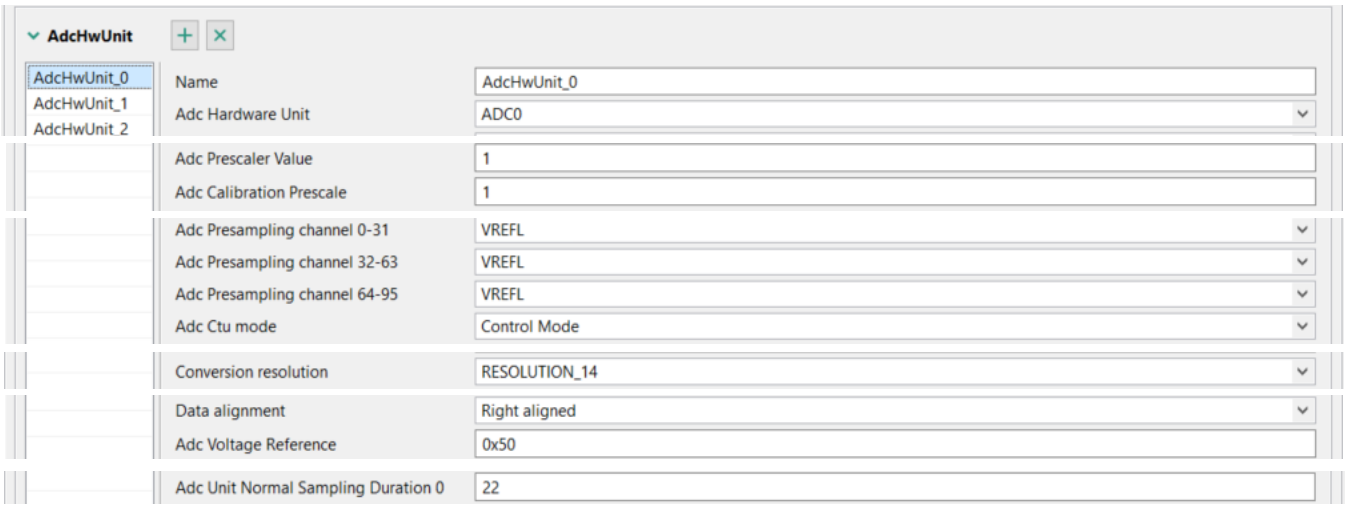

Figure 36. **ADC configuration for MC measurements**

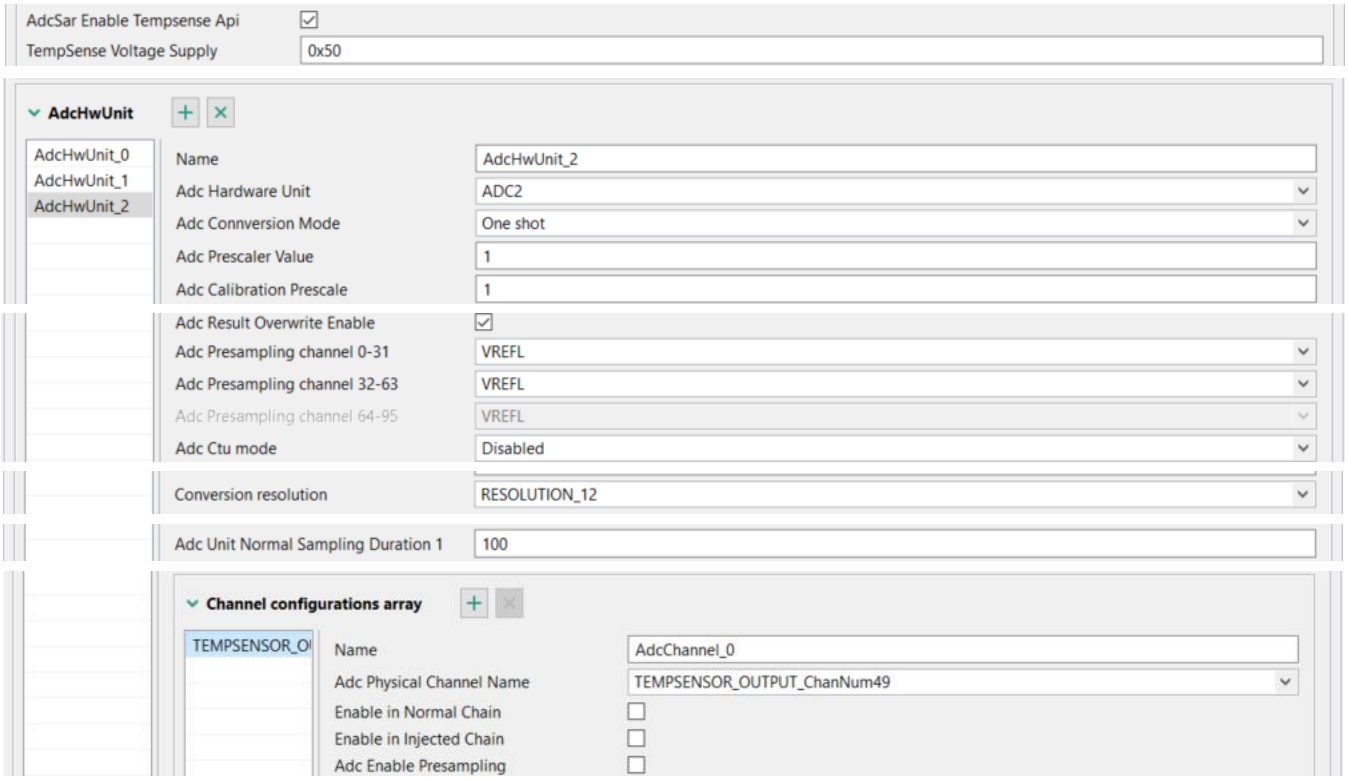

Figure 37. **ADC configuration for temperature measurements**

Example 6. ADC API

```
void main (void)
{
...

    /*******************************************************************************************
    * ADC Driver
    *******************************************************************************************/
    do {
        status = (StatusType)Adc_Sar_Ip_Init(0U, &AdcHwUnit_0_BOARD_INITPERIPHERALS);
    } while (status != E_OK);
```

```
    do {
        status = (StatusType)Adc_Sar_Ip_Init(1U, &AdcHwUnit_1_BOARD_INITPERIPHERALS);
    } while (status != E_OK);

    do {
        status = (StatusType)Adc_Sar_Ip_Init(2U, &AdcHwUnit_2_BOARD_INITPERIPHERALS);
    } while (status != E_OK);

    do {
        status = (StatusType)Adc_Sar_Ip_DoCalibration(0U);
    } while (status != E_OK);

    do {
        status = (StatusType)Adc_Sar_Ip_DoCalibration(1U);
    } while (status != E_OK);

    do {
        status = (StatusType)Adc_Sar_Ip_DoCalibration(2U);
    } while (status != E_OK);

    /* TempSenseEnable */
    Adc_Sar_Ip_TempSenseEnable(0U);
...
    /* MCU chip temperature detection. */
    TemperatureGetStatus = Adc_Sar_Ip_TempSenseGetTemp(2U, 0U, &TemperatureRaw);
...

}
```

4.2.4.2. BCTU

S32K344 has single instance of a BCTU. The BCTU accepts ADC conversion-request trigger inputs and routes those requests to one or more ADCs. There are 72 trigger inputs. 69 inputs are coming from eMIOS channels (connection is realized through channels flag) and three from TRGMUX (TRGMUX output is routed to BCTU). All triggers can be also invoked by a software instead of the HW source. Every trigger can be configured to invoke single conversion or predefined list of conversions. Conversion result can be stored into BCTU data register (there is one register per ADC instance), one of the BCTU FIFOs or into a memory buffer by DMA transfer. Conversion results remain also in the result register of ADC channel.

In this example eMIOS0 CH4 is selected as a trigger for MC analogue quantities measurement. FIFO1 is selected for *“Data Destination”*. The trigger is configured as a list of parallel conversions ADC0, ADC1 in *“Adc Target Mask”*. List of ADC channels is defined in *“BCTU List Items”* while order is given by the *“Adc Target Mask”*: BctuListItems_0 is ADC0, BctuListItems_1 is ADC1 etc. Watermark of the FIFO1 *“Watermark Value”* is set on 3 and *“Interrupt Notification”* is enabled. When the trigger comes, parallel conversion of the first two list items starts (phase currents) and once conversion has been completed, next channel couple takes a place (DC bus voltage and dummy measurement). Once all results has been stored into the FIFO, an interrupt is raised and handled by BCTU RTD interrupt handler and custom notification function Bctu_FIFO1_WatermarkNotification is called. Conversion result (Data and additional information about conversion like trigger number, ADC channel and ADC instance) are read from the FOFO using Bctu_Ip_GetFifoResult() function. Settings are applied by calling Bctu_Ip_Init() function. After enabling the notification function and BCTU global trigger, BCTU is active.

Table 4. **Possible variations of ADC target mask**

ADC target mask			Defines the BCTU ADC command list operating mode	
			LIST	SINGLE
0	0	1	of single conversions ADC0	conversion ADC0
0	1	0	of single conversions ADC1	conversion ADC1
1	0	0	of single conversions ADC2	conversion ADC2
0	1	1	of parallel conversions ADC0, ADC1	X
1	1	0	of parallel conversions ADC1, ADC2	X
1	0	1	of parallel conversions ADC0, ADC2	X
1	1	1	of parallel conversions ADC0, ADC2, ADC3	X

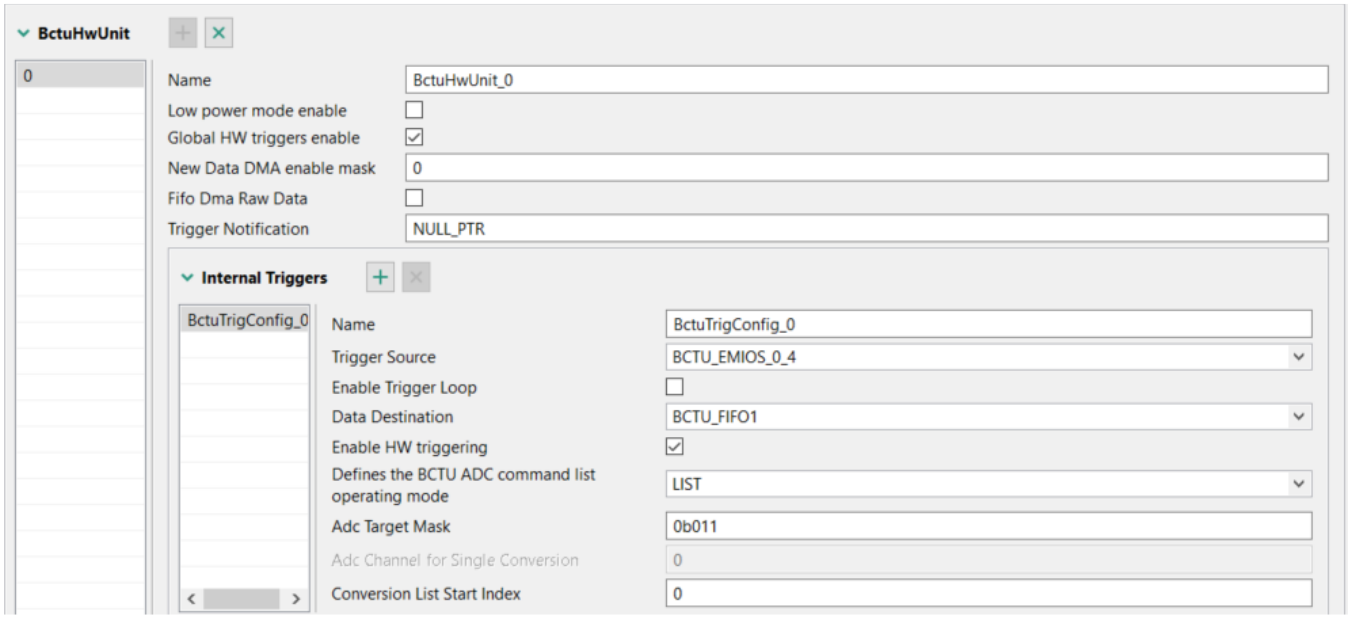

Figure 38. **BCTU Trigger configuration**

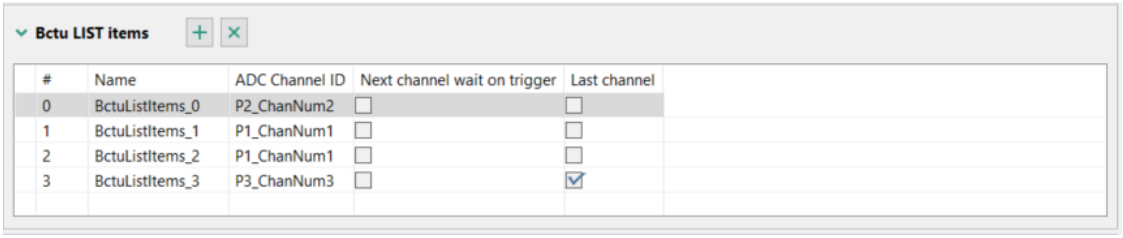

Figure 39. **BCTU list configuration**

NOTE

BctuListItems_3 (P3_ChanNum3) is dummy measurement since the list is list of two parallel measurements and only 3 motor control quantities are measured (current phase A , current phase B, DC bus voltage)

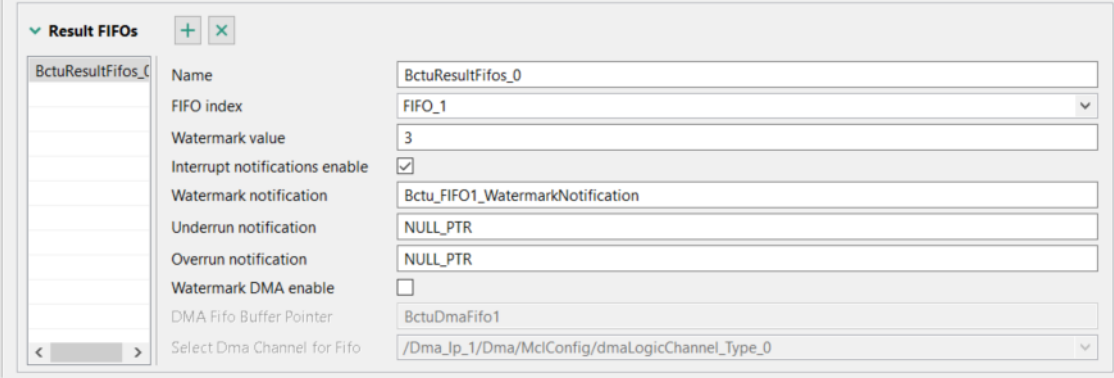

Figure 40. **BCTU FIFO configuration**

Example 7. BCTU API

```
void main (void)
{
...

    /*******************************************************************************************************
    * BCTU Driver
    *******************************************************************************************************/

    Bctu_Ip_Init(0U, &BctuHwUnit_0_BOARD_INITPERIPHERALS);
    Bctu_Ip_EnableNotifications(0U, BCTU_IP_NOTIF_LIST);
    Bctu_Ip_SetGlobalTriggerEn(0U, TRUE);
...

}

void Bctu_FIFO1_WatermarkNotification (void)
{
...
    mCount = 0;
    while (Bctu_Ip_GetFifoCount(0U, 0U))
    {
        Bctu_Ip_GetFifoResult(0U, 0U, &measuredValues[mCount++]);
    }
...
```

4.2.4.3. eMIOS

eMios channel 4 is configured to generate the trigger for BCTU in precise moment. Same modes and drivers are used as in the use case of PWM generation in chapter *eMIOS*. Trigger channel uses a global time base A (CH 23). This trigger time base is synchronized with PWM time base with no delay. Considering 160 MHz, a period 16000 tick means 100 µs so the sampling frequency of motor quantities is 10 kHz. An important setting for CH4 is *"Flag generation"*. Trialing_Edge means generating flag(what is the event when BCTU is triggered) on compare with register B. With a used *"Polarity"* it is

in the falling edge of the CH4 output, which can be visualized on the pin using TRGMUX. Trigger is generated five cycles after reload (PWM time base reload and Trigger time base reload are overlapping since there is no delay). In this example the trigger moment is not changing during the runtime, but it possible to change trigger moment in same way like update of PWM channels. Settings are applied by calling Emios_Pwm_Ip_InitChannel() and Emios_Mcl_Ip_Init functions().

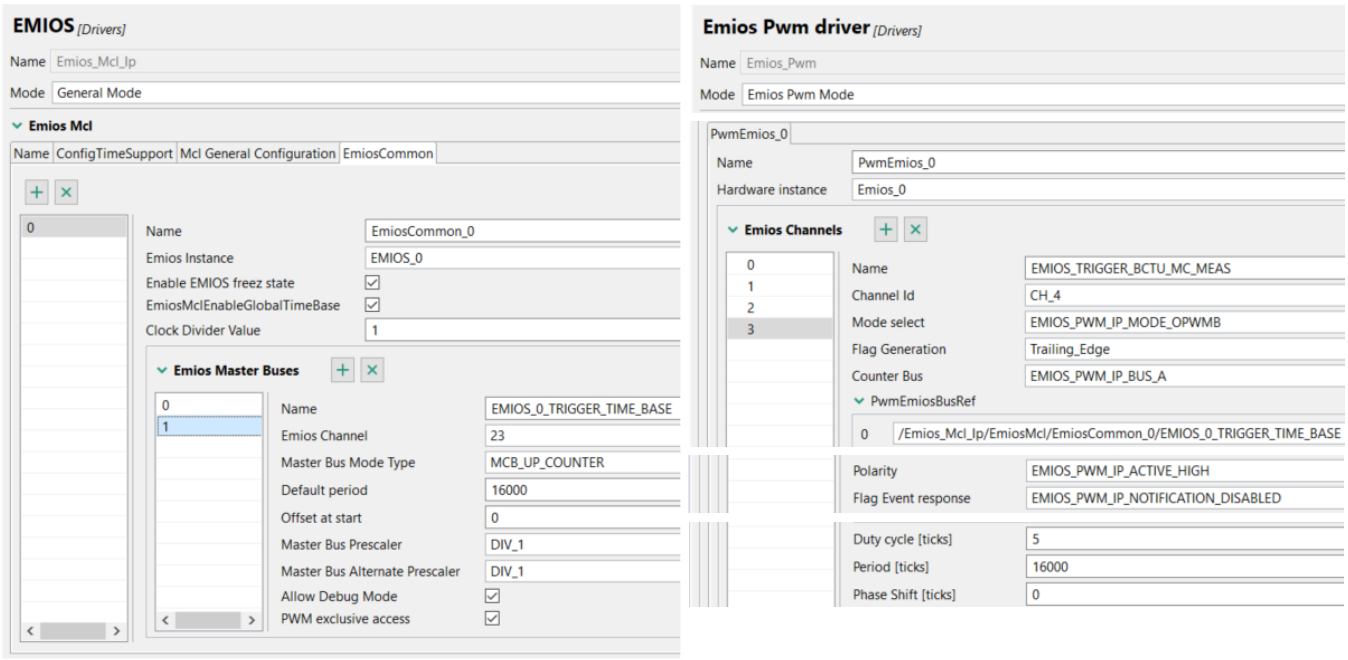

Figure 41. **eMIOS trigger configuration**

Example 8. eMIOS API for PWM

```
void main (void)
{
...

    /*******************************************************************************************
    * eMios Driver
    *******************************************************************************************/
...
    Emios_Pwm_Ip_InitChannel(0U, &Emios_Pwm_Ip_BOARD_InitPeripherals_I0_Ch4);
...
    /*Enable eMIOS clock at last to ensure the correct trigger order*
    Emios_Mcl_Ip_Init(0U, &Emios_Mcl_Ip_0_Config_BOARD_INITPERIPHERALS);
...

}
```

4.2.5. Quadrature decoder

Quadrature decoder feature is achieved by cooperation of eMIOS, TRGMUX and LCU modules. This feature is used to decode the quadrature signals generated by rotary sensors used in motor control domain. This mode is used to process encoder signals and determine rotor position and speed.

There are three output signals generated by incremental encoder as shown in *Figure 42*. Phase A and Phase B signals consist of a series of pulses which are phase-shifted by 90° (therefore the term

"quadrature" is used). The third signal (called "Index") provides the absolute position information. In the motion control, it is used to check the pulse-counting consistency. Index signal is not used in this example hence position offset is calibrated during the rotor alignment.

In order to get the rotor position encoder signals PHA and PHB are brought to the LCU. LCU preprocess them and generates pulses based on rotor speed direction. eMIOS acts as a counter to get the rotor absolute position. An angle tracking observer (ATO) is used to calculate the final rotor speed and position. Emios_Icu, Lcu_Ip and Trgmux_Ip drivers are used to control and to configure peripherals for this use case.

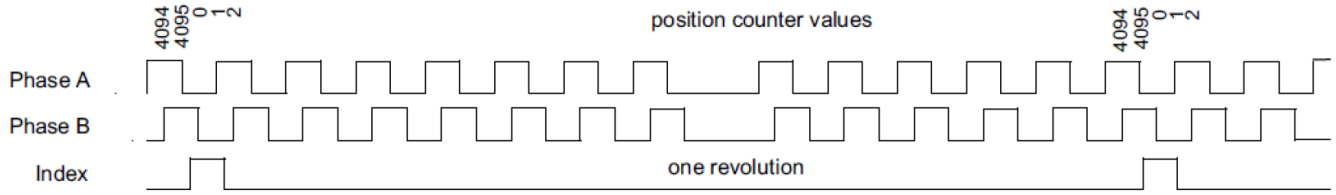

Figure 42. **Output signals of the 1024 pulses Encoder**

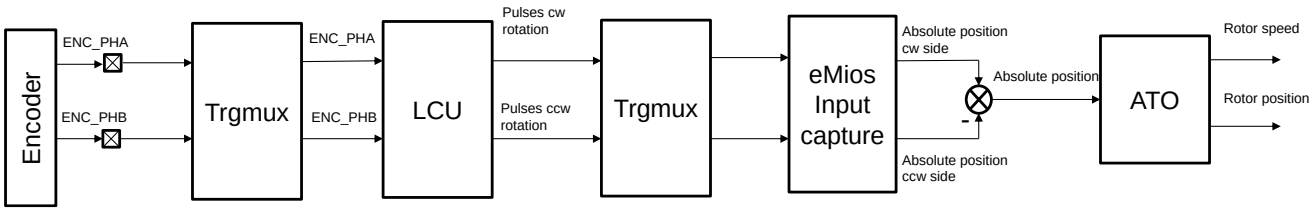

Figure 43. **Peripherals interconnection for quadrature encoder**

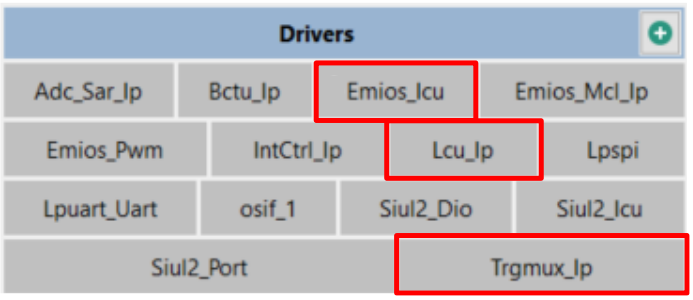

Figure 44. **Drivers for quadrature decoder feature**

NOTE

This routine is disabled by default, since PM motor of the S32K344 motor control kit is not equipped with encoder sensor. To enable encoder signal processing routine, set ENCODER macro to *true*.

4.2.5.1. TRIGGER MUX

TRGMUX ensures a connection between Input pins and LCU and between eMIOS and LCU. Settings within the *"Hardware Group"* ENCODER_PINS_TO_LCU connects PTA19(TRGMUX_IN13) and PTA20(TRGMUX_IN14) to LCU1 inputs 0-1. Settings within the *"Hardware Group"*

ENCODER_LCU_TO_EMIOS connects LCU1 LC0 outputs 2-3 to eMIOS0 inputs of channels 5-6. The setting is applied by calling Trgmux_Ip_Init() function.

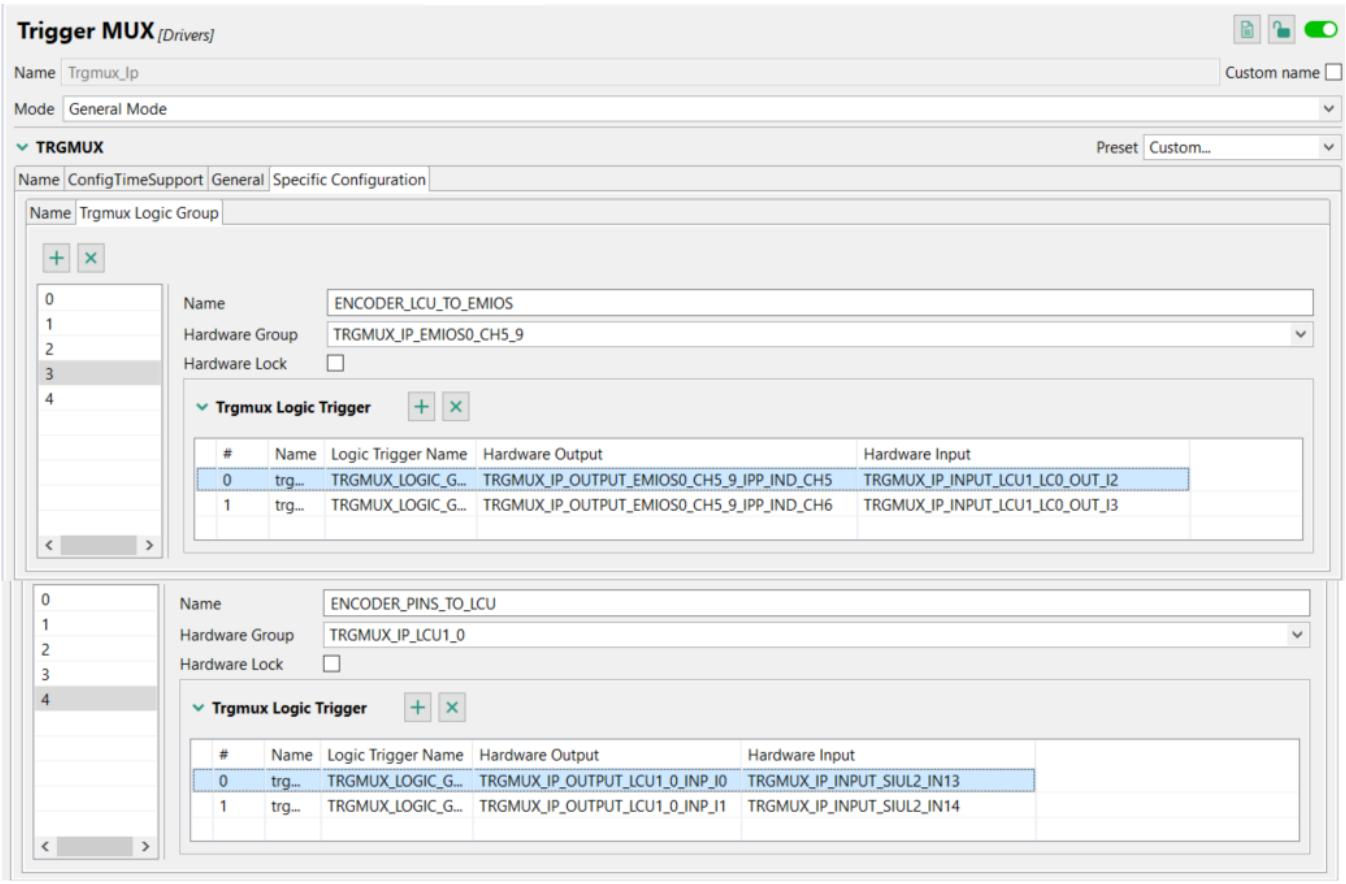

Figure 45. **TRGMUX settings for quadrature decoder**

4.2.5.2. LCU

In this example LCU1 instance is selected for preprocessing encoder signals phase A and phase B. Same LCU features are used as in chapter *LCU* but, whole preprocessing is realized in LC0. Configurations 3-6 in *"Lcu Logic Input"* tab create a connection between LCU instance inputs and LC inputs. Multiplexor inputs 0,1(pins PTA19,PTA20) are connected to LC0 input 0,1 and multiplexor feedback input 0,1 (LC0 out 0,1) is connected to LC0 input 2,3. Outputs configurations are in a tab *"Lcu Logic Output"* configurations 6-9. True tables of outputs 0,1 just mirror inputs 0,1 and rising and falling edge is delayed by digital filters. True tables of outputs 2,3 use information of all inputs. They detect edges of the encoder phases PHA and PHB using auxiliary signals PHA0 and PHB0 as is depicted in waveform *Figure 46.* Detected edge is represented by short pulse (in this example 5 ticks filters settings of outputs 0,1). Based on actual value of signals PHA and PHB, logic function in LUT distinguishes the direction of rotation and a detected edge is placed on proper output (cw or ccw). True tables of all outputs (given by LUT) can be found in *Table 5*. Filters of outputs 2 and 3 work as a glitch filters. If generated pulse is shorter than 4 ticks it will not appear on the output. It is protection against a noise on ENC_PHA and ENC_PHB pins. All settings are applied by calling Lcu_Ip_Init() function.

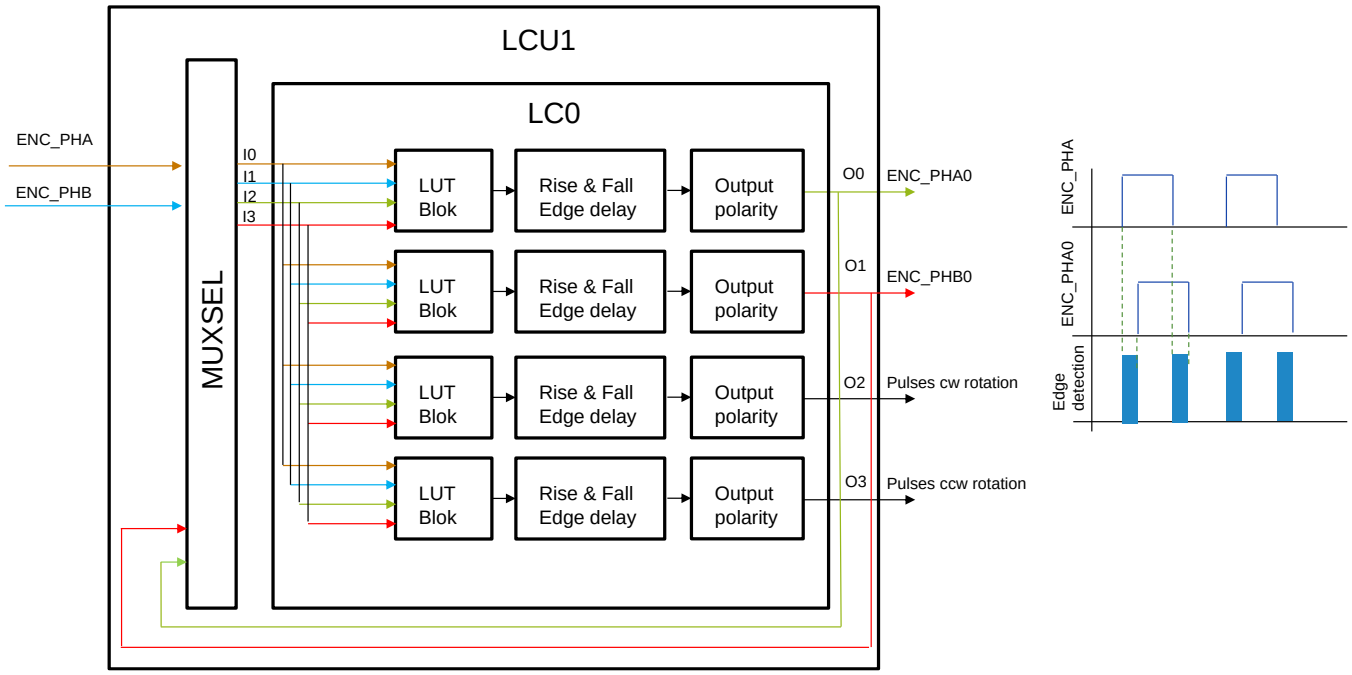

Figure 46. **Simplified LCU features block diagram for quadrature decoder**

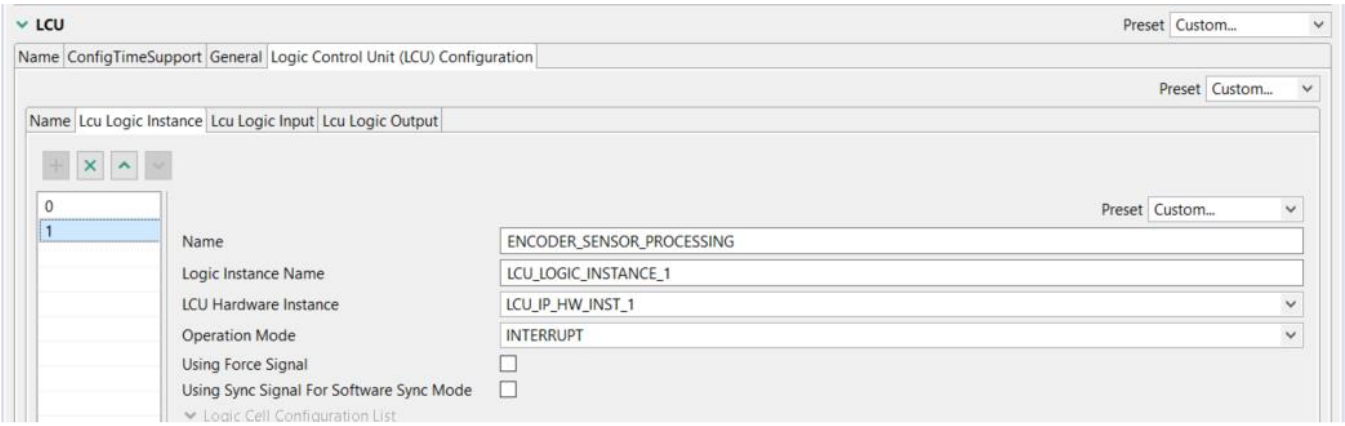

Figure 47. **LCU instance configuration for quadrature decoder**

Table 5. **LUT configurations for LCU1 LC0**

LC0_I3	LC0_I2	LC0_I1	LC0_I0		LC0_O0	LC0_O1	LC0_O2	LC0_O3
ENC_PHB0	ENC_PHA0	ENC_PHB	ENC_PHA		ENC_PHA0	ENC_PHB0	Pulses cw	Pulses ccw
0	0	0	0		0	0	0	0
0	0	0	1		1	0	1	0
0	0	1	0		0	1	0	1
0	0	1	1		1	1	0	0
0	1	0	0		0	0	0	1
0	1	0	1		1	0	0	0
0	1	1	0		0	1	0	0
0	1	1	1		1	1	1	0
1	0	0	0		0	0	1	0
1	0	0	1		1	0	0	0
1	0	1	0		0	1	0	0
1	0	1	1		1	1	0	1
1	1	0	0		0	0	0	0
1	1	0	1		1	0	0	1
1	1	1	0		0	1	1	0

LC0_I3	LC0_I2	LC0_I1	LC0_I0		LC0_O0	LC0_O1	LC0_O2	LC0_O3
1	1	1	1		1	1	0	0
LUT					**0xAAAA**	**0xCCCC**	**0x4182**	**0x2814**

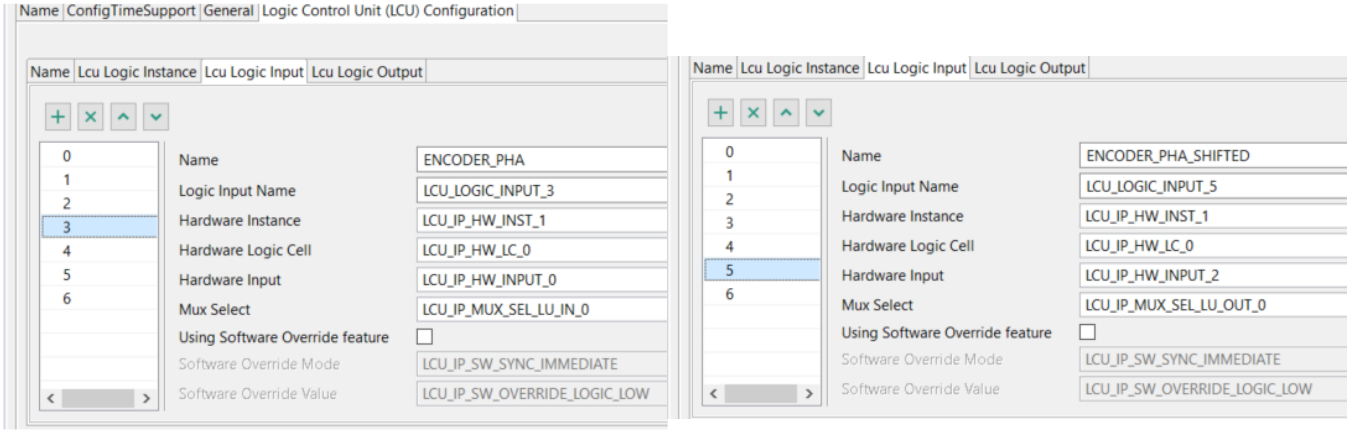

Figure 48. **LCU inputs configuration for quadrature decoder**

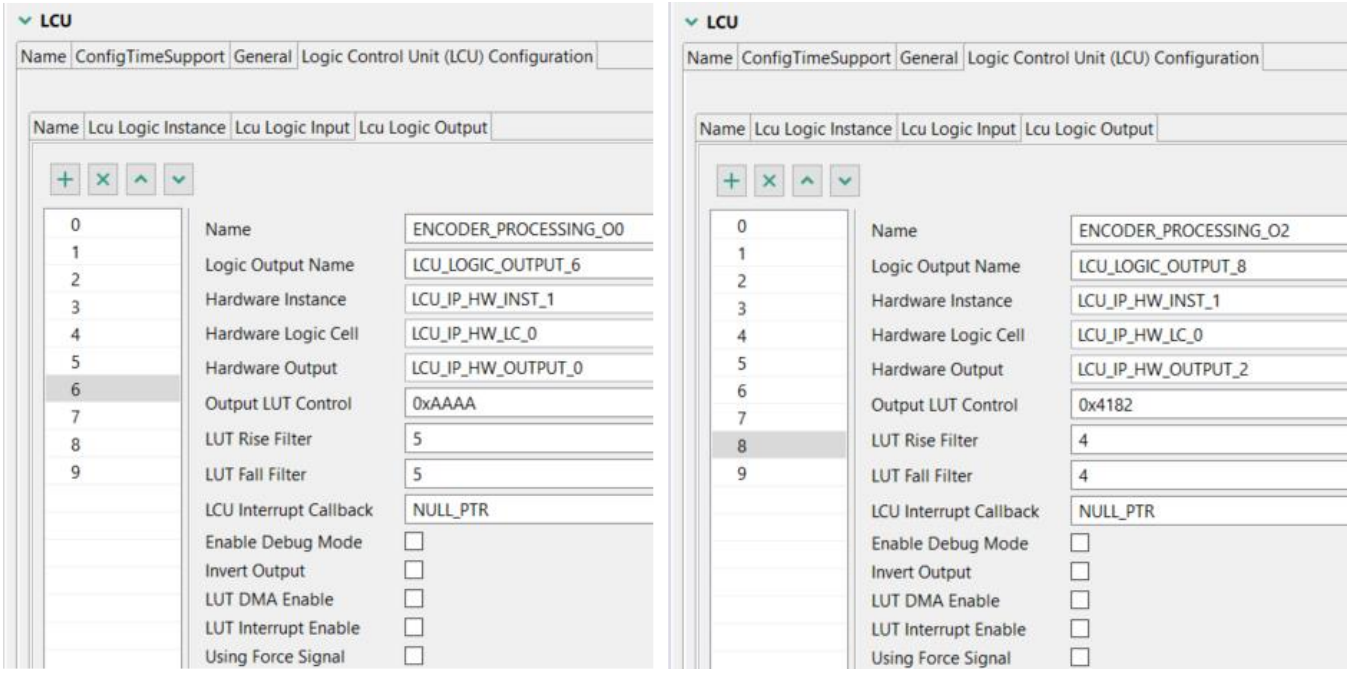

Figure 49. **LCU outputs for quadrature decoder**

4.2.5.3. eMIOS

eMIOS0 channels 5 and 6 are channels of type G so they contain their own counter and are able to count edges of the channel input signal. In the example, those channels operates in modulus counter buffered (MCB) mode and count rising edges of signals coming from LCU which represents detected edges of signals PHA and PHB of the encoder sensor. For more details about eMIOS channel types see S32K3xx Reference Manual [7] . All settings are applied by calling Emios_Icu_Ip_Init() function and by enabling edge counting using Emios_Icu_Ip_EnableEdgeCount() functions. Actual counter value is obtained by calling Icu_GetEdgeNumbers() for particular channel.

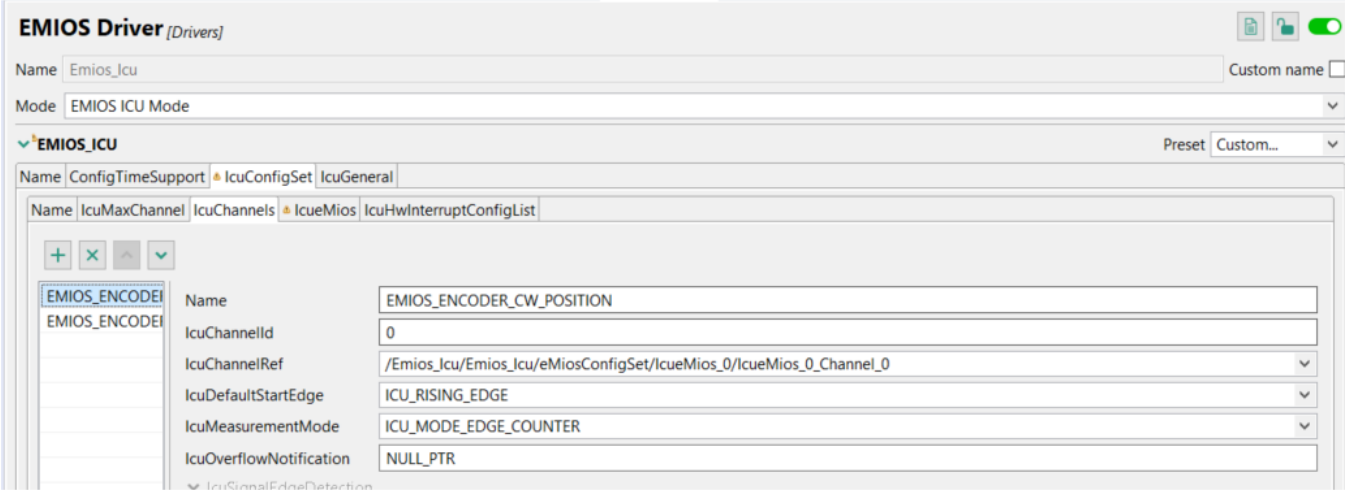

Figure 50. **General ICU configuration**

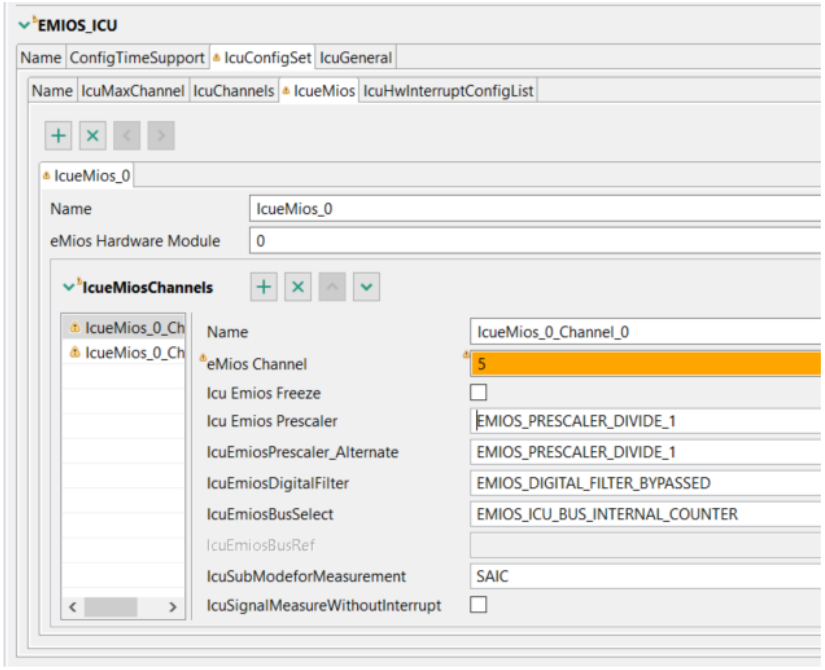

Figure 51. **eMIOS channels for input capture**

NOTE

Property IcuSubModeforMeasurement is not applicable for ICU_MODE_EDGE_COUNTER. Channels are set into MCB mode by calling Emios_Icu_Ip_EnableEdgeCount function.

Example 9. eMIOS API for quadrature decoder

```
void main (void)
{
...

    /*****************************************************************************************
    * eMios Driver
    *****************************************************************************************/
```

```
    Emios_Icu_Ip_Init(0U, &eMios_Icu_Ip_0_Config_PB_BOARD_INITPERIPHERALS);
    Emios_Icu_Ip_EnableEdgeCount(0u, 5U);
    Emios_Icu_Ip_EnableEdgeCount(0u, 6U);
...

}

tBool POSPE_GetPospeElEnc (encoderPospe_t *ptr)
{
...

    counterCW   = (uint16_t) ((Icu_GetEdgeNumbers(IcuChannel_1))- ptr->counterCwOffset);      /* CW  counter */
    counterCCW  = (uint16_t) ((Icu_GetEdgeNumbers(IcuChannel_2))- ptr->counterCcwOffset);     /* CCW counter */
...
}
```

NOTE

Various input pins or TRGMUX output can be selected for eMIOS input. This selection is realized in SIUL2 IMCR register.

4.2.6. Communication

4.2.6.1. UART

LPUART6 is used as a communication interface between S32K344 MCU and FreeMASTER run-time debugging and visualization tool. Lpuart_Uart RTD driver is used to configure LPUART. Configuration is applied by calling Lpuart_Uart_Ip_Init(). LPUART must be configure before any API of FreeMASTER embedded driver is called (functions: FMSTR_Init(), FMSTR_Poll(), FMSTR_Recorder()).

For more about FreeMASTER see [4] .

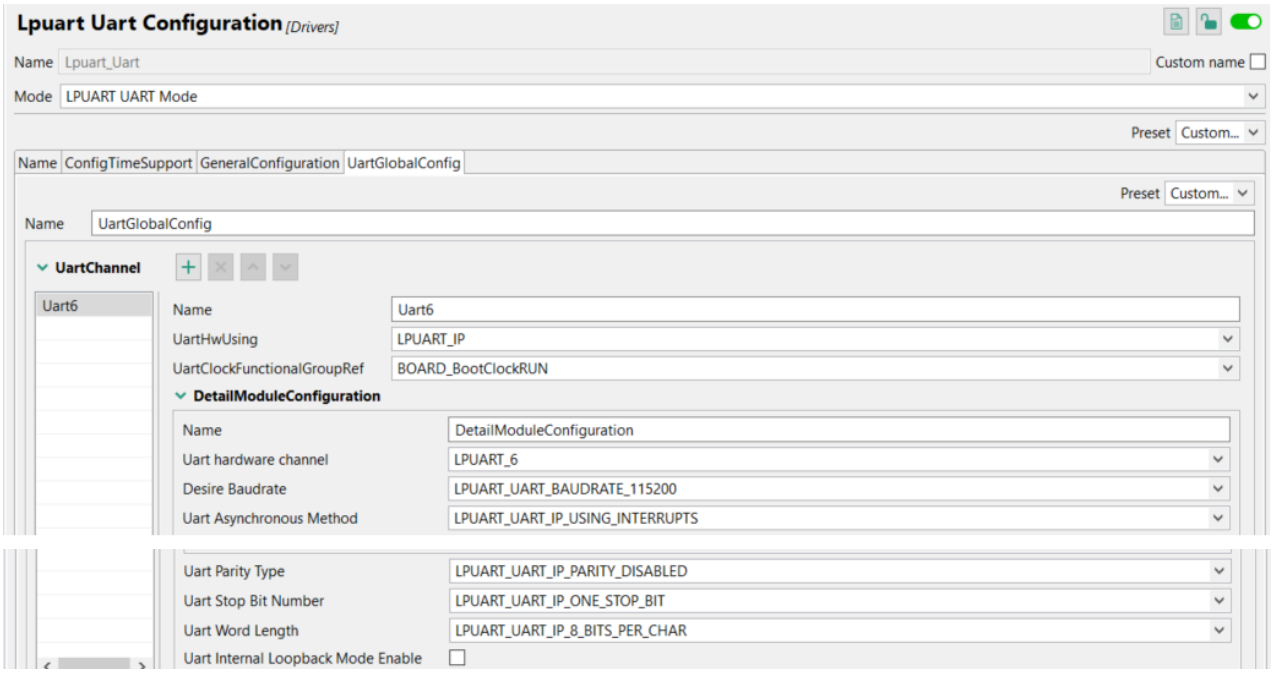

Figure 52. **LPUART configuration**

4.2.6.2. LPSPI

LPSPI1 is used as communication interface between S32K344 MCU and analog FET pre-driver GD3000. NXP's Three-Phase Brushless Motor Pre-Driver Software Driver (TPP) uses RTD LPSPI driver to establish a communication and to configure GD3000 properly. Included embedded driver provides access to all features of GD3000 FETs driver such as writing/reading status registers, dead time insertion and fault protection. SPI settings are applied by calling Lpspi_Ip_Init(). LPSPI must be initialize before the TPP driver is used (Functions: GD3000_Init(), TPP_GetStatusRegister(), TPP_ClearInterrupts()).

For more information about TPP driver see [11] .

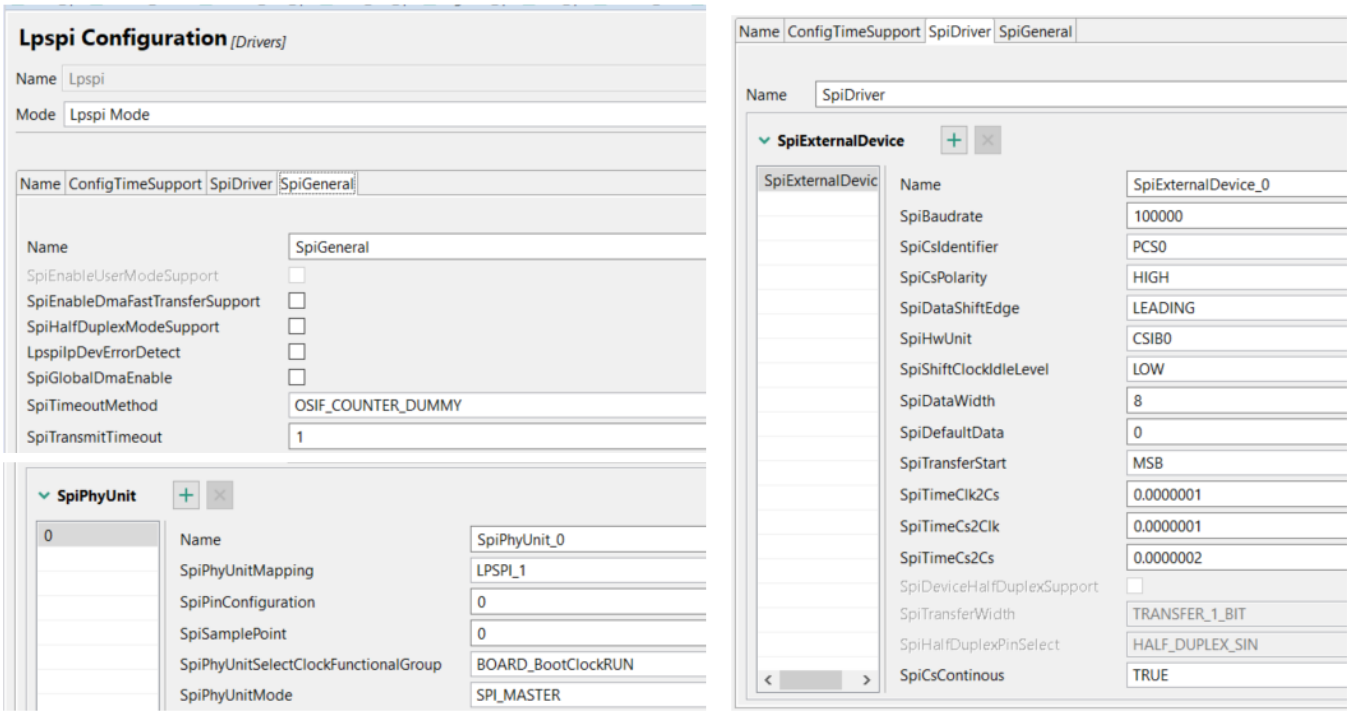

Figure 53. **SPI configuration**

4.3. Software architecture

4.3.1. Introduction

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

4.3.2. Application data flow overview

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

the application state machine shown in *Figure 56*, and application flowcharts shown in *Figure 54* and *Figure 55*.

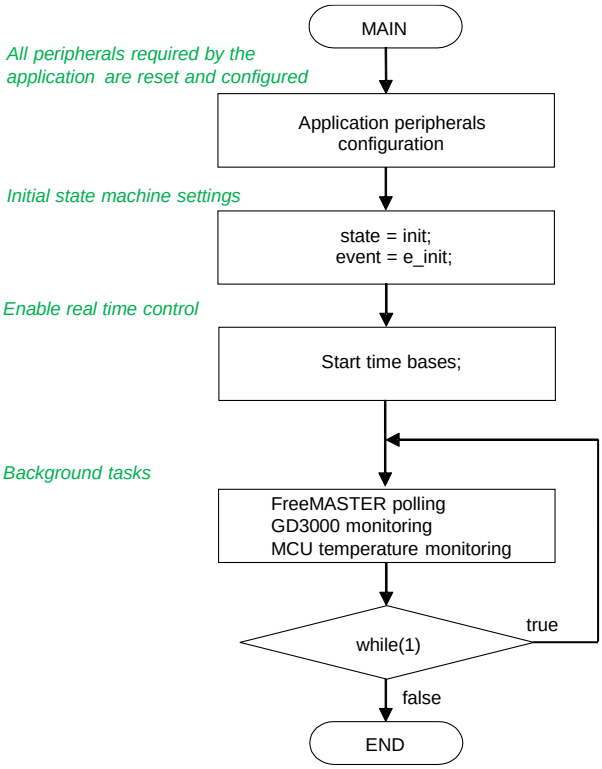

Figure 54. **Flow chart diagram of main function with background loop**

To achieve precise and deterministic sampling of analog quantities and to execute all necessary motor control calculations, the state machine functions are called within a periodic notification function. Hence, in order to actually call state machine functions, the peripheral causing this periodic interrupt must be properly configured and the interrupt enabled. As described in section *S32K344 device initialization* all peripherals are initially configured and all interrupts are enabled after reset of the device. As soon as all S32K344 peripherals are correctly configured, the state machine functions are called from the BCTU notification function. The background loop handles non-critical timing tasks, such as the FreeMASTER communication polling, GD3000 status pooling and microcontroller temperature measurement.

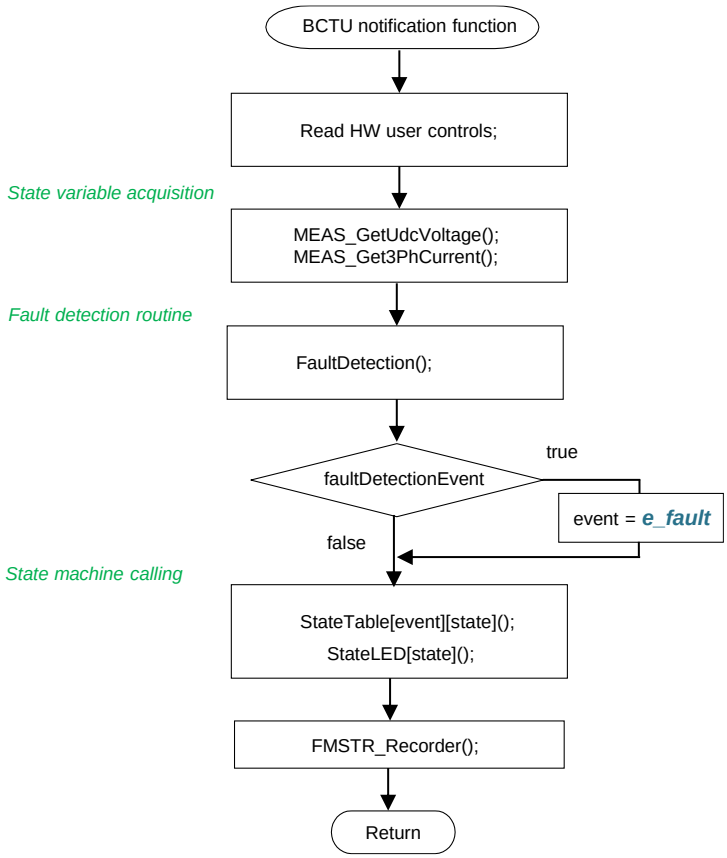

Figure 55. **Flow chart diagram of periodic interrupt notification function**

4.3.3. State machine

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

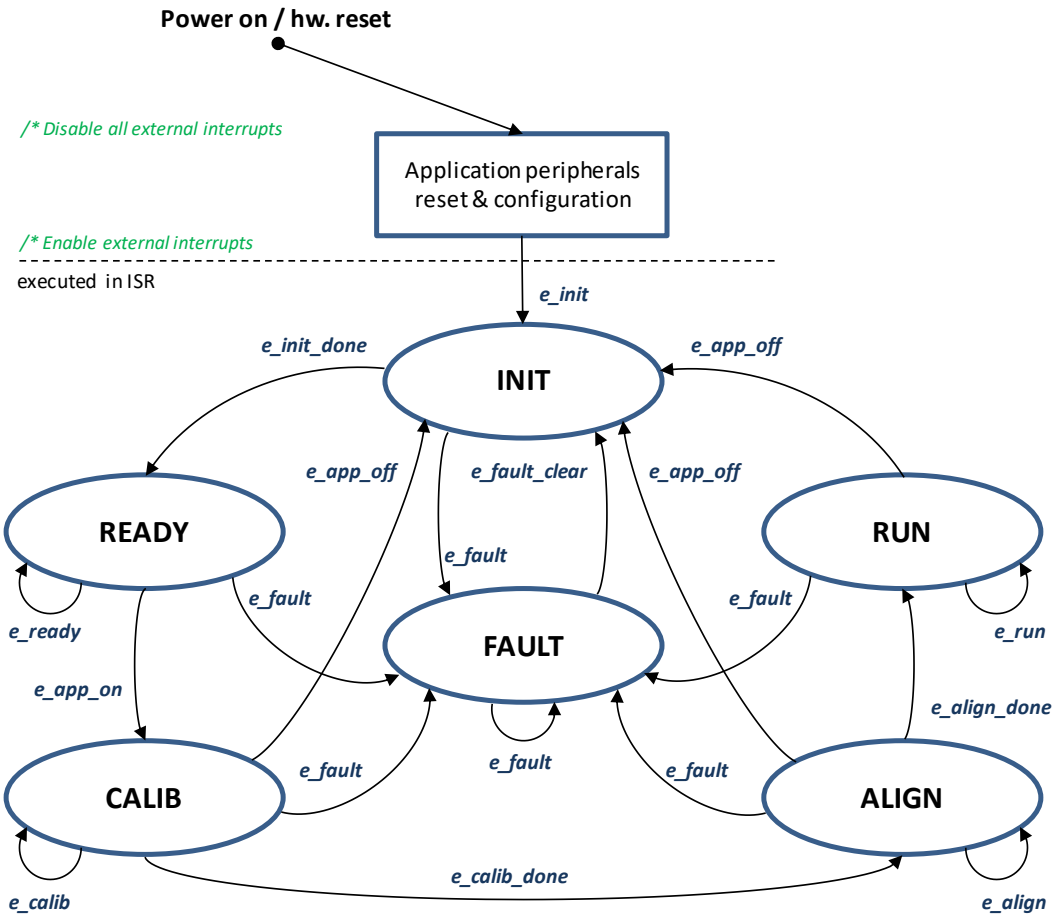

Figure 56. **Application state machine**

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

AppStates:

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

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

AppEvents:

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

4.3.3.1. State – FAULT

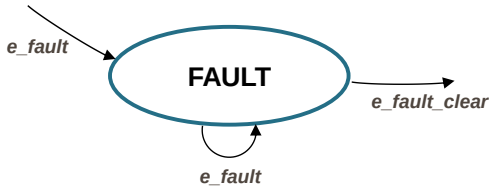

Figure 57. **FAULT state with transitions**

The application goes immediately to this state when a fault is detected. The system allows all states to pass into the FAULT state by setting *cntrState.event* = *e_fault*. State FAULT is a state that transitions back to itself if the fault is still present in the system and the user does not request clearing of fault flags. There are two different variables to signal fault occurrence in the application. The warning register *tempFaults* represents the current state of the fault pin/variable to warn the user that the system is getting close to its critical operation. And the fault register *permFaults* represents a fault flag, which is set and put the application immediately to fault state. Even if fault source disappears, the fault remains set until manually cleared by the user. Such mechanisms allow for stopping the application and analyzing the cause of failure, even if the fault was caused by a short glitch on monitored pins/variables. State FAULT can only be left when application variable *switchFaultClear* is manually set to *true* (using FreeMASTER) or by simultaneously pressing the user buttons (SW5 and SW6) on the S32K344EVB evaluation board. That is, the user has acknowledged that the fault source has been removed and the application can be restarted. When the user sets *switchFaultClear = true*; the following sequence is automatically executed, see *Example 10*.

Example 10. Fault clearing sequence

```
void StateFault(void)
{
...
    if (cntrState.usrControl.switchFaultClear)
   {
        // Clear permanent and temporary SW faults
        permFaults.mcu.R                = 0;                    // Clear mcu faults
        permFaults.motor.R              = 0;                    // Clear motor faults
        permFaults.stateMachine.R       = 0;                    // Clear state machine faults
        gd3000Status.B.gd3000ClearErr = true;            // Clear GD3000 faults

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

Setting event to *cntrState.event* = *e_fault_clear* while in FAULT state represents a new request to proceed to INIT state. This request is purely user action and does not depend on actual fault status. In other words, it is up to the user to decide when to set *switchFaultClear* true. However, according to the interrupt data flow diagram shown in *Figure 55*, function *faultDetection()* is called before state machine function *state_table[event][state]()*. Therefore, all faults will be checked again and if there is any fault condition remaining in the system, the respective bits in *permFaults* and *tempFaults* variables will be set. As a consequence of *permFaults* not equal to zero, function *faultDetection()* will modify the application event from *e_fault_clear* back to *e_fault*, which means jump to fault state when state machine function *state_table[event][state]()* is called. Hence, INIT state will not be entered even though the user tried to clear the fault flags using *switchFaultClear*. When the next state (INIT) is entered, all fault bits are cleared, which means no fault is detected (*permFaults* = 0x0) and application variable *switchFaultClear* is manually set to true.

The application is scanning for following system warnings and errors:

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

The thresholds for fault detection can be modified in INIT state. Please see *[13]* for further information on how to set these thresholds using the MCAT. In addition to previous thresholds, fault state is entered if following errors are detected:

- BCTU trigger faults
- GD3000 pre-driver errors (overtemperature, desaturation fault, low supply voltage, DC bus overcurrent, phase error, framing error, write error after block, existing reset). See *[10]*
- FOC Error (irrelevant event call in state machine or eBEMF observer failure)

4.3.3.2. State – INIT

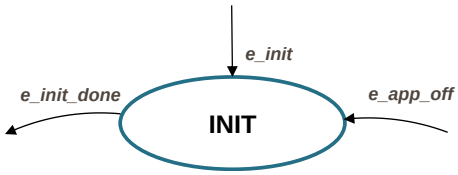

Figure 58. **INIT state with transitions**

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

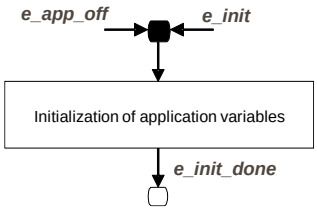

Figure 59. **Flow chart of state INIT**

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

4.3.3.3. State – READY

Figure 60. **READY state with transitions**

In READY state, application is waiting for user command to start the motor. The application is released from waiting mode by pressing the on board button SW5 or SW6 or by FreeMASTER interface setting the variable *switchAppOnOff = true* (see flow chart in *Figure 61*).

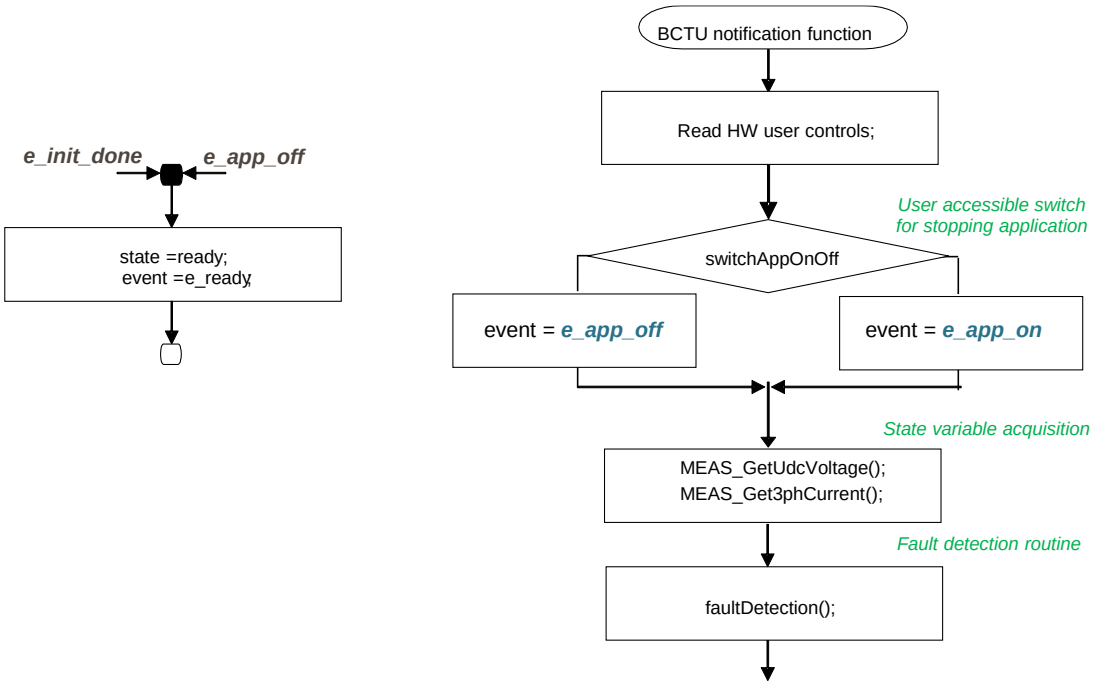

Figure 61. **Flow chart of state READY**

4.3.3.4. State – CALIB

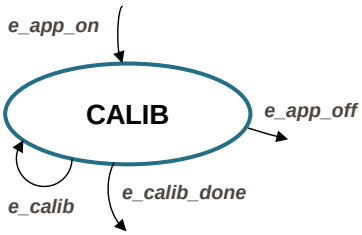

Figure 62. **CALIB state with transitions**

In this state, ADC DC offset calibration is performed. Once the state machine enters CALIB state, all PWM output are enabled. Calibration of the DC offset is achieved by generating 50% duty-cycle on the PWM outputs, and taking several measurements of the ADC0 and ADC1 channels connected to the current sensors. These measurements are then averaged, and the average value for the channel is stored. This value will be subtracted from the measured value when in normal operation. This way the half range DC offset, caused by voltage shift of 2.5 V in conditional circuitry (see *Figure 5*), is removed in the measured phase. State CALIB is a state that allows transition back to itself, provided no faults are present, the user does not request stop of the application (by *switchAppOnOff=true*), and the calibration process has not finished. The number of samples for averaging is set by macro FILTER_SAMPLE_NO_MEAS where actual number of samples is 2^(FILTER_SAMPLE_NO_MEAS+4) . After all samples have been taken and the averaged values successfully saved, the application event is automatically set to *cntrState.event=e_calib_done* and state machine can proceed to state ALIGN (see flow chart in *Figure 63*).

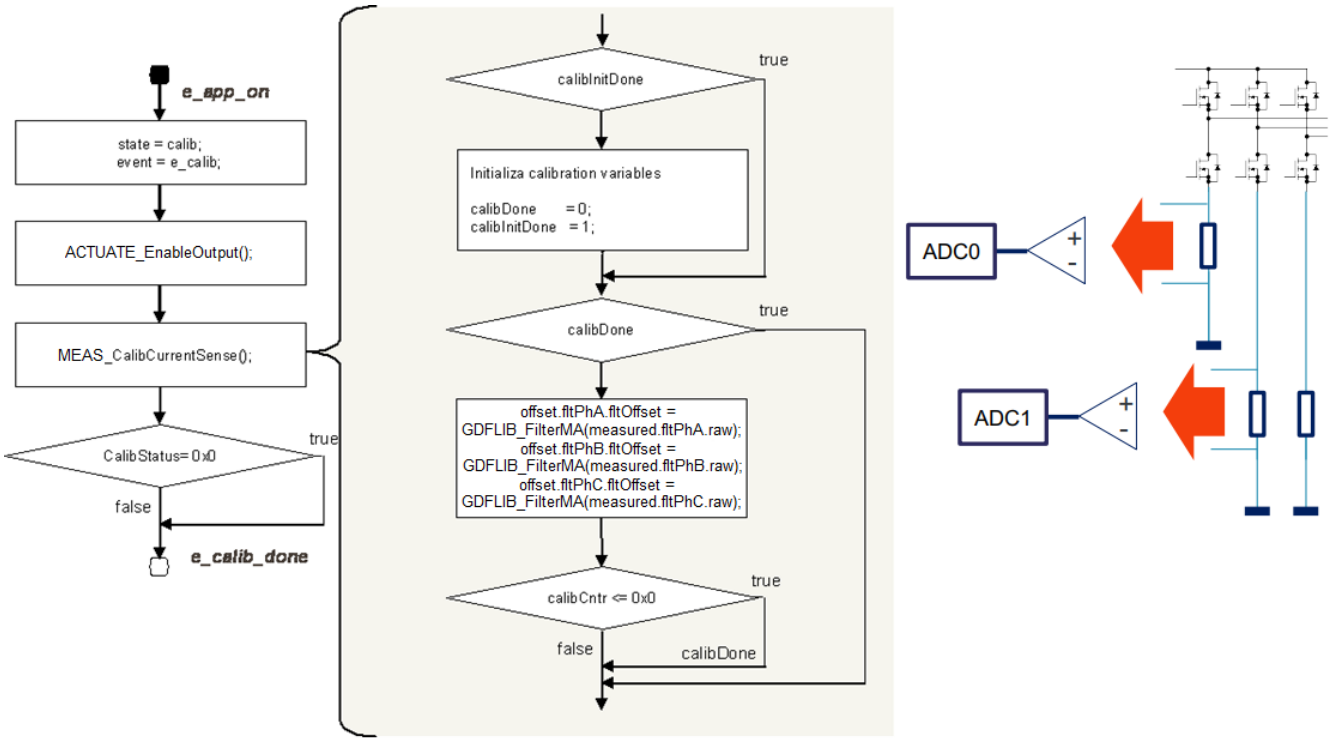

Figure 63. **Flow chart of state CALIB**

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

4.3.3.5. State – ALIGN

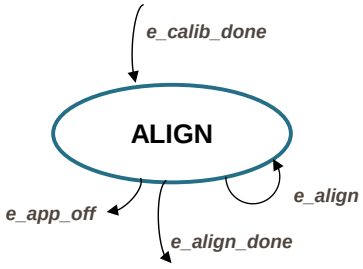

Figure 64. **ALIGN state with transitions**

This state manages alignment of the rotor and stator flux vectors to mark zero position. When using a model based approach for position estimation, the zero position is not known. The zero position is obtained at ALIGN state, where two state alignment is used to avoid sticking at 180deg. A DC voltage is applied to q-axis voltage for a certain period and after that to d-axis voltage for the rest of the alignment time. Ratio between d and q axis alignment time is given by macro ALIGN_D_FACTOR. This causes the rotor to rotate to "align" position, where stator and rotor fluxes are aligned. The rotor position in which the rotor stabilizes after applying this DC voltage is set as zero position. To get rotor stabilized at aligned position, a certain time is selected for alignment process. This time and the amplitude of DC voltage used for alignment can be modified by MCAT tool. Timing is implemented using a software

counter that counts from a pre-defined value down to zero. During this time, the event remains set to *cntrState.event=e_align*. When the counter reaches zero, the counter is reset back to the pre-defined value, and event is automatically set to *cntrState.event=e_align_done*. This enables a transition to RUN state see flow chart in *Figure 65*.

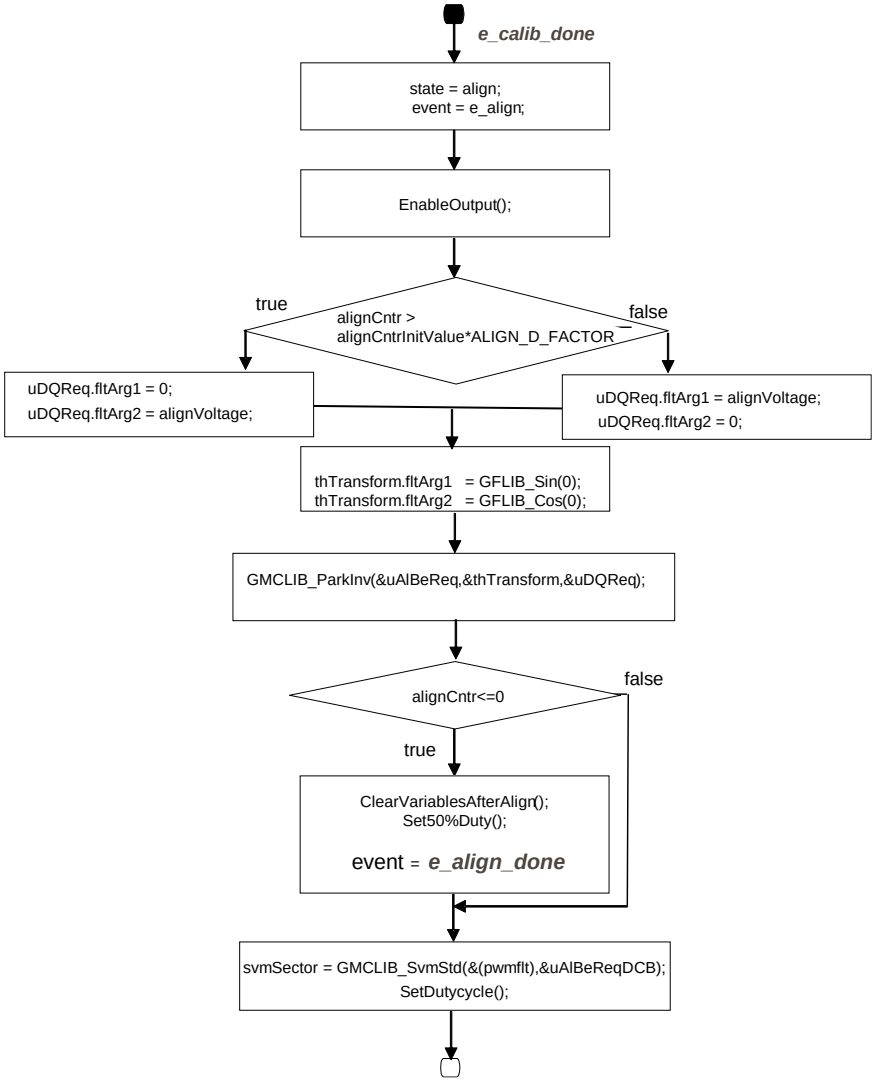

Figure 65. **Flow chart of state ALIGN**

A transition to FAULT state is performed automatically when a fault occurs. Transition to INIT state is performed by setting the event to *cntrState.event=e_app_off*, which is done automatically on falling edge of *switchAppOnOff=false* using FreeMASTER or simultaneously pressing SW5 and SW6.

4.3.3.6. State – RUN

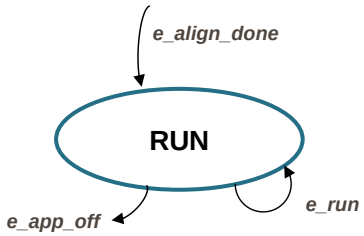

Figure 66. **RUN state with transitions**

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

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

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

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

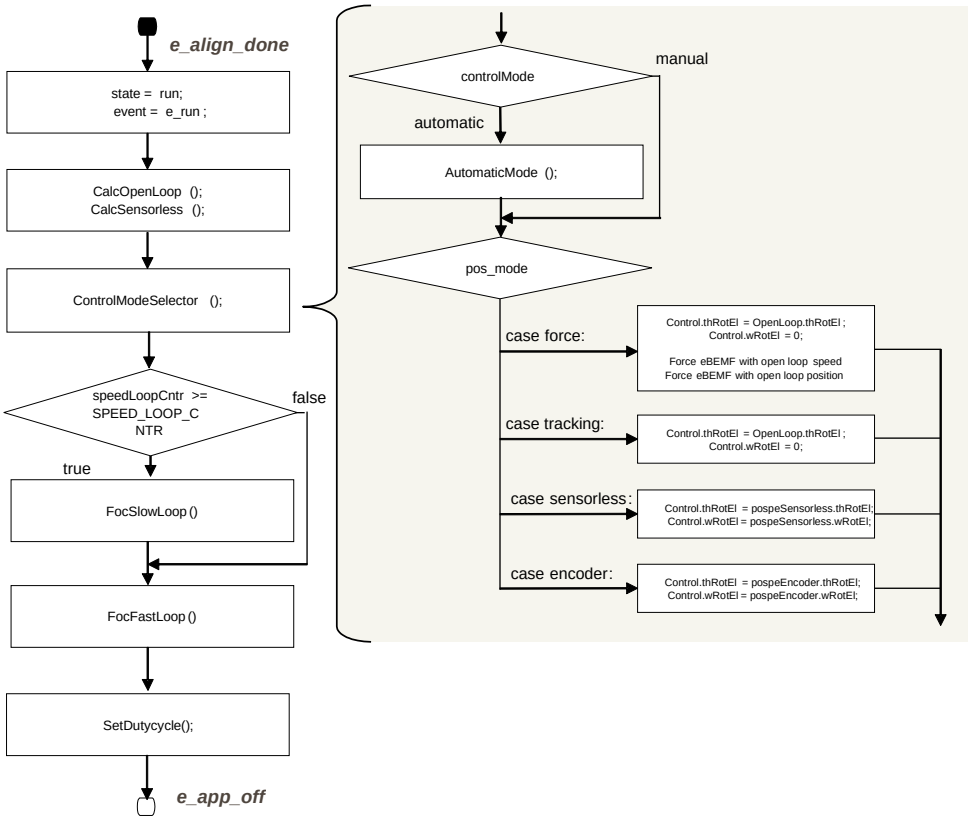

Figure 67. **Flow chart of state RUN**

Calculation of fast current loop is executed every BCTU interrupt, while calculation of slow speed loop is executed every Nth BCTU interrupt. Arbitration is done using a counter that counts from value N down to zero. When zero is reached, the counter is reset back to N and slow speed loop calculation is performed. N value (macro SPEED_LOOP_CNTR) is automatically calculated by MCAT form current loop sample time and speed loop sample time parameters. This way, only one interrupt is needed for both loops and timing of both loops is synchronized. Slow loop calculations are finished before entering fast loop calculations (see flow chart in *Figure 67*).

Figure 68 shows implementation of FOC algorithm, used functions and variables. As can be seen from the diagram, rotor position and speed are estimated by eBEMF observer. This is a default rotor position and speed feedback for FOC. To run Encoder based FOC, ENCODER macro must be set to *true* and PM motor provided with this motor control kit replaced by PM motor of the comparable power and equipped with Encoder sensor. As mentioned previously, Encoder based FOC can be activated/deactivated by setting *switchSensor* variable to *encoder/sensorless*.

A transition from RUN state to FAULT state is performed automatically when a fault occurs. A transition to INIT state is performed by setting the event to *cntrState.event=e_app_off*, which is done automatically on falling edge of *switchAppOnOff=false* using FreeMASTER or keeping user buttons SW5 and SW6 pressed.

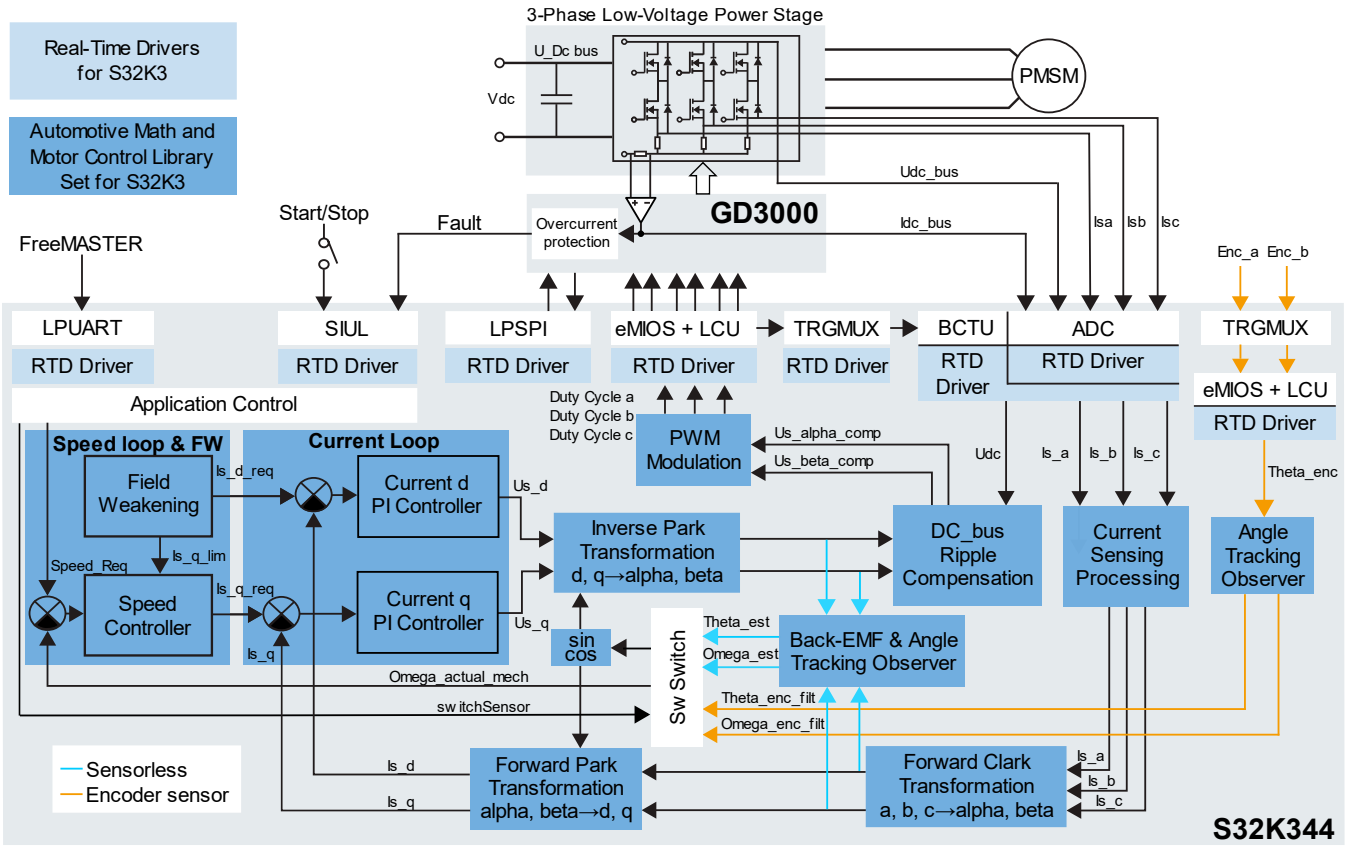

Figure 68. **Sensorless and Sensorbased FOC with FW implementation on S32K344**

4.3.4. AMMCLib Integration

Application software of the FOC Sensorless control with field weakening is built using NXP's Automotive Math and Motor Control Library set (AMMCLib), a precompiled, highly speed-optimized off-the-shelf software library designed for motor control applications. The most essential blocks of the FOC structure are presented in *Figure 68*. AMMCLib supports all available data type implementations: 32-bit fixed-point, 16-bit fixed-point and single precision floating-point. In order to achieve high performance of the S32K344 core, floating point arithmetic is used as a reference for this motor control application.

Current Loop function AMCLIB_CurrentLoop unites and optimizes most inner loop of the FOC cascade structure *Figure 68*. It consists of two PI controllers and basic mathematical operations which calculate errors between required and feedback currents and limits for PI controllers based on the actual value of the DC bus voltage. All functions and data structures are presented in *Figure 69*.

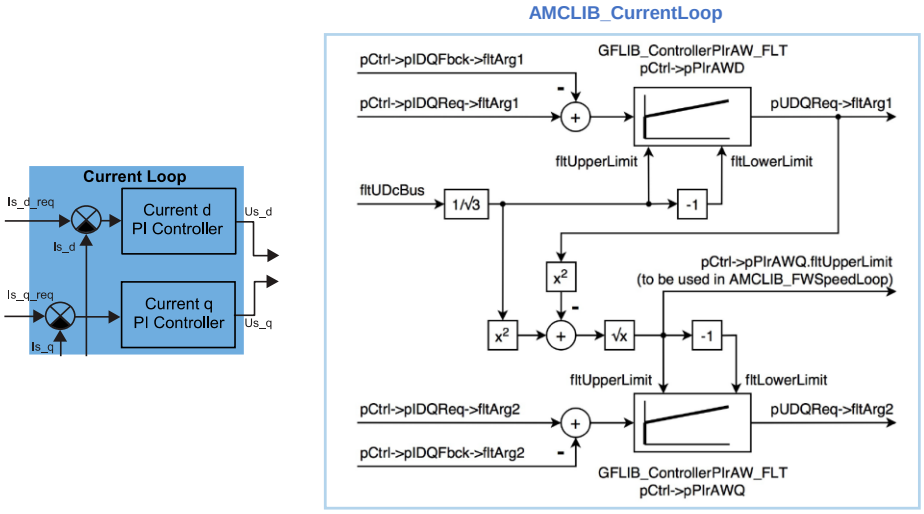

Figure 69. **Functions and data structures in AMCLIB_CurrentLoop**

Required d- and q-axis stator currents can be either manually modified or generated by outer loop of the cascade structure consisting of: Speed Loop and Field Weakening (FW) as shown in *Figure 68*. To achieve highly optimized level, AMCLIB_FWSpeedLoop merges two functions of the AMMCLIB, namely speed control loop AMCLIB_SpeedLoop and field weakening control AMCLIB_FW, *Figure 70*. AMCLIB_SpeedLoop consists of speed PI controller GFLIB_ControllerPIpAW, speed ramp GFLIB_Ramp placed in feedforward path and exponential moving average filter GFLIB_FilterMA placed in the speed feedback. AMCLIB_FW function is NXP's patented algorithm (US Patent No. US 2011/0050152 A1) that extends the speed range of PMSM beyond the base speed by reducing the stator magnetic flux linkage as discussed in section *Field weakening*. All functions and data structures used in AMCLIB_FW function are shown in *Figure 70*.

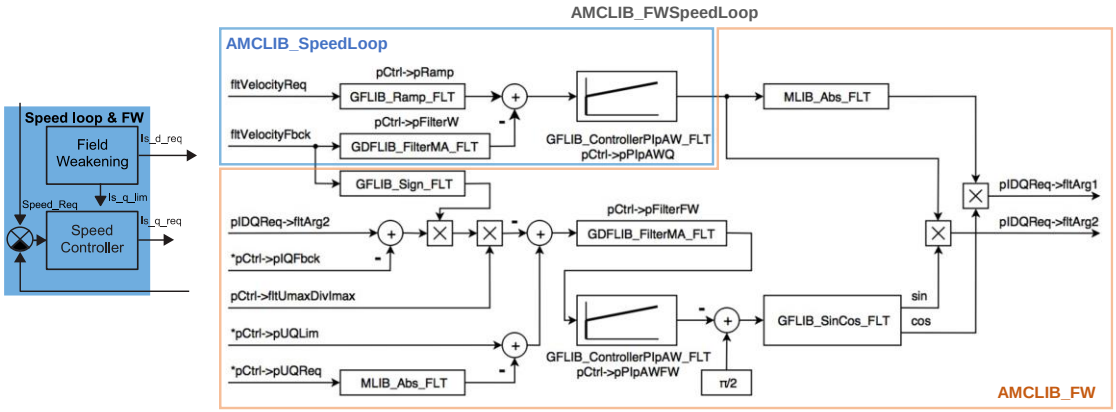

Figure 70. **Functions and data structures in AMCLIB_FWSpeedLoop**

AMCLIB_FW key advantages:

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

eBEMF observer AMCLIB_BemfObsrv and Angle tracking observer AMCLIB_TrackObsrv constitute important blocks in this application, *Figure 68*. They estimate rotor position and speed based on the inputs, namely stator voltages $u_{\alpha\beta}$ and currents $i_{\alpha\beta}$, *Figure 71*. AMCLIB_BemfObsrv transforms inputs quantities from stationary reference frame α/β to quasi-synchronous reference frame γ/δ that follows the real synchronous rotor flux frame d/q with an error θ_{err}. AMCLIB_BemfObsrv algorithm is based on the mathematical model of the PMSM motor with excluded BEMF terms $e_{\gamma\delta}$. BEMF terms are estimated as disturbances in this model, generated by PI controllers. The estimated BEMF values are used for calculating the phase error θ_{err}, which is provided as an output of the BEMF observer.

To align both frames and provide accurate estimates, this phase error θ_{err} must be driven to zero. This is a main role of the Angle tracking observer AMCLIB_TrackObsrv which is attached to function of the eBEMF observer AMCLIB_BemfObsrv, *Figure 71*. AMCLIB_TrackObsrv is an adopted phase-locked-loop algorithm that estimates rotor speed and position keeping $\theta_{err} = 0$. This is ensured by a loop compensator that is PI controller. While PI controller generates estimated rotor speed, integrator used in this phase-locked-loop algorithm serves estimated rotor position.

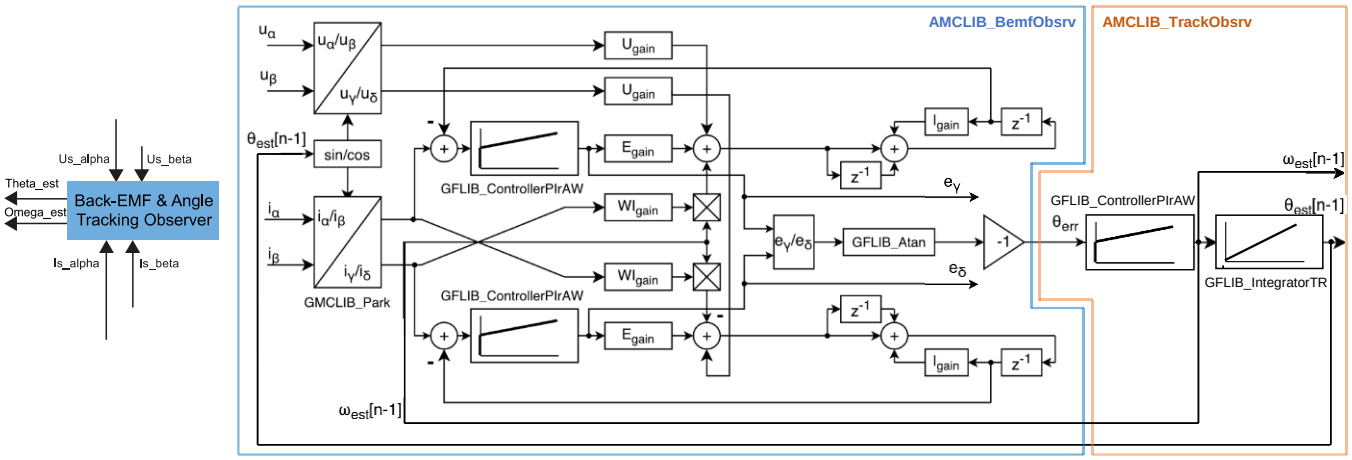

Figure 71. **Structure of the AMCLIB_BemfObsrv and AMCLIB_TrackObsrv**

More details related to AMMCLib FOC functions can be found in S32K34x AMMCLib User's Guide on standard installation path C:\NXP\AMMCLIB\S32K3xx_AMMCLIB_vX.Y.Z\doc\S32K3XXMCLUG.pdf. Parameters of the PI controllers placed in the speed control loop, current control loop, eBEMF and Angle tracking observer can be tuned by using NXP's Motor Control Application Tuning tool (MCAT). Detailed instructions on how to tune parameters of the FOC structure by MCAT are presented in [14] , *[15]* .

4.3.5. MCAT Integration

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

The MCAT tool runs under FreeMASTER online monitor, which allows the real-time tuning of the motor control application. Respecting the parameters of the controlled drive, the correct values of control structure parameters are calculated, which can be directly updated to the application or stored in

an application static configuration file. The electrical subsystems are modeled using physical laws and parameters of the PI controllers are determined using Pole-placement method. FreeMASTER MCAT control and tuning is described in the section *FreeMASTER and MCAT user interface*.

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

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

5. FreeMASTER and MCAT user interface

The FreeMASTER debugging tool is used to control the application and monitor variables during run time. Communication with the host PC passes via USB. However, because FreeMASTER supports serial port communication, there must be a driver for the physical USB interface, OpenSDA, installed on the host PC that creates a virtual COM port from the USB. The driver shall be installed automatically plugging S32K344EVB to USB port. The application configures the LPUART module of the S32K344 for a communication speed of 115200bps. Therefore, the FreeMASTER user interface also needs to be configured respectively.

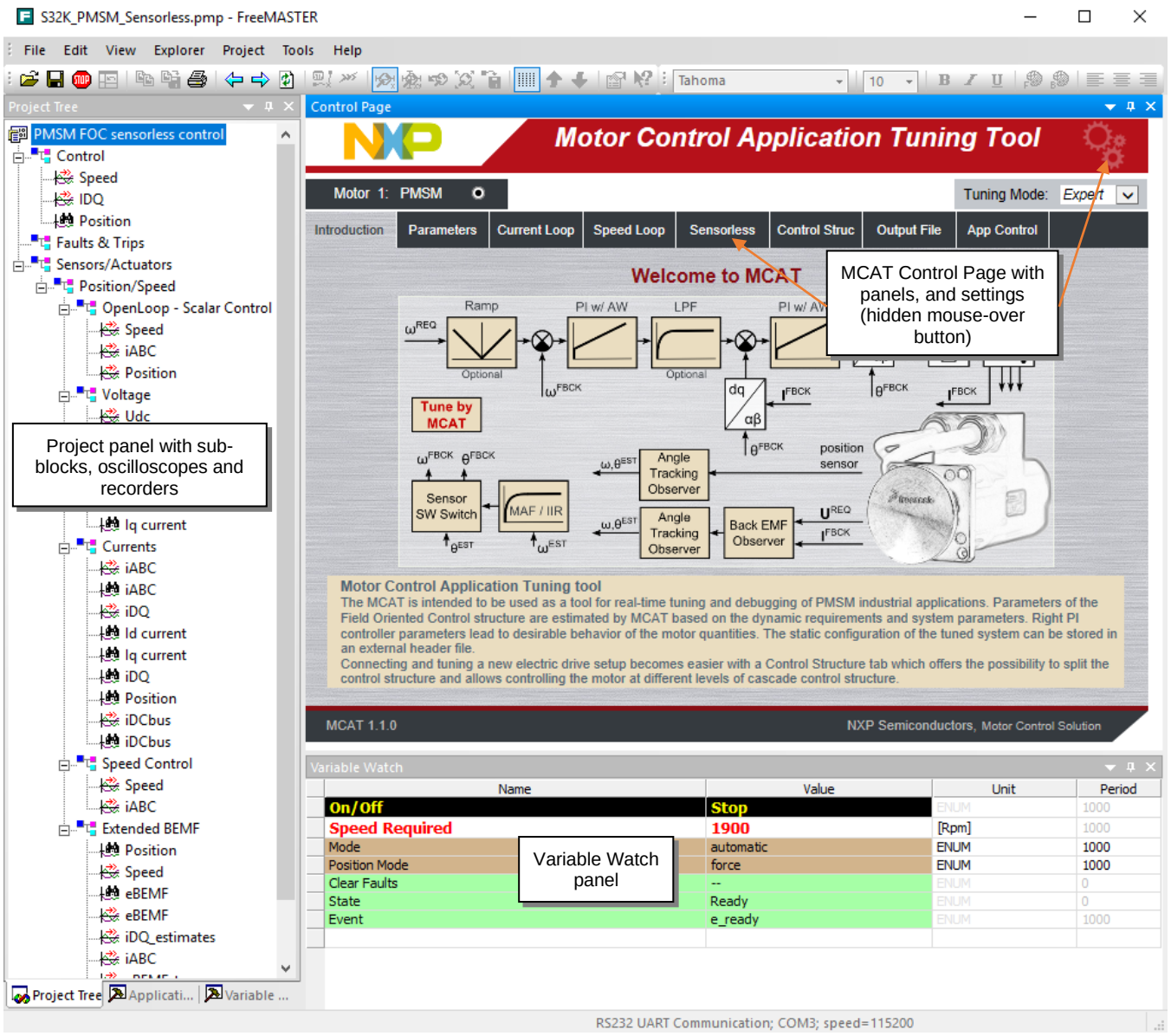

Figure 72. **FreeMASTER and Motor Control Application Tunning Tool**

5.1. MCAT Settings and Tuning

5.1.1. Application configuration and tuning

FreeMASTER and MCAT interface (*Figure 72*) enables online application tuning and control. The MCAT tuning shall be used before the very first run of the drive to generate the configuration header file (PMSM_appconfig.h). Most of the variables are accessible via MCAT online tuning (thus can be updated anytime). They are highlighted when mouse pointer is over the button “Update Target” (see *Figure 73*). Some parameters (especially the fault limit thresholds) must be set using the configuration header file generation, which can be done on the “Output File” panel by clicking the “Generate Configuration File” (see *Figure 74*).

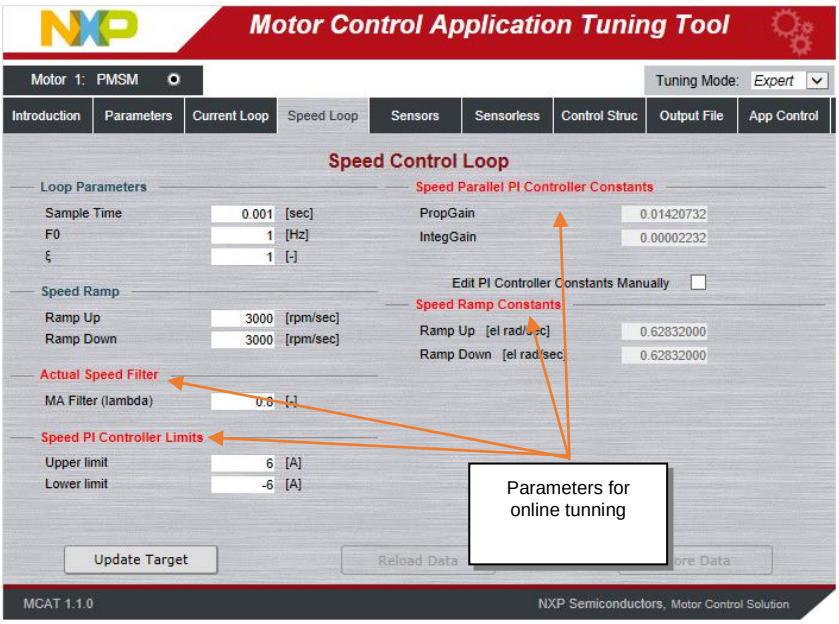

Figure 73. **Parameters for online tunning**

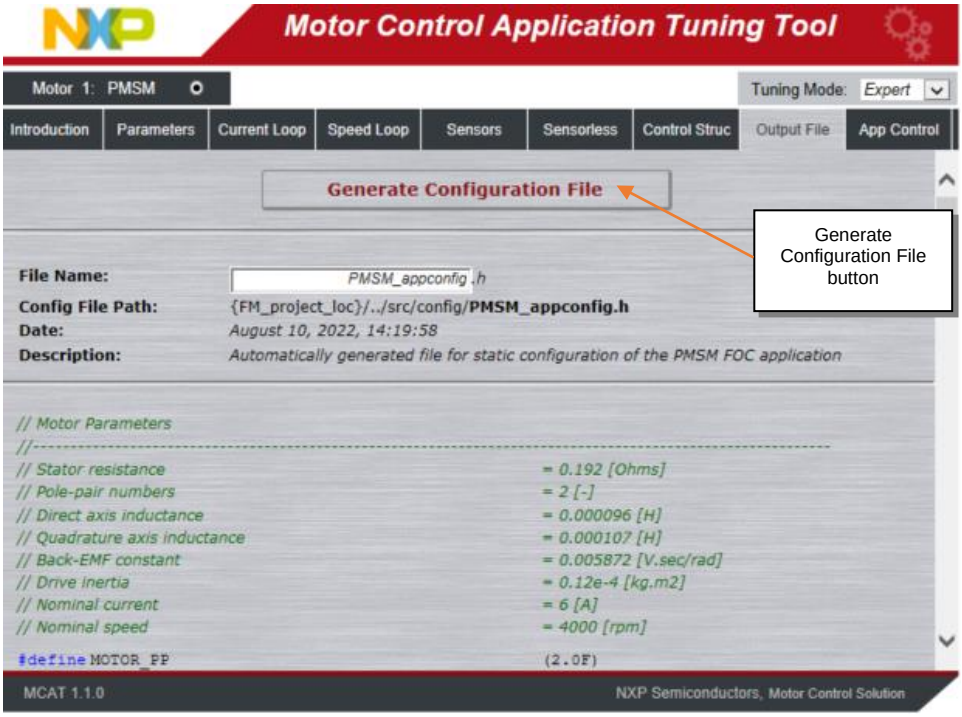

Figure 74. **Output File panel and "Generate Configuration File" button**

Parameters runtime update is done using the "Update Target" button (see *Figure 75*). Changes can be also saved using "Store Data" button, or reloaded to previously saved configuration using "Reload Data" button. Only stored configuration can be generated to PMSM_appconfig.h header file. File holding the configuration is "{Project Location}\FreeMASTER_control\ MCAT\param_files\M1_params.txt". Settings for various motors, scenarios can be backed up and selected setting can be loaded by replacing the content of M1_params.txt.

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

NOTE

MCAT tool can be configured using hidden mouse-over “Settings” button (see *Figure 72*), where a set of advanced settings, for example PI controller types, speed sensors and other blocks of the control structure can be changed. However, it is not recommended to change these settings since it will force the MCAT to look for a different variables names and to generate different set of constants than the application is designed for. See MCAT tool documentation available at nxp.com.

The application tuning is provided by a set of MCAT pages dedicated to every part of the control structure. An example of the Application Parameters Tuning page is in *Figure 75.* Following list of settings pages is based on the PMSM sensorless application.

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

Changes can be tested using MCAT “Control Struc” page (*Figure 76*), where the following control structures can be enabled:

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

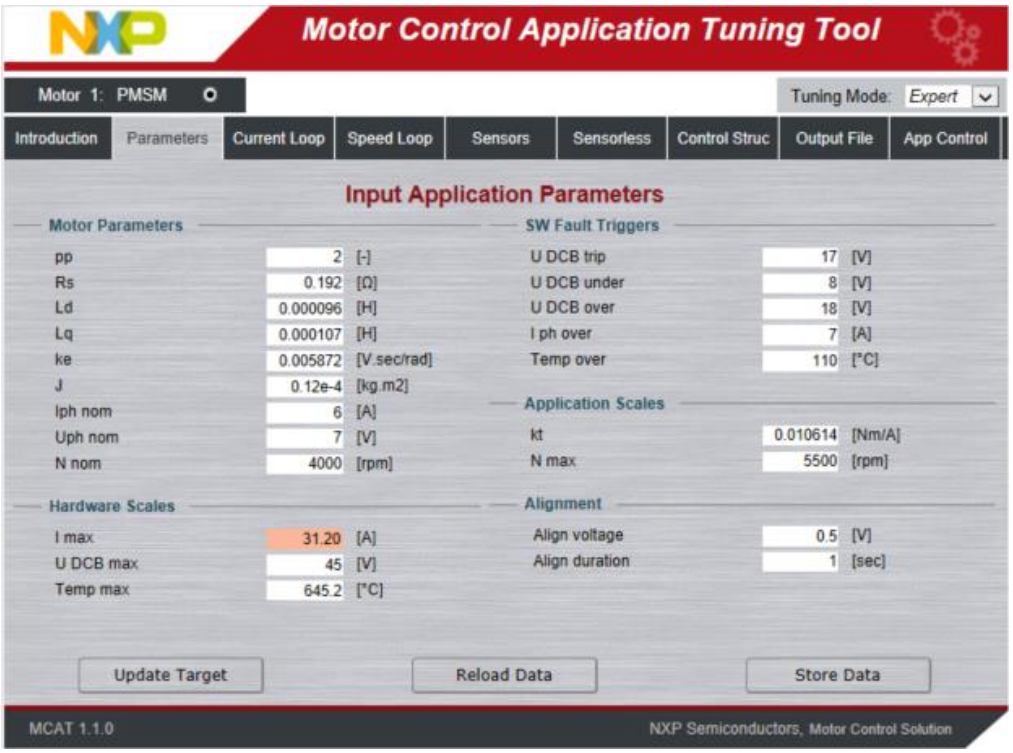

Figure 75. **MCAT input application parameters page**

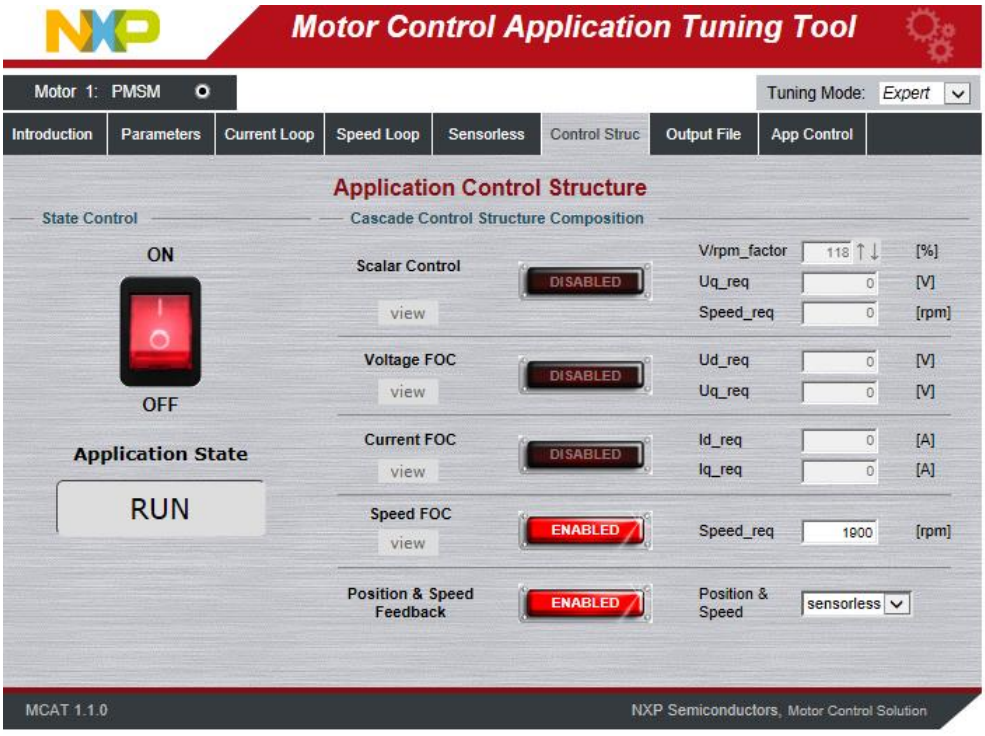

Figure 76. **MCAT application control structure page**

5.2. MCAT application Control

All application state machine variables can be seen on the FreeMASTER MCAT App control page as shown in *Figure 77*. Warnings and faults are signaled by a highlighted red color bar with name of the fault source. The warnings are signaled by a round LED-like indicator, which is placed next to the bar with the name of the fault source. The status of any fault is signaled by highlighting respective indicators. In *Figure 77*, for example, there is pending fault flag and one warning indicated ("Udcb LO" - DC bus voltage is close to its under voltage conditions). That means that the measured voltage on the DC bus exceeds the limit set in the MCAT_Init function. The warning indicator is still on if the voltage is higher than the warning limit set in INIT state. In this case, the application state FAULT is selected, which is shown by a frame indicator hovering above FAULT state. After all actual fault sources have been removed, no warning indicators are highlighted, but the fault indicators will remain highlighted. The pending faults can now be cleared by pressing the "FAULT" button. This will clear all pending faults and will enable transition of the state machine into INIT and then READY state. After the application faults have been cleared and the application is in READY state, all variables should be set to their default values. The application can be started by application On/Off switch. Successful selection is indicated by highlighting the On/Off button in green. Required speed can be set by clicking on speed gauge or by modifying FreeMASTER variable ” Speed Required”.

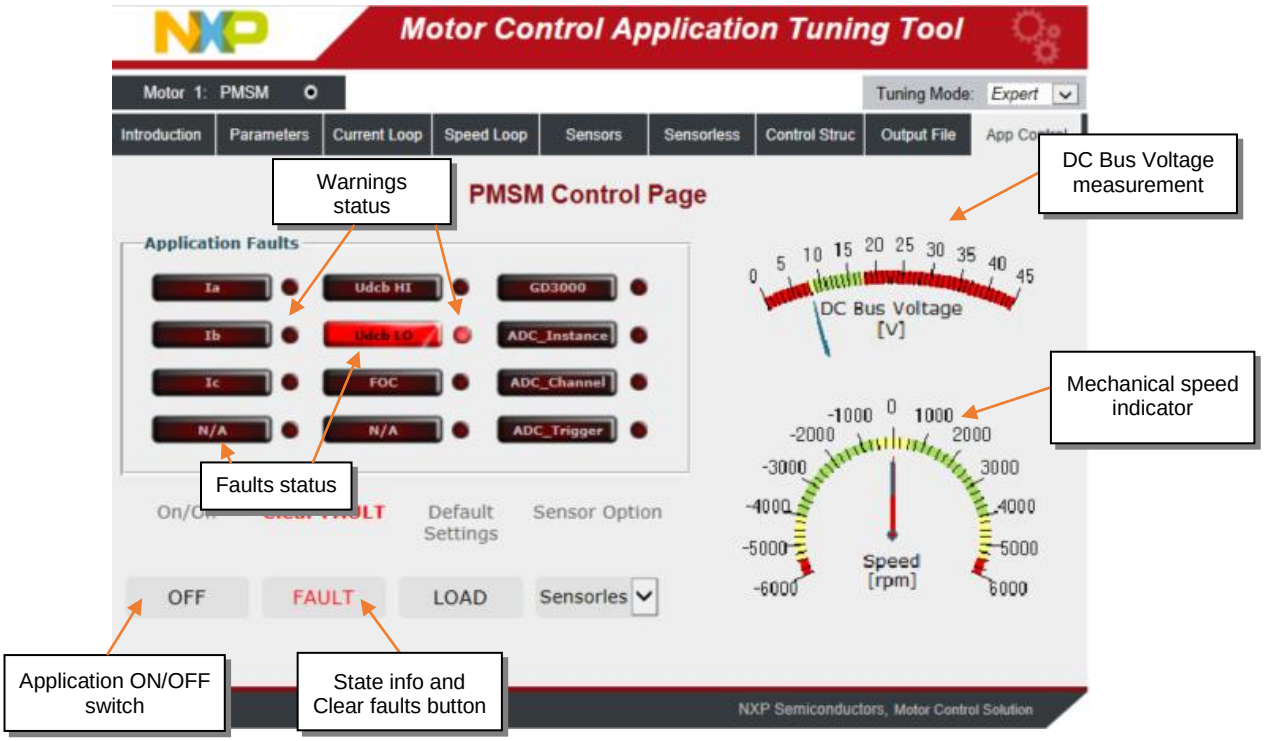

Figure 77. **FreeMASTER MCAT Control Page for controlling the application**

6. Conclusion

Design, described in this application note shows the simplicity and efficiency in using the S32K344 microcontroller for Sensorless PMSM motor control and introduces it as an appropriate candidate for various applications in the automotive area. MCAT tool provides interactive online tool which makes the PMSM drive application tuning friendly and intuitive.

7. References

[1] MCSPTE1AK344: S32K344 Development Kit for BLDC and PMSM motor control

[2] S32 Design Studio IDE for S32 Platform

[3] Real-Time Drivers (RTD)

[4] FreeMASTER Run-Time Debugging Tool

[5] Automotive Math and Motor Control Library Set

[6] S32K3xx MCU Family

[7] S32K3xx MCU Family - Reference Manual

[8] S32K3x4-Q172 General Purpose Development Board

[9] DEVKIT-MOTORGD: Low-Cost Motor Control Solution for DEVKIT Platform

[10] GD3000: 3-Phase Brushless Motor Pre-Driver

[11] S32K3 Motor control: The new TPPSDK (GD3000 driver) based on RTD

[12] Rashid, M. H. Power Electronics Handbook, 2nd Edition. Academic Press

[13] Motor Control Application Tuning (MCAT) Tool

[14] AN4912 Tuning 3-Phase PMSM Sensorless Control Application Using MCAT Tool

[15] AN4642 Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors. In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for

Table continues on the next page...

doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products —This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer. In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

Translations — A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 11/2021
Document identifier: AN13767

arm