AN13879

3-phase PMSM Motor Control Power Inverter Module

Rev. 1.0 — 16 January 2024 **Application note**

Document Information

Information	Content
Keywords	Power Invert module, Field Oriented Control, GD3160, MPC5775E
Abstract	Application note AN13879 describes the design of a Field field-oriented control for 3-phase PMSM motors based on LEM current sensors and resolver position sensing. The design targets automotive motor control applications.

1 Introduction

Application note AN13879 describes the design of a 3-phase Permanent Magnet synchronous Motor (PMSM) vector control drive with (Hall effect) LEM current sensors and resolver position sensing. The design is targeted at automotive motor control (MC) applications.

The application note describes an example of motor control design (EV-INVERTERHD) using the NXP family of automotive motor control MCUs based on a 32-bit Power ArchitectureÒ technology optimized for a full range of automotive applications. The system in Figure 1 is based on an advanced MOSFET SiC power module 1200 V, 612 A, Starpower MD612HTC120P6H with Cree 13 mOhm SiC Mosfet.

The power module is driven by an advanced gate driver GD 3160 dedicated to inverter applications.

The following are the supported features:

- 3-phase PMSM speed field-oriented control.
- Current sensing with LEM sensors.
- Application control user interface using the FreeMASTER debugging tool.
- Motor Control Application Tuning (MCAT) tool.
- Rotor position and speed measurement using a resolver transducer.

2 System concept

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

- Targeted at the MPC5775E (refer to the dedicated Reference manual for MPC5775E available at https://www.nxp.com). See References for more information.
- S32 Design Studio (see References)
- MOSFET SiC Starpower power module 1200 V, 612 A,
- GD3160 advanced High Voltage Isolated Gate Driver with Segmented Drive for SiC MOSFETs
- Control technique incorporating:
- Field-Oriented Control of 3-phase PM synchronous motor with resolver position sensor
- Closed-loop speed control with action period 1 ms.
- Closed-loop current control with action period 100 µs.
- Bidirectional rotation.
- Flux and torque-independent control.
- Field weakening control extending the speed range of the PMSM beyond the base speed.
- The Enhanced Time Processing Unit (eTPU) computes the position and speed.
- Sensing of three-phase motor currents.
- FOC state variables sampled with 100 µs period.
- Automotive Math and Motor Control Library (AMMCLIB) - FOC algorithm built on blocks of precompiled SW library (see section References).
- Use of eTPU Motor control function set to offload CPU.
- FreeMASTER software control interface (motor start/stop, speed setup).
- FreeMASTER software monitor
- FreeMASTER embedded Motor Control Application Tuning (MCAT) tool (motor parameters, current loop, speed loop) (see section References).
- FreeMASTER software MCAT graphical control page (required speed, actual motor speed, start/stop status, DC-Bus voltage level, motor current, system status).
- FreeMASTER software speed scope (observes actual and desired speeds, DC-Bus voltage and motor current).

- FreeMASTER software high-speed recorder (reconstructed motor currents, vector control algorithm quantities).
- DC-Bus overvoltage and undervoltage, overcurrent, overload, and overtemperature protection.

Figure 1. SiC Power inverter Module with MPC5775E and GD3160

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. Therefore, 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 Equation 1:

$$T_e = \frac{3}{2} \cdot pp \cdot \bar{\psi}_s \times \bar{i}_s \tag{1}$$

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 nonpower 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. The 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.

Separation of flux and torque components 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 alters 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 used to measure the rotor position and the position of the rotor flux axis.

When the reference frame phase sets 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 in d and q-axis current components appearing as DC values. A synchronous reference frame implies the utilization of simple current controllers to control the machine's demanded torque and magnetizing flux, therefore simplifying the control structure design.

Figure 2 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 the d, q rotating frame.
4. The control output is the stator voltage space vector and is transformed by an inverse Park transformation 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 Automotive Math and Motor Control Library Set for MPC577xC).

The motor-magnetizing flux's position must be known to decompose currents into torque and flux-producing components (i_{sd}, i_{sq}). Decomposing SW requires knowledge of accurate rotor position as being strictly fixed with magnetic flux. AN13879 deals with the FOC control, where the position and velocity is obtained by either a position/velocity estimator.

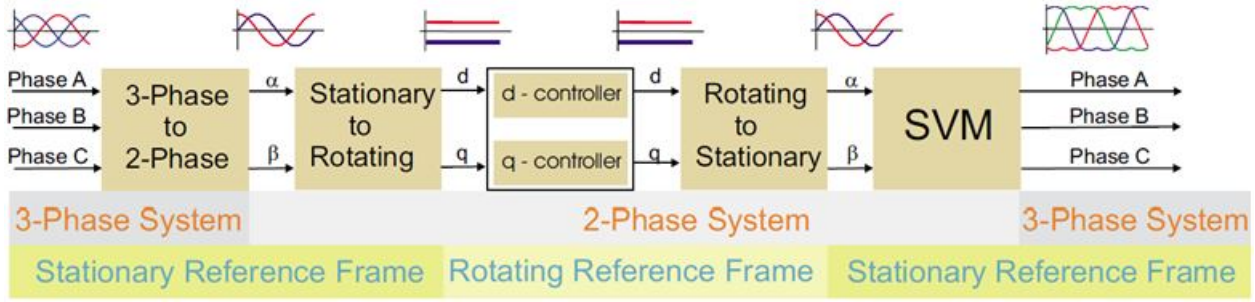

Figure 2. Field-oriented control transformations

3.2 PMSM model in quadrature phase synchronous reference frame

The quadrature phase model in a synchronous reference frame is popular for field-oriented control structures because both controllable quantities, current and voltage, are DC values. The quadrature phase model in a synchronous reference frame allows employing only simple controllers to force the machine currents into the defined states. Furthermore, complete 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 Equations 2 and 3:

$$\begin{bmatrix} u_a \\ u_b \\ u_c \end{bmatrix} = R_s \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} + \frac{d}{dt} \begin{bmatrix} \psi_a \\ \psi_b \\ \psi_c \end{bmatrix} \quad (2)$$

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

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

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

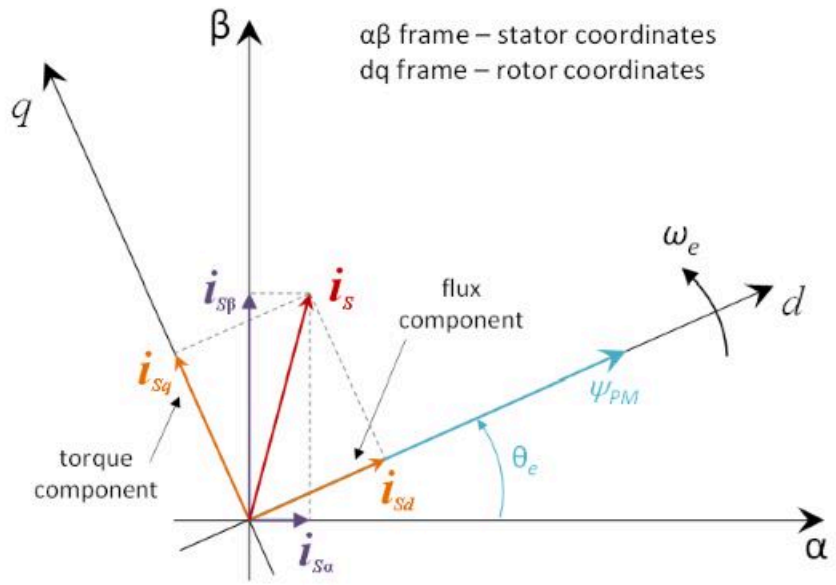

Figure 3. Orientation of stator (stationary) and rotor (rotational) reference frames, with current components transformed into both frames

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 3. Such transformation, after some mathematical corrections, yields Equation 4:

$$\begin{bmatrix}u_d\\ u_q\end{bmatrix}=R_s\begin{bmatrix}i_d\\ i_q\end{bmatrix}+\begin{bmatrix}L_d & 0\\ 0 & L_q\end{bmatrix}\frac{d}{dt}\begin{bmatrix}i_d\\ i_q\end{bmatrix}+\omega_e\begin{bmatrix}0 & -L_q\\ L_d & 0\end{bmatrix}\begin{bmatrix}i_d\\ i_q\end{bmatrix}+\omega_e\Psi_{PM}\begin{bmatrix}0\\ 1\end{bmatrix} \quad (4)$$

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

$$u_d = R_s i_d + L_d\frac{di_d}{dt} \quad (5)$$

$$u_q = R_s i_q + L_q\frac{di_q}{dt} \quad (6)$$

Equation 5 and 6 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, by Equation 7 and 8:

$$K_p = 2\xi\omega_0 L - R \tag{7}$$

$$K_i = \omega_0^2 L \tag{8}$$

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

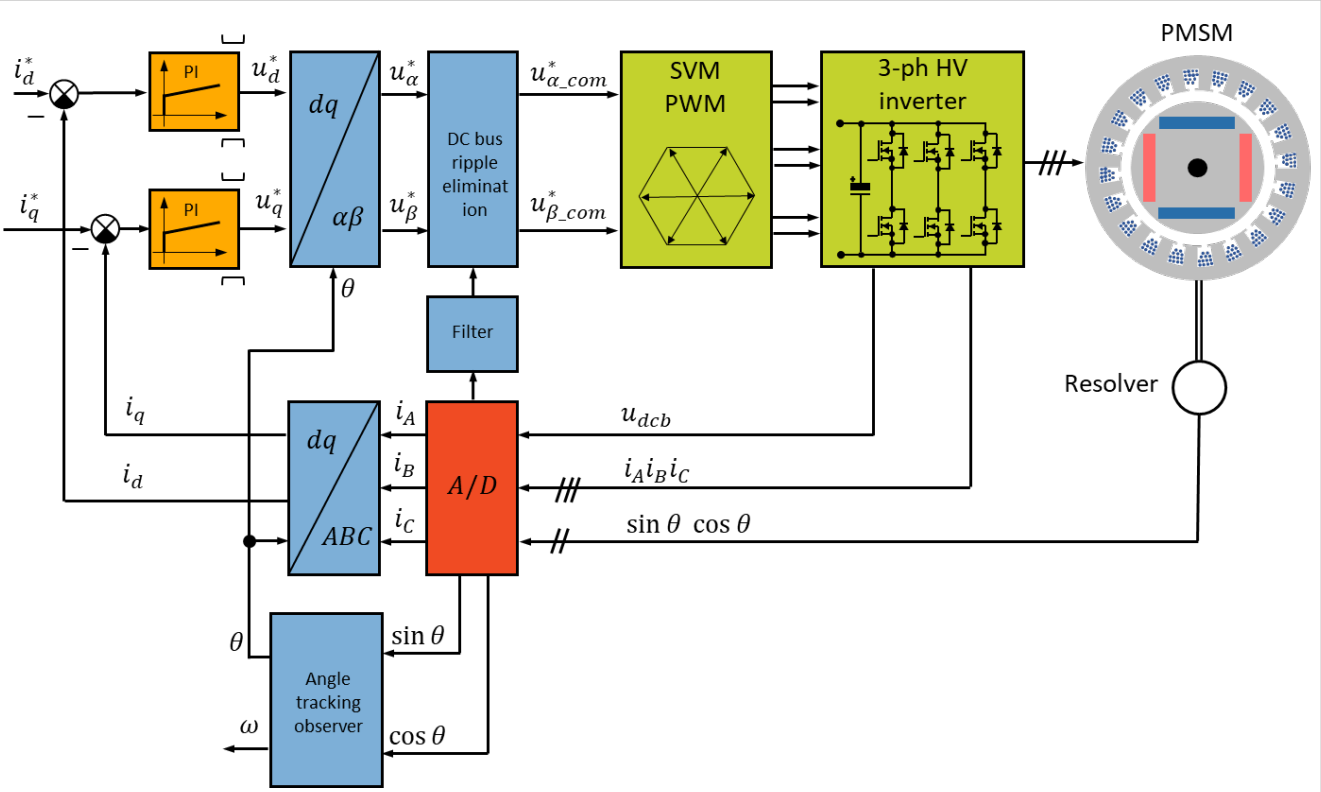

Figure 4. FOC control structure

3.3 FOC feedback - current voltage and position sensing

One leg of the 3-phase voltage inverter shown in Figure 5 uses three LEM sensors (see Figure 6 U30, U31, U32) placed in output phases as current sensors. DC, AC, or pulsed stator phase current, which flows through the LEM sensor, produces a voltage drop based on the Hall effect, which is interfaced with the AD converter of the microcontroller through conditional circuitry (see Figure 7).

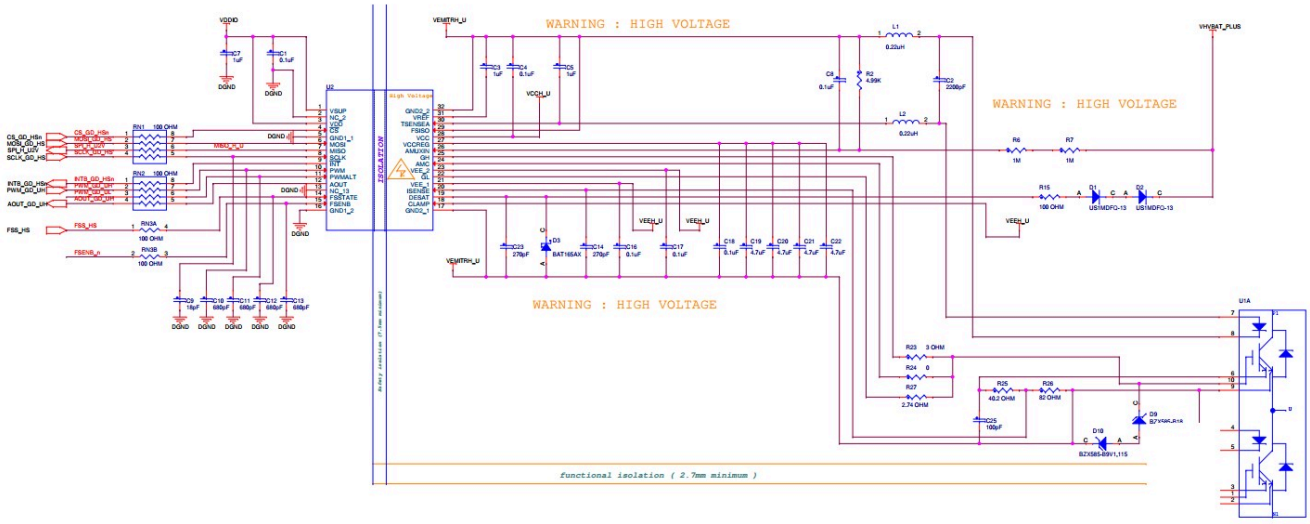

Figure 5. One leg of a three-phase inverter with LEM current sensors

Figure 6 and Figure 7 shows a gain setup and input signal filtering circuit for an operational amplifier, which provides the conditional circuitry and adjusts voltages to fit into the ADC input voltage range.

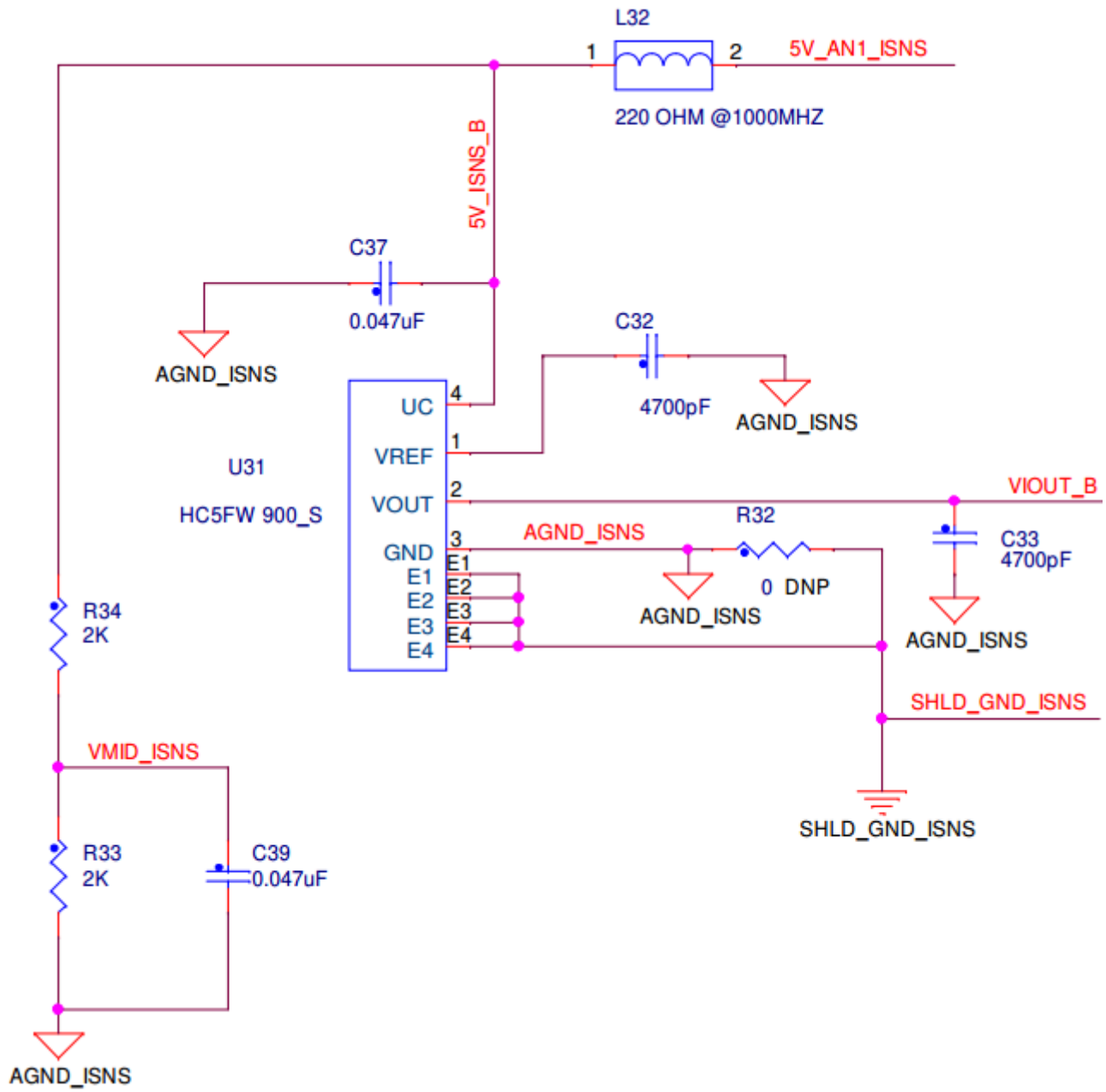

Figure 6. Phase current measurement circuitry

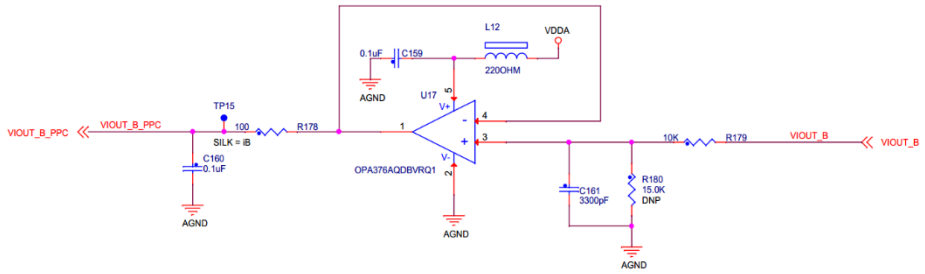

Figure 7. Phase current measurement conditional circuitry

The phase current is sampled in the middle of the PWM period. In standard motor operation, where the supplied voltage is generated using the space vector modulation, the sampling instant of phase current occurs in the middle of the PWM period, where all bottom transistors are switched on. The sampling point is delayed due to a turn-on gate signal propagation through the Gate driver. Delay can be modified in the eTPU AS function setting. The typical bandwidth for open loop transducers H5FW is 40 kHz.

MPC5775E uses an eTPU timer for switching pattern generation. The duty cycle of PWMs is limited to 98% by default. Users can change the default settings and type of modulation. All available functions for eTPU can be found in eTPU function selector.

In a typical voltage sensing implementation, a resistive voltage divider is used to scale the DC-link voltage to suit the input range of the voltage sensor U11 shown in Figure 8. A differential output voltage that is proportional to the input voltage is created on the other side of the optical isolation barrier. A typical 100 kHz wide bandwidth provides enough information for the effective motor control use case.

All listed feedbacks, three phase currents and DCbus voltage are sampled at once by eQADCs.

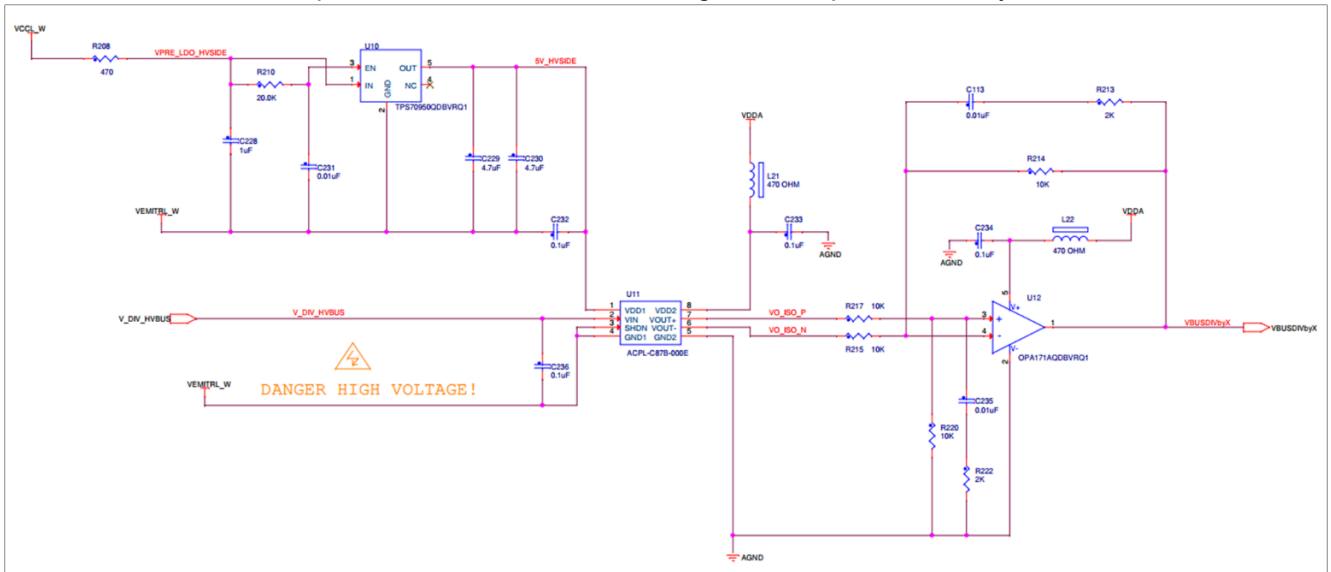

Figure 8. Voltage sensing and conditioning circuit

The excitation signal for the resolver is generated by circuitry in Figure 9. Excitation circuits consist of a Sallen-key third-order filter and output amplifier. The Sallen key filter produces a sine wave for resolver excitation at 10 kHz. The source of the excitation signal is eTPU. The output amplifier modulates the excitation amplitude at the specific level necessary for a variable inductance resolver.

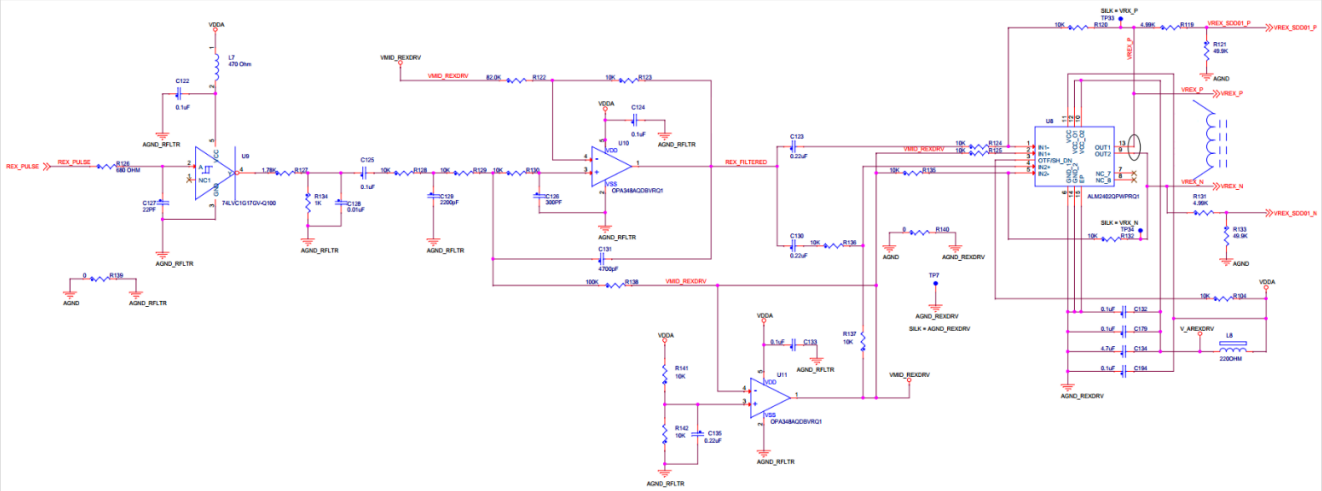

Figure 9. Sallen key third order filter and output adapter

Output signals from resolver sin and cos signals are connected directly to conditioning circuits Figure 10. After conditioning, signals are processed in SDADC running in differential mode.

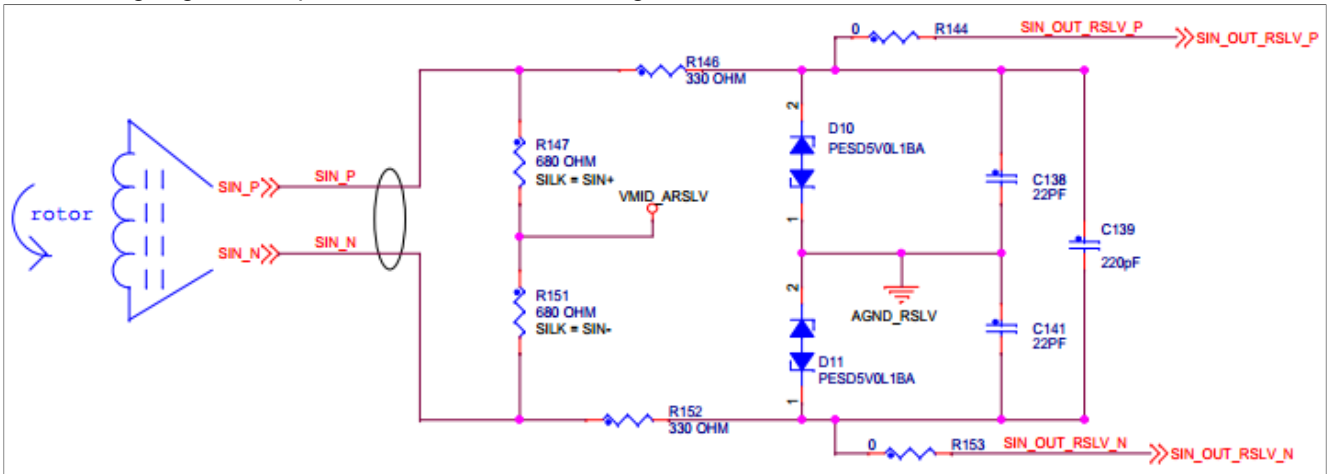

Figure 10. Resolver feedback signals conditioning circuits

3.4 Rotor position/speed estimation

Different sensor types may require different approaches to evaluate the speed and position of the motor. The NXP approach for resolver systems uses an Angle Tracking Observer (ATO). Figure 11 is based on the Phase Lock Loop technique. The ATO input is a position error between the position given by the sensor and the estimated ATO position. The PI controller in the ATO loop minimizes the input error by adjusting a control variable. In this case, the control variable is equivalent to a motor speed. Integration of the speed leads to the estimated position.

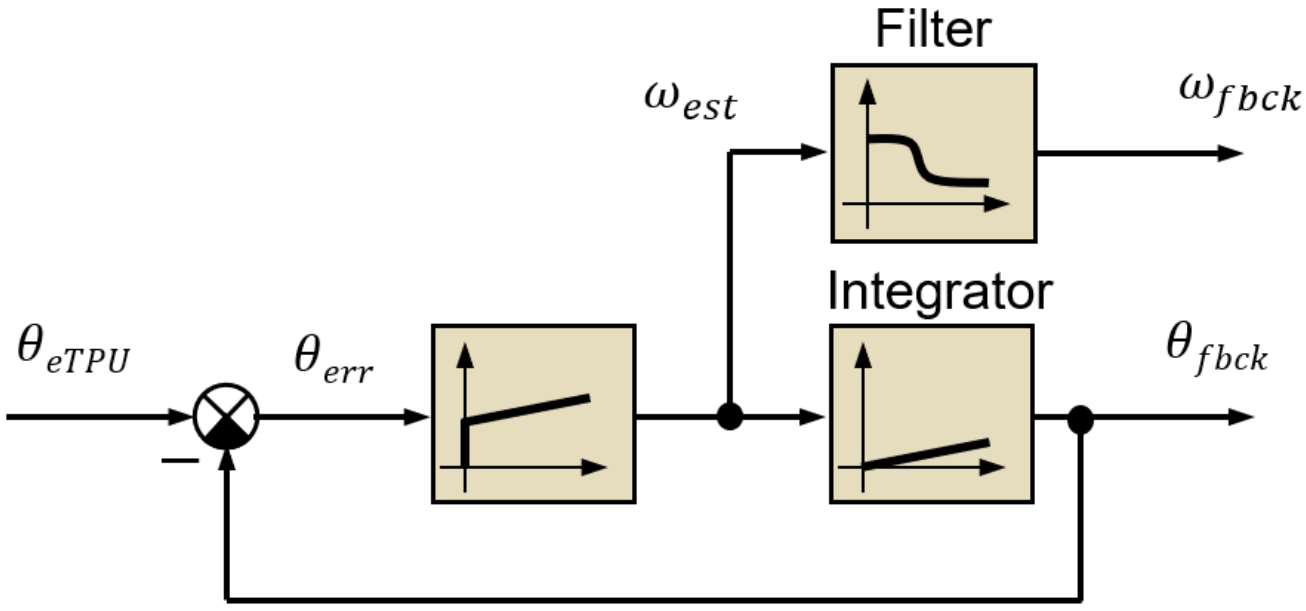

Figure 11. ATO for Resolver systems

The ATO for resolver system is characterized by the position error calculation. The observer error corresponds to the equation 9:

$$\sin(\theta_r)\cos(\theta_{est}) - \sin(\theta_{est})\cos(\theta_r) = \sin(\theta_r - \theta_{est}) \quad (9)$$

The ATO PI controller, Integrator, and filter coefficients can be tuned by the MCAT tool. The ATO function is a member of the motor control SW library (see References) and is available in AMMCLIB Integration.

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

Note: *MPC5775E uses eTPU for resolver feedback signal demodulation. eTPU-based resolver to digital converter is described in Software implementation on the MPC5777E.*

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. Up to the base speed, sufficient voltage is available from the inverter in the operation. Beyond the base speed, motor voltages u_d and u_q are limited and cannot be increased because of the ceiling voltage given by the inverter. Base speed defines the rotor speed at which the BEMF reaches the maximal value, and the motor still produces the maximal torque.

The phase current flow is limited as the difference between the induced BEMF and the supply voltage decreases. Hence, the i_d and i_q of the current cannot be controlled sufficiently. Further speed increase eventually results in a BEMF voltage equal to the limited stator voltage, which means a complete loss of current control. The only way to retain the current control 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 22.

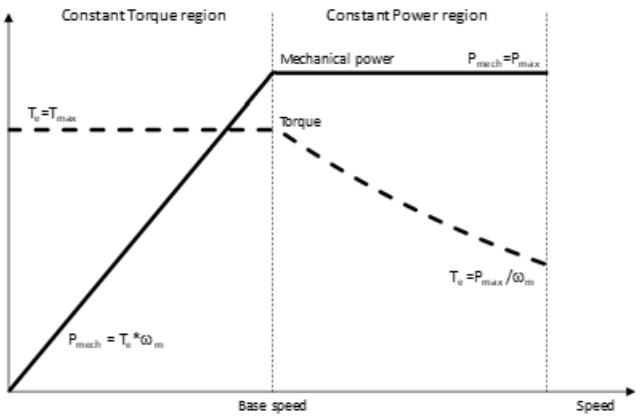

Figure 12. Constant torque/power operating regions

Operation in a 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. A rapid decrease in developed torque characterizes the operation in a constant power region 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 33.

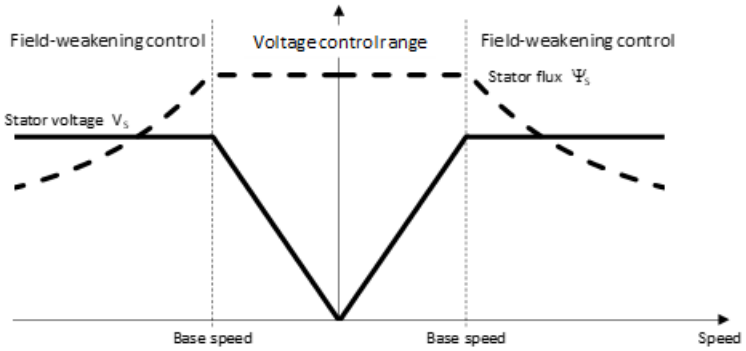

Figure 13. Constant flux/voltage operational regions

FOC splits phase currents into the q-axis torque component and d-axis flux component. The flux current component $I_d$weakens the stator magnetic flux linkage Ψ_S. Reduced stator flux Ψ_S yields to lower BEMF, and the condition of field weakening is met. For more details, see the following phasor diagrams of the PMSM motor operated, exposing FOC control without (left) and with FW (right), as shown in Figure 44.

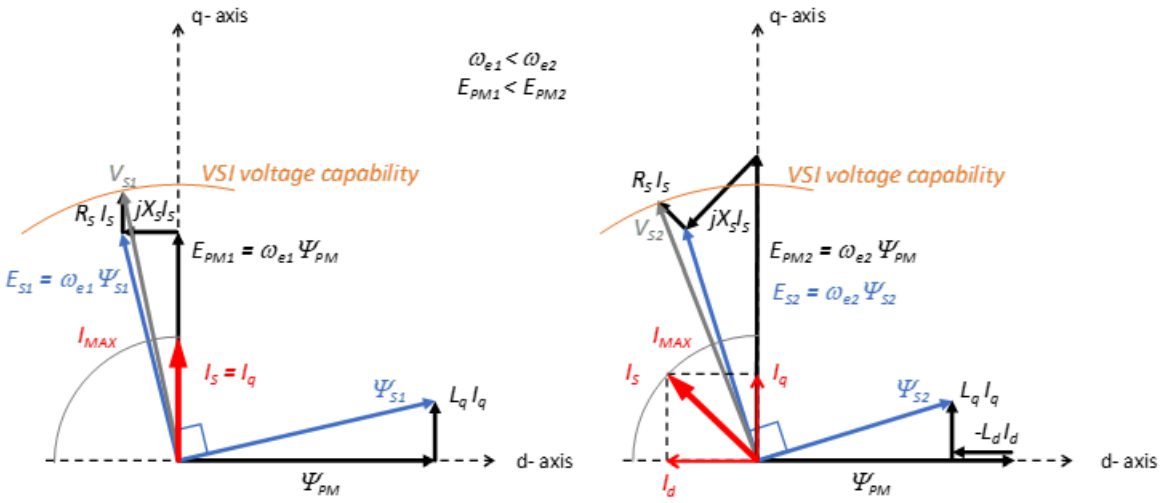

Figure 14. Steady-state phasor diagram of PMSM operation up to base speed (left) and above speed (right)

FOC without FW is operated, demanding the d-axis current component to be zero ($I_d = 0$) to excite the electric machine by permanent magnets mounted on the rotor. $I_d = 0$ represent an operation within a constant torque region (see Figure 12), since the whole amount of the stator current consists of the torque-producing component I_q only (see Figure 14 left). Stator magnetic flux linkage Ψ_{S1} comprises rotor magnetic flux linkage Ψ_{PM}, which represents the major contribution and small amount of the magnetic flux linkage in q-axis $L_q I_q$ produced by q-axis current component I_q. Based on Faraday's law, rotor magnetic flux linkage Ψ_{PM} and stator magnetic flux linkage $\Psi S1$ 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 14 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. Therefore, 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 $-L_d I_d$ component as shown in Figure 14 right. The field weakening approach reduces the BEMF voltage induced in the stator windings E_{S2} 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 a constant power region (see Figure 12), where I_q current is limited due to I_s current vector size limitation (see Figure 15 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 $\Psi_q = L_q I_q$ produced by q-axis current component I_q and magnetic flux linkage in d-axis $\Psi_d = -L_d I_d$ produced by negative d-axis I_d current component that counteracts to Ψ_{PM}.

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 15 left.
- Phase current amplitude *i_max* is limited by the capabilities of power devices and motor thermal design as shown in Figure 15 right.
- Flux linkage in the d-axis is limited to prevent demagnetization of the permanent magnets.

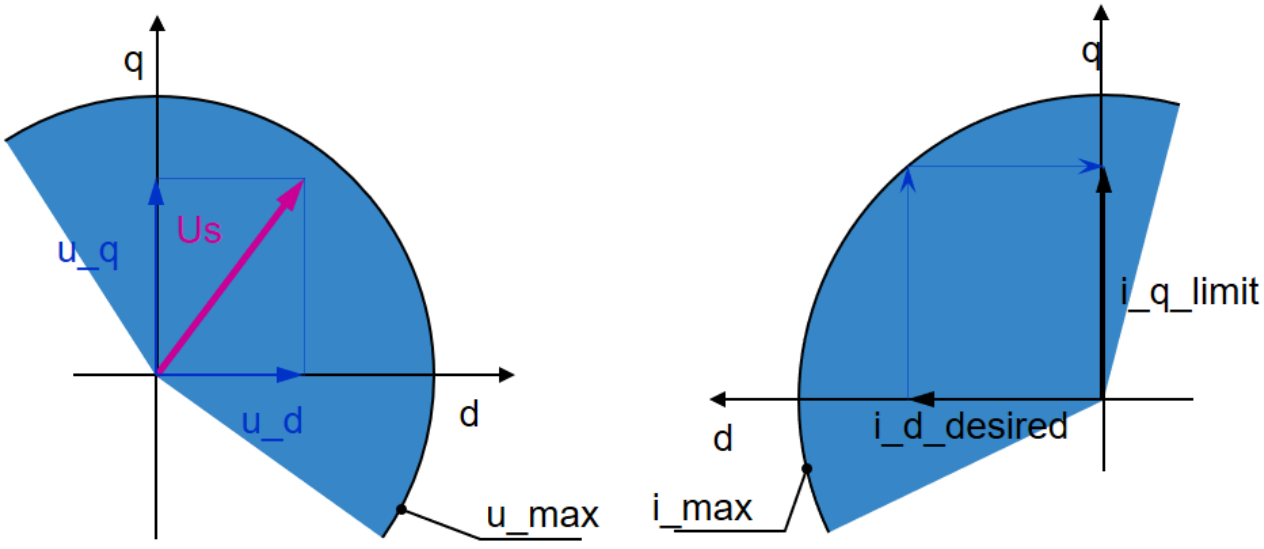

Figure 15. Voltage (left) and current (right) limits for PMSM drive operation

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

4 Software implementation on the MPC5777E

4.1 eTPU

The Enhanced Time Processing Unit (eTPU) is a programmable I/O controller with its core and memory system, allowing it to perform complex timing and I/O management independently of the CPU. The eTPU is used as a coprocessor, specialized for advanced timing functions, such as managing complex engine control, motor control, and communication tasks independently of the CPU.

A new complex library of eTPU functions enabling the eTPU to drive motor control applications is developed. Motor control eTPU library represents a step forward compared to its predecessor, the motor control function sets (set 3 and set 4). The new Motor control eTPU library benefits from NXP eTPU development tools from CodeWarrior. The eTPU(2) Development Suite is based on the Eclipse IDE and includes the C, assembly compiler, simulator, and debugger.

4.1.1 eTPU PWMM

The Motor Control PWM eTPU function (PWMM) uses three eTPU channels to generate three PWM output signals or six eTPU channels to generate three complementary PWM output signal pairs to drive a 3-phase electrical motor. One extrachannel PWMM Master synchronizes all the outputs and is responsible for all the necessary calculations. The master channel does not generate any PWMM output, but the output is used for debugging and visualization of PWMM function processing. Figure 16 shows an example of eTPU PWMM function.

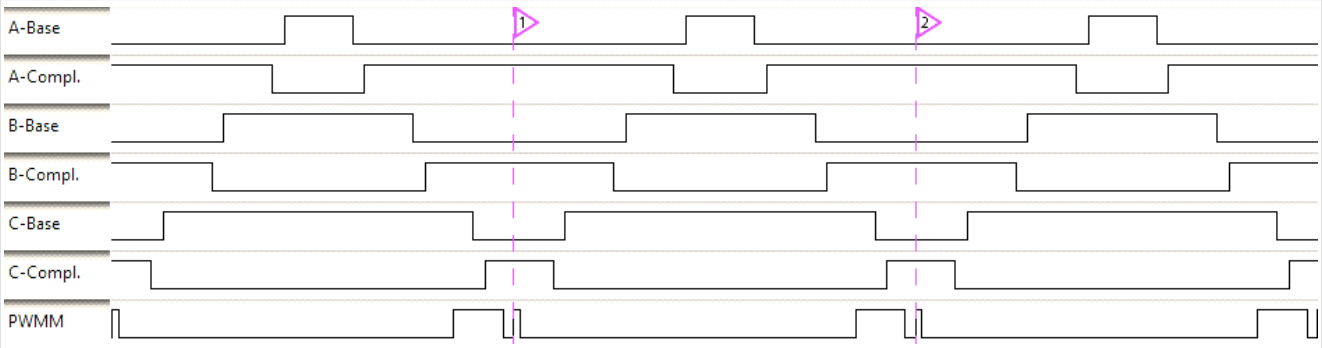

Figure 16. Center aligned 3-phase PWM output with complementary channels and one Master channel (PWMM).

Features:

- Generates 3 phases of PWM signals to drive an electrical motor.
- Based on the selected phase type, either single PWM outputs or complementary PWM pairs with dead time are generated for each motor phase.
- The PWM polarity can be separately configured for the base and the complementary PWM outputs.
- The synchronous update of all PWM phases can happen either once or twice per PWM period:
  - Frame update
  - Frame and center update (half-cycle update)
- Transform the PWMM inputs into PWM output duty cycles by a selected modulation. It can be one of:
  - Unsigned voltages
  - Signed voltages
  - Standard space vector modulation
  - Space vector modulation With O_{000} nulls
  - Space vector modulation With O_{111} nulls
  - Inverse Clark Transformation
  - Sine table modulation
- Support four PWM modes. Switching between the PWM modes in the runtime is also supported:
  - Left-aligned
  - Right-aligned
  - Center-aligned
  - Inverted center-aligned
- The PWM period can be changed in runtime. The new period value is always applied at the frame update only, not at the center update.
- Generation of short pulses can be limited by a minimum pulse width – a threshold for pulse deletion.

4.1.2 eTPU based Resolver to digital converter (RDC)

The Resolver Digital Interface eTPU function (RESOLVER) uses one eTPU channel to generate a 50% duty cycle PWM output signal to be passed through an external low-pass filter and used as a resolver excitation signal. In the resolver position sensor, this excitation signal is modulated by the sine and cosine of the actual motor angle. On-chip ADC samples the feedback Sine and Cosine signals, and the conversion results can be transferred to eTPU DATA RAM by eDMA. Then, the eTPU function RESOLVER can process the digital samples of resolver output signals. Motor angular position, angular speed, a revolution counter, and diagnostics are the results of the Sine and Cosine feedback signal processing (see Figure 17).

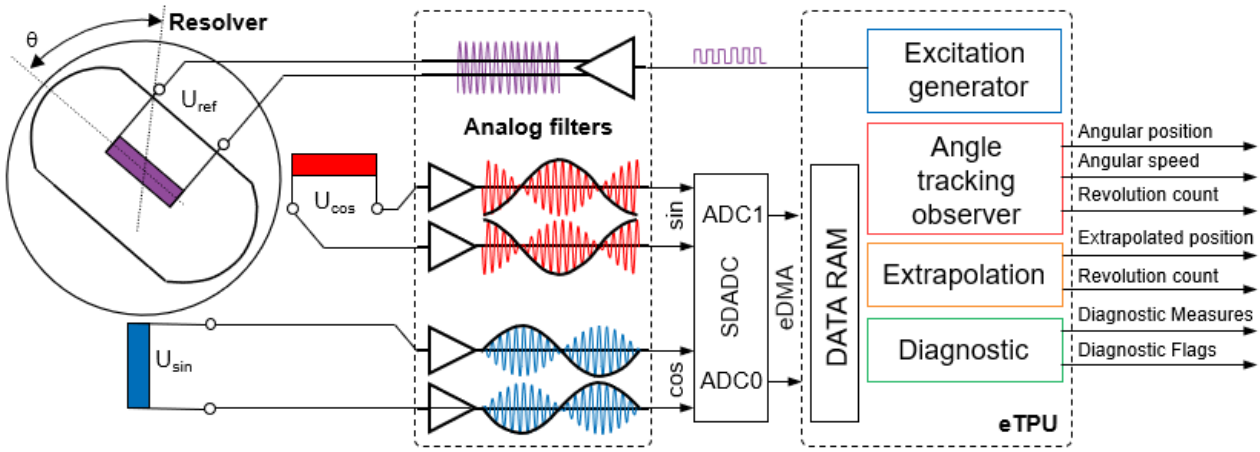

Figure 17. eTPU Resolver Digital Interface block diagram

Processing of the feedback signals is executed on a separate channel. Another channel performs linear extrapolation of the last updated position from ATO to any other time. Extrapolation is an important feature since ATO updates come with a certain period (~50 µs), most likely not aligned with control loop frequency.

Optionally, another eTPU channel can be used to process diagnostics either on the same eTPU engine after the feedback signal processing is finished or on the other eTPU engine parallel to the motor angle and speed calculation. Parallelization enables the CPU application to read the new motor angle and simultaneously check the diagnostic results to ensure the correct motor angle.

The Sine and Cosine analog feedback signals should be converted to a digital representation and transferred to eTPU data RAM. Conversion and transfer should be done independently of the CPU using an on-chip ADC and eDMA. Although any of the ADC modules can be used, the described configuration adopts the sigma-delta ADC (SDADC).

Two SDADC modules continuously sample the Sine and Cosine signals in parallel (see Figure 18). They are configured to obtain 32 samples of each signal per period instead of one sample at the presumed peak, as it is implemented in most of the SW resolver applications. Oversampling method, together with demodulation and filtration, brings more robustness toward the induced noise. Furthermore, the position is evaluated twice per resolver excitation period.

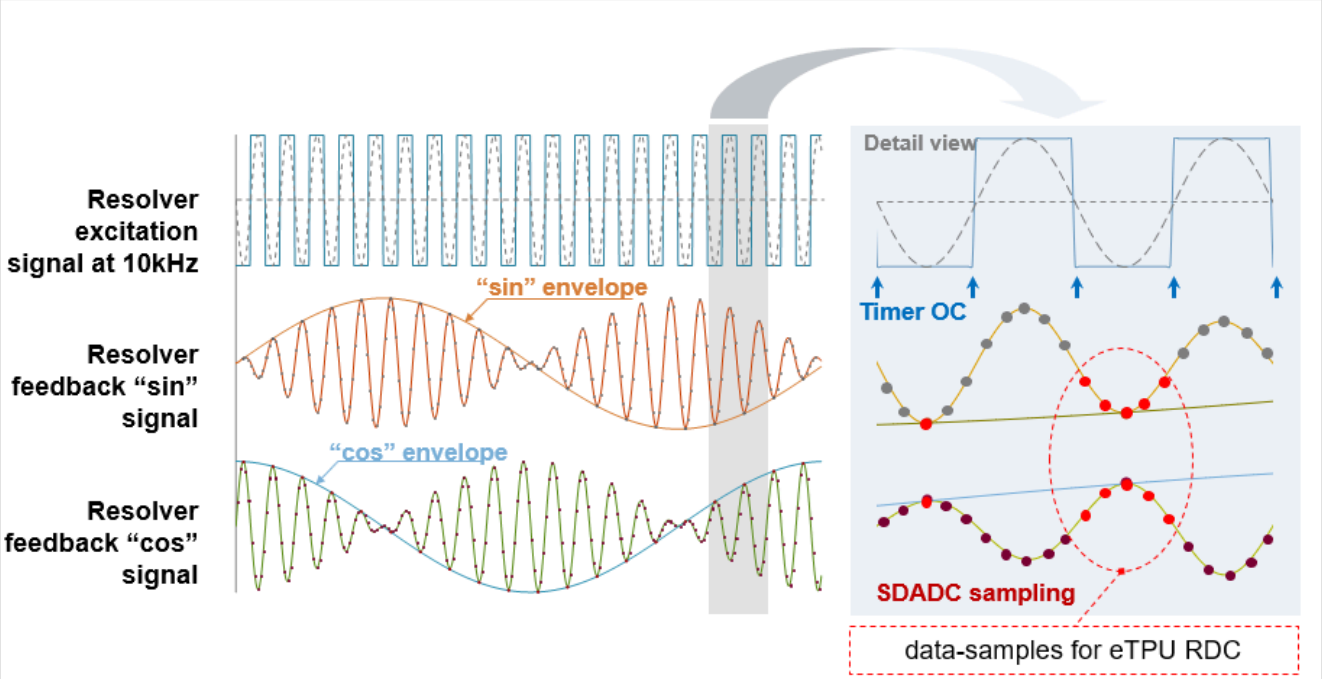

Figure 18. Oversampling and demodulation of Resolver feedback signals

4.1.3 eTPU Analog Sensing Function (AS)

The Analog Sensing eTPU function (AS) uses one eTPU channel to generate adjustable ADC trigger pulses. On the selected eTPU channels, the trigger signal generated by the AS function can internally route to an ADC module. Using eDMA, the A/D conversion results can move back to the eTPU data RAM for further processing by the AS eTPU function. Those preprocessed analog samples are then available for consequential processing, for example, an eTPU function handling the closed-loop motor control. It can run independently with a given period (periodic mode) or be synchronized with any other eTPU function (Synchronized mode).

Features:

- Generates one or two adjustable trigger pulses per period:
  - Frame pulse
  - Center pulse
- Generates interrupts, DMA requests, and eTPU links at none, one, or more of the selected time positions:
  - Frame pulse start
  - Frame pulse end
  - Center pulse start
  - Center pulse end
- Preprocess a defined number of analog signals (ADC conversion results) using:
  - Gain
  - DC-offset
  - Forgetting factor (low-pass EWMA filter).
- Supports processing phase currents – calculation of one-phase current value using the other two-phase current values based on an SVM sector value.
- When working with eQADC, it supports CPU-independent modification of command queues.

4.2 MPC5777E – Key modules for PMSM FOC control

The key module for motor control on the MPC5775E device is the Enhanced Time Processing Unit (eTPU). Using the eTPU Motor control library function set, this programmable timer coprocessor can generate 3-phase PWM complementary output, analog signal measurement triggering and synchronization signal, and Resolver feedback signal processing. To enable full eTPU Resolver functionality, using sigma-delta analog-to-digital converters and the Enhanced Direct Memory Access module is crucial. The Enhanced Queued Analog to Digital Converters are used for phase current sensing.

Figure 19 shows module interconnection for a typical PMSM FOC application working in sensor-based mode using LEM current sensing with eTPU Motor Control Library functions involvement.

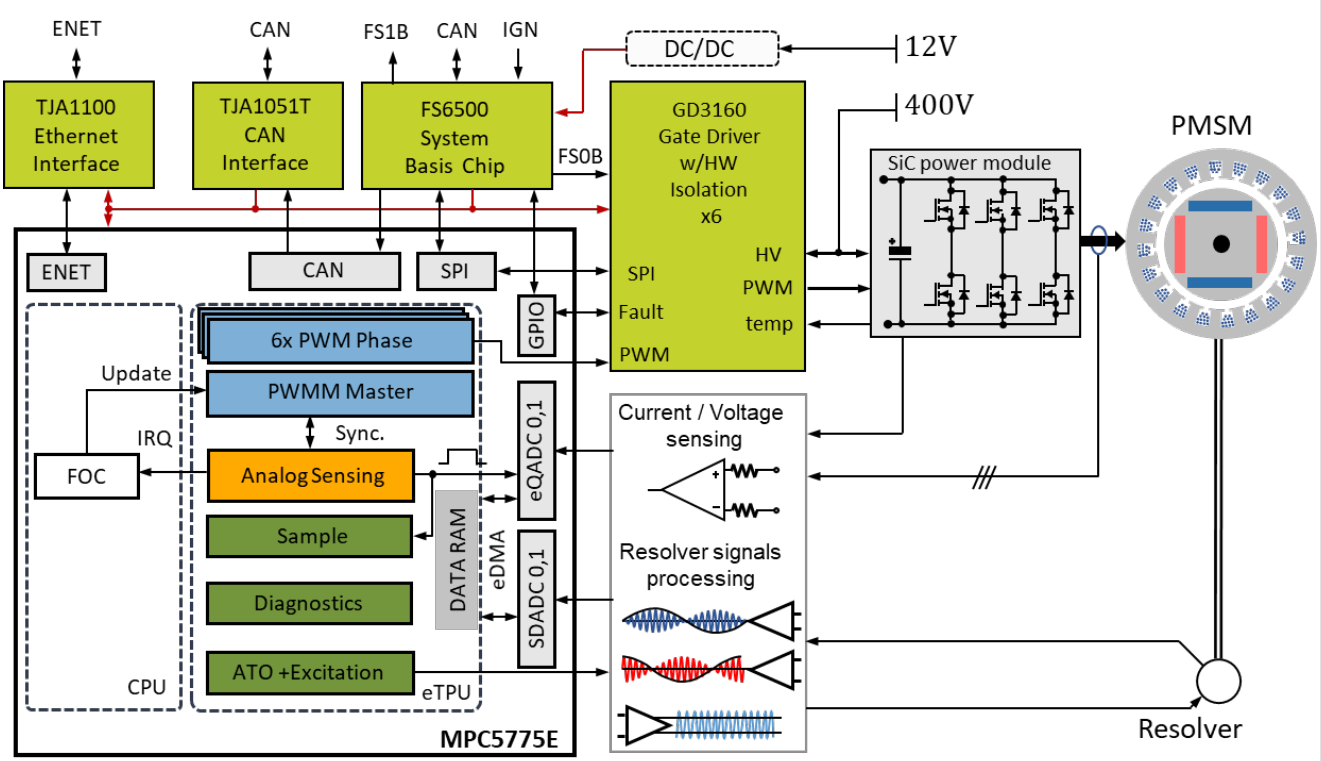

Figure 19. MPC5775E module interconnection block diagram

4.2.1 GD3160 advanced high voltage-isolated gate driver

The GD3160 is an advanced, single-channel gate driver for IGBTs and SiC power devices. Integrated Galvanic isolation and low on-resistance drive transistors provide high charging and discharging current, low dynamic saturation voltage, and rail-to-rail gate voltage control. Current and temperature sense minimizes IGBT/SiC stress during faults. Accurate and configurable undervoltage lockout (UVLO) protects while ensuring sufficient gate-drive voltage headroom. The GD3160 autonomously manages severe faults and reports faults and status via INTB and/or INTA pins and an SPI interface. It can directly drive the gates of most IGBTs and SiC MOSFETs. Self-test, control, and protection functions are included in the design of high-reliability systems (ASIL C/D). The GD3160 meets the stringent requirements of automotive applications and is fully AEC-Q100 grade 1 qualified.

Configuration of the GD3160 predriver is performed via SPI commands. The GD3160 allows different operating modes to be set and locked using SPI commands. All SPI commands have a CRC, which ensures each command's integrity. The GD3160 validates all commands that it receives for a correct CRC. The responses

received from the gate driver by the MCU are validated in the application software. Commands with invalid CRCs are ignored.

Initialization is performed in the GD31xxInit() routine, which can be found in the BSW/Drv/gd/GD31xx_dc.c file. The first task performed in this routine is setting the GD3160 configuration parameters in configuration register variables. Each parameter has a set of predefined constants that cover all possible values for the parameter. Information on each parameter can be found in the GD3160 data sheet. The values of all parameters are set in the application software and are optimized for the PIM hardware configuration. After the parameter values are set, the GD31xxInit() routine puts the GD3160 into configuration mode so all parameters can be written to the gate drivers. Before writing the parameters to the device, the initialization routine clears all faults that may be present. The parameters are then written to all devices, and then a command is set to take the GD3160s out of configuration mode. The parameters are not changed during operation. The parameters must be written to the gate driver every time the application starts because it has no nonvolatile memory.

SPI commands also report the condition of the GD3160 based on the internal monitoring circuits and fault detection logic. The MPC5777E detects the fault state of the GD3160 through the INTB fault signal, which triggers an interrupt routine if a fault occurs. Once a fault occurs, a fault condition sets, and the source of the fault is determined by reading two status registers (STATUS1 and STATUS2) using the SPI interface. There are a total of ten bits in each status register. The fault indicated by each bit can be found in the GD3160 data sheet.

The GD3160 also monitors the temperature of the SiC power device and its internal die temperature. The temperatures are monitored by sending periodic SPI commands, which return values that are converted into temperatures. Warning and shutdown temperature limits can be set for the gate driver's internal die temperature. The temperature of the SiC device is monitored by the application software, which can shut down the inverter to prevent damage to the SiC power device.

4.2.2 Module involvement in PMSM FOC control

This section discusses timing and module synchronization to accomplish PMSM FOC on the MPC5777E and the internal hardware features.

The following figure shows the time diagram of the automatic synchronization between PWM and ADC in the PMSM application.

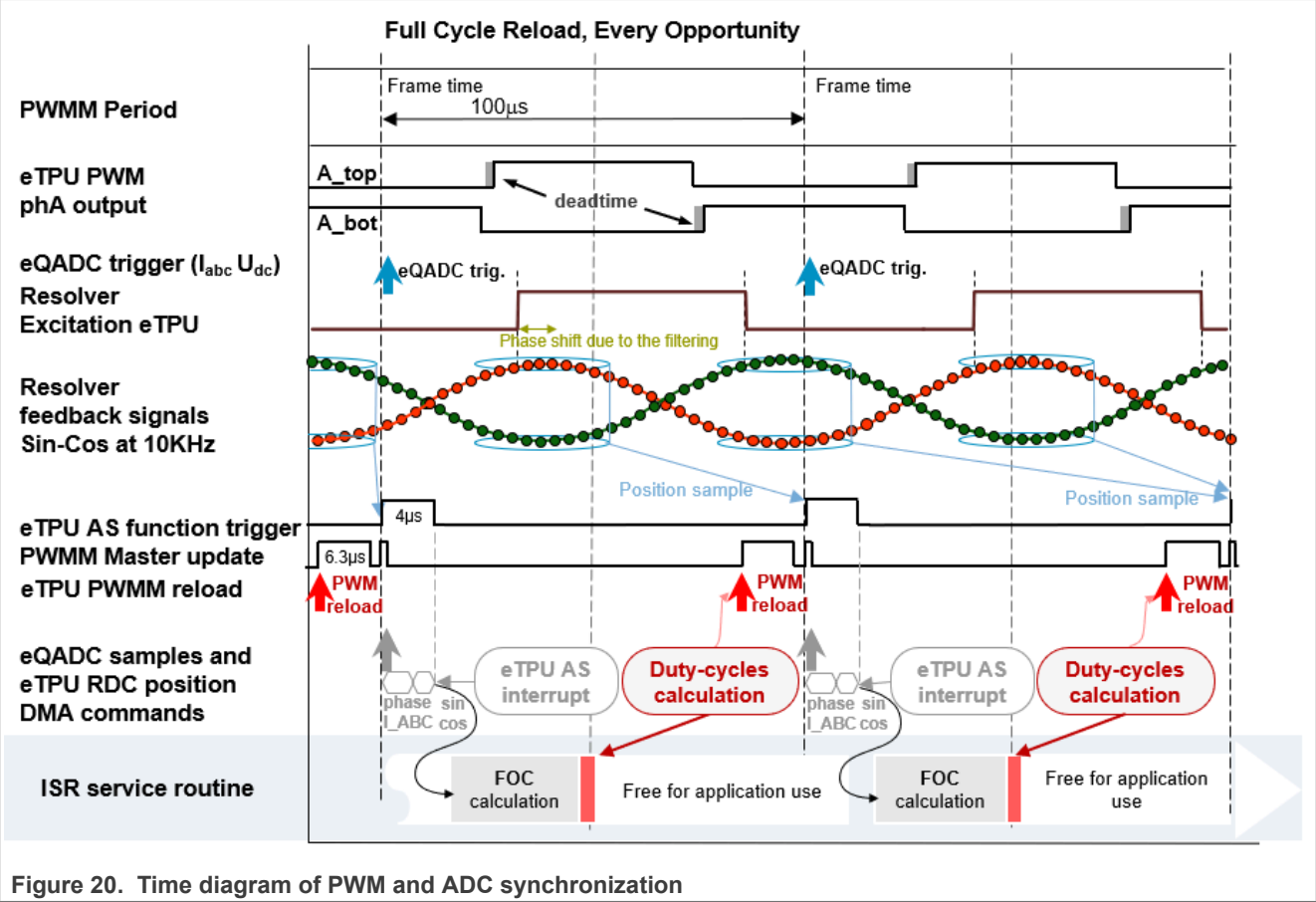

Figure 20. Time diagram of PWM and ADC synchronization

The PMSM FOC control with LEM current measurement is based on static timing, meaning the ADC conversions' trigger point instances are at the same place within one control loop cycle.

Each control cycle starts at the beginning of the PWM cycle, as shown in Figure 20. The PWMM function runs with a certain period that corresponds to the control loop frequency. AS function is configured to run in sync with PWMM and generates a trigger pulse at the beginning of the PWMM period. Pulse rising edge triggers eQADCs to sample phase currents and DC bus voltage. Phase currents are measured simultaneously. The rising edge of the pulse also triggers Resolver position sampling. The pulse width is configurable and is configured for 4 µs. The falling edge of the AS pulse triggers an IRQ where the FOC is calculated. The result of the FOC calculation is new duty cycles that have to be applied for the next period. To apply the newly updated duty cycles for the next period, the update (new duty cycle inputs) has to come before the “update time.” Update time is the time configured in the PWMM function needed for applying duty cycles by the eTPU PWMM function for the next period (see Figure 21). If new inputs come after the “update time” happens, then new inputs are not applied for the next period and is applied in the second next period. In that case, the PWMM function does not stop outputting the PWM signal and continues with the last duty cycle values configured. Missing update IRQ is generated from the eTPU PWMM Master channel if no new input comes before the update time threshold.

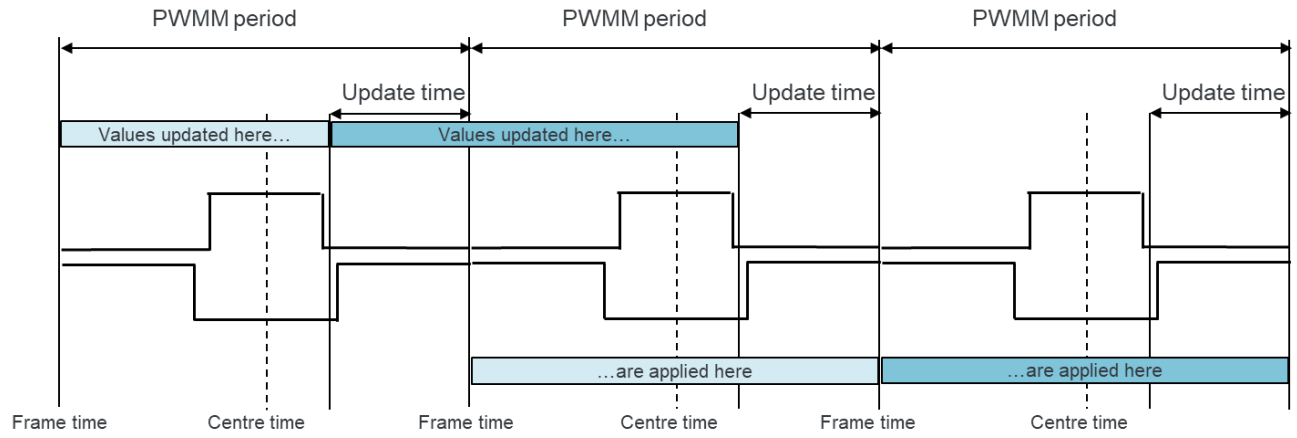

Figure 21. PWMM update input values for frame update

4.3 MPC5775E device initialization

An embedded part of the PMSM motor control application has been created to simplify and accelerate application development. Peripherals are initialized at the beginning of the *main()* function. For each MPC5775E module, there is a specific configuration function:

- SysClk_Init();
- GPIO_Init();
- STM_Init(); // system timer module inning for delay functions
- FCCU_clear_faults();, FCCU_Init(); // FCCU initialization
- pim_system_etpu_init(); // initialization eTPU (loading eTPU code and configuration)
- DMA_Init(); // init DMA channels for from eQADCs (sending commands, transferring measured data)
- rsvlr_edma_init(); //Init DMA channels for SDADC (transferring data in to eTPU memory)
- SDADC_data_streaming_init(); // resolver data flow streaming for debugging
- eqADC_Init(); // eQADC initialization
- sdADC_Init(); //SDADC initialization
- SPI_Init(); // SPI initialization for GD31xx
- DSPI_Init(0, 1, 260000000/4, 1000000, 0, 16); // SPI initialization for FS65
- FS65_Init(); //initialization FS65
- FS65_IsrPIT_WD(); // refresh FS65 watchdog
- CAN_Init(); // Initialization CAN communication for FreeMASTER
- Int_Init(); // interrupt initialization
- pim_system_etpu_start(); // start eTPU engines
- M_GPIO_Set(204, 1); // Enable Bottom Flyback
- M_GPIO_Set(432, 1); // Enable Top Flyback
- Configure fail-safe logic (SCH-45863 schematic sheet: 8/12)
  - M_GPIO_Set(79, 1); // FSE fail state deactivate (schematic sheet)
  - M_GPIO_Set(221, 0); // High side safe state configuration FSH set low
  - M_GPIO_Set(222, 1); // Low side safe state configuration FSL set low (reverse logic, refer to schematic)
- PIT_Init(); //configuring timer for SF65 watchdog refresh
- GD31xxInit(); // Gate drivers configuration
- FMSTR_Init(); // FreeMASTER initialization
- MCAT_Init(); //MCAT initialization

- resolver_start(); // Resolver checker

4.3.1 Clock configuration

The clock architecture of the MPC5775E device contains two clock domains from separate PLLs. The first one is the system clock for the cores, cross-bar switch, peripheral bridges, memories, debug logic, and memory-mapped portion of peripherals. The second one feeds the state machines and protocol engines of communication and timer peripheral modules. These two clock domains can be configured to be completely asynchronous between each other.

The PLL0 module is supplied from a 40 MHz external crystal oscillator (XOSC). PLL0 circuit multiplies the input XOSC frequency by 10 and divides by 2 for the PHI output to be 200 MHz, further used as an eTPU clock, SDADC clock, and further divided by 2 used as a peripheral clock. The second output of PLL0 PHI1 is divided by 4 to be 50 MHz, further used as an input clock for the PLL1 module.

The PLL1 module generates a 260 MHz system clock and a 130 MHz platform clock. The clocking configuration within this application is listed in the following table.

Table 1. MCP5775E clocking configuration

Clock	Frequency
core_clk	260 MHz
plat_clk	130 MHz
etpu_clk	200 MHz
per_clk	100 MHz

The configuration of the overall system clock is performed within a function call *SysClk_Init ()*, see the following code example:

```
void SysClk_Init(void)
{
  SIU.SYSDIV.B.SYSCLKSEL = 0;    //run system clock from IRC
  PLLDIG.PLL0CR.B.CLKCFG = 0;    //disable PLL0
  PLLDIG.PLL1CR.B.CLKCFG = 0;    //disable PLL1
  SIU.SYSDIV.B.PLL0SEL = 0x00;    //connect XOSC to the PLL0 input
  SIU.SYSDIV.B.PLL1SEL = 0x01;    //connect PLL0-PHI1 to the PLL1 input
// Set PLL0 to 200 MHz with 40MHz XOSC reference
  PLLDIG.PLL0DV.R = 0x5002100A;            // PREDIV =  1, MFD = 10, RFDPHI = 2,
 RFDPHI1 = 10
while(SIU.RSR.B.XOSC==0){};    //wait for stable XOSC
  PLLDIG.PLL0CR.B.CLKCFG = 3;    //turn on PLL0
while(PLLDIG.PLL0SR.B.LOCK==0){};    //wait for PLL0 lock
// Set PLL1 to 260 MHz with 40MHz PLL0-PHI1 reference
  PLLDIG.PLL1DV.R = 0x0002001A;             // MFD = 26, RFDPHI = 2
  PLLDIG.PLL1CR.B.CLKCFG = 3;    //turn on PLL1
while(PLLDIG.PLL1SR.B.LOCK==0){};          //wait for PLL1 lock
// Select clock dividers and sources
//    PLL0 SEL = 0 … XOSC selected
//    PLL1 SEL = 1 … PLL0-PHI1 selected
//    PERCLKSEL = 1 … PLL0 selected
//    FMPERDIV = 0 … div by 2
//    PERDIV = 0 … div by 2
//    MCALSEL = 0 … XOSC selected
//    SYSCLKSEL = 2 … PLL1
//    ETPUDIV = 1 … div by 1
//    SYSCLKDIV = 4 … div by 1
//    PCS = 0 … Disabled
```

```
  SIU.SYSDIV.R = 0x05002110;
//    ENGDIV = 16 … div by 32
//    ECCS = 0 … PLAT_CLK selected
//    EBDF = 1 … div by 2
  SIU.ECCR.R = 0x00001001;
//    SDDIV = 12 … div by 13
  SIU.SDCLKCFG.R = 0x0000000C;
//    LFCLKSEL = 0 … XOSC selected
//    LFDIV = 1 … div by 2
  SIU.LFCLKCFG.R = 0x00000001;
//   PSDIV = 1 … div by 2
//   PSDIV1M = 199 … div by 200
  SIU.PSCLKCFG.R = 0x000100C7;
}
```

4.3.2 CAN configuration

Controller Area Network CAN module is used for communication between the MPC5775E MCU and FreeMASTER runtime debugging and visualization tool. Function *CAN_Init()* configures module A with a baud rate of 500 kB. See the code example below.

```
void CAN_Init() {
uint32_t *pointer=0;
//enable FlexCAN
CAN_A.MCR.B.MDIS = 0x0;
while (CAN_A.MCR.B.MDIS);
//XOSC clock selected (40 MHz)
CAN_A.CTRL1.B.CLKSRC = 0x0; //0x0
//baud rate 500kB/s
CAN_A.CTRL1.B.PROPSEG = 0x3;
    CAN_A.CTRL1.B.PSEG1 = 0x2;
CAN_A.CTRL1.B.PSEG2 = 0x1;
CAN_A.CTRL1.B.PRESDIV = 0x7;        // 500kB
//CAN SRAM initialization
CAN_A.CTRL2.B.WRMFRZ = 0x1;         //enable write into the CAN SRAM memory
for (pointer=(uint64_t*)(0xFFFC0000 + 0x080);pointer<0xFFFC0AE0;pointer++)
 *pointer = 0x0;
CAN_A.RXMGMASK.R = 0x1FFFFFFF;;
CAN_A.RX14MASK.R = 0x1FFFFFFF;;
CAN_A.RX15MASK.R = 0x1FFFFFFF;;
CAN_A.RXFGMASK.R = 0x1FFFFFFF;;
CAN_A.CTRL2.B.WRMFRZ = 0x0;         //disable write into the CAN SRAM memory
CAN_A.MCR.B.RFEN = 0x1;
CAN_A.MCR.B.FRZ = 0x0;
CAN_A.MCR.B.HALT = 0x0;
}
```

4.3.3 eTPU configuration

eTPU function configuration is performed within so-called GCT files (etpu_gct.c/.h). First, there is a general eTPU engine configuration for both engines A and B, then eTPU functions are assigned to selected channels, and those channels are configured for that particular eTPU functionality. The parameters for eTPU engine configuration are held in the my_etpu_config structure. See part of the structure in the example below.

```
/struct etpu_config_t my_etpu_config =
{
/* etpu_config.mcr - Module Configuration Register */
```

```
  FS_ETPU_GLOBAL_TIMEBASE_DISABLE  /* keep time-bases stopped
 during intialization (GTBE=0) */
  | FS_ETPU_MISC_DISABLE, /* SCM operation disabled (SCMMISEN=0) */
/* etpu_config.misc - MISC Compare Register*/
  FS_ETPU_MISC, /* MISC compare value from etpu_set.h */
/* etpu_config.ecr_a - Engine A Configuration Register */
  FS_ETPU_ENTRY_TABLE_ADDR /* entry table base address = shifted
 FS_ETPU_ENTRY_TABLE from etpu_set.h */
| FS_ETPU_CHAN_FILTER_2SAMPLE /* channel filter mode = three-sample mode
 (CDFC=0) */
  | FS_ETPU_FCSS_DIV2 /* filter clock source selection = div 2 (FSCC=0) */
  | FS_ETPU_FILTER_CLOCK_DIV2 /* filter prescaler clock control = div 2
 (FPSCK=0) */
| FS_ETPU_PRIORITY_PASSING_ENABLE /* scheduler priority passing is enabled
 (SPPDIS=0) */
  | FS_ETPU_ENGINE_ENABLE, /* engine is enabled (MDIS=0) */
/* etpu_config.tbcr_a - Time Base Configuration Register A */
FS_ETPU_TCRCLK_MODE_2SAMPLE /* TCRCLK signal filter control mode = two-sample
 mode (TCRCF=0x) */
| FS_ETPU_TCRCLK_INPUT_DIV2CLOCK /* TCRCLK signal filter control clock = div 2
 (TCRCF=x0) */
    | FS_ETPU_TCR1CS_DIV1 /* TCR1 clock source = div 1 (TCR1CS=1)*/
    | FS_ETPU_TCR1CTL_DIV2 /* TCR1 source = div 2 (TCR1CTL=2) */
    | FS_ETPU_TCR1_PRESCALER(1) /* TCR1 prescaler = 1 (TCR1P=0) */
    | FS_ETPU_TCR2CTL_DIV8 /* TCR2 source = etpuclk div 8 (TCR2CTL=4) */
    | FS_ETPU_TCR2_PRESCALER(1) /* TCR2 prescaler = 1 (TCR2P=0) */
    | FS_ETPU_ANGLE_MODE_DISABLE, /* TCR2 angle mode is disabled (AM=0) */
/* etpu_config.stacr_a - Shared Time And Angle Count Register A */
  FS_ETPU_TCR1_STAC_ENABLE /* TCR1 on STAC bus = enabled (REN1=1) */
  | FS_ETPU_TCR1_STAC_SERVER /* TCR1 resource control = server (RSC1=1) */
  | FS_ETPU_TCR1_STAC_SRVSLOT(0) /* TCR1 server slot = 0 (SRV1=0) */
  | FS_ETPU_TCR2_STAC_DISABLE /* TCR2 on STAC bus = disabled (REN2=0) */
| FS_ETPU_TCR1_STAC_CLIENT /* TCR2 resource control = client (RSC2=0) */
  | FS_ETPU_TCR2_STAC_SRVSLOT(0), /* TCR2 server slot = 0 (SRV2=0) */
/* etpu_config.ecr_b - Engine B Configuration Register */
FS_ETPU_ENTRY_TABLE_ADDR /* entry table base address = shifted
 FS_ETPU_ENTRY_TABLE from etpu_set.h */
| FS_ETPU_CHAN_FILTER_2SAMPLE /* channel filter mode = three-sample mode
 (CDFC=0) */
  | FS_ETPU_FCSS_DIV2 /* filter clock source selection = div 2 (FSCC=0) */
  | FS_ETPU_FILTER_CLOCK_DIV2 /* filter prescaler clock control = div 2
 (FPSCK=0) */
| FS_ETPU_PRIORITY_PASSING_ENABLE /* scheduler priority passing is enabled
 (SPPDIS=0) */
  | FS_ETPU_ENGINE_ENABLE, /* engine is enabled (MDIS=0) */
/* etpu_config.tbcr_b - Time Base Configuration Register B */
  FS_ETPU_TCRCLK_MODE_2SAMPLE /* TCRCLK signal filter control mode = two-sample
 mode (TCRCF=0x) */
  | FS_ETPU_TCRCLK_INPUT_DIV2CLOCK /* TCRCLK signal filter control clock = div 2
 (TCRCF=x0) */
  | FS_ETPU_TCR1CS_DIV1 /* TCR1 clock source = div 1 (TCR1CS=1)*/
  | FS_ETPU_TCR1CTL_DIV2 /* TCR1 source = div 2 (TCR1CTL=2) */
  | FS_ETPU_TCR1_PRESCALER(1) /* TCR1 prescaler = 1 (TCR1P=0) */
  | FS_ETPU_TCR2CTL_DIV8 /* TCR2 source = etpuclk div 8 (TCR2CTL=4) */
  | FS_ETPU_TCR2_PRESCALER(1) /* TCR2 prescaler = 1 (TCR2P=0) */
  | FS_ETPU_ANGLE_MODE_DISABLE, /* TCR2 angle mode is disabled (AM=0) */
/* etpu_config.stacr_b - Shared Time And Angle Count Register B */
  FS_ETPU_TCR1_STAC_ENABLE /* TCR1 on STAC bus = enabled (REN1=1) */
| FS_ETPU_TCR1_STAC_CLIENT /* TCR1 resource control = client (RSC1=0) */
  | FS_ETPU_TCR1_STAC_SRVSLOT(0) /* TCR1 server slot = 0 (SRV1=0) */
```

```
  | FS_ETPU_TCR2_STAC_DISABLE /* TCR2 on STAC bus = disabled (REN2=0) */
  | FS_ETPU_TCR1_STAC_CLIENT /* TCR2 resource control = client (RSC2=0) */
  | FS_ETPU_TCR2_STAC_SRVSLOT(0), /* TCR2 server slot = 0 (SRV2=0) */
/* etpu_config.wdtr_a - Watchdog Timer Register A(eTPU2 only) */
FS_ETPU_WDM_DISABLED /* watchdog mode = disabled */
  | FS_ETPU_WDTR_WDCNT(0), /* watchdog count = 0 */
/* etpu_config.wdtr_b - Watchdog Timer Register B (eTPU2 only) */
  FS_ETPU_WDM_DISABLED /* watchdog mode = disabled */
  | FS_ETPU_WDTR_WDCNT(0) /* watchdog count = 0 */
};
The entire eTPU initialization and function configuration is then performed
 in my_system_etpu_init() function call. See the example code below.
int32_t pim_system_etpu_init(void)
{
int32_t err_code;
/* Initialization of eTPU DATA RAM */
  fs_memset32((uint32_t*)fs_etpu_data_ram_start, 0, fs_etpu_data_ram_end -
 fs_etpu_data_ram_start);
/* Initialization of eTPU global settings */
err_code = fs_etpu_init(
    my_etpu_config,
    (uint32_t *)etpu_code, sizeof(etpu_code),
    (uint32_t *)etpu_globals, sizeof(etpu_globals));
if(err_code != 0) return(err_code);
#ifdef FS_ETPU_ARCHITECTURE
#if FS_ETPU_ARCHITECTURE == ETPU2
/* Initialization of additional eTPU2-only global settings */
err_code = fs_etpu2_init(
my_etpu_config,
#ifdef FS_ETPU_ENGINE_MEM_SIZE
    FS_ETPU_ENGINE_MEM_SIZE);
#else
    0);
#endif
if(err_code != FS_ETPU_ERROR_NONE) return(err_code);
#endif
#endif
/* Initialization of eTPU channel settings */
err_code = fs_etpu_pwmm_init(&pwmm_instance, &pwmm_config);
if(err_code != FS_ETPU_ERROR_NONE) return(err_code + (ETPU_PWMM_MASTER_CH<<16));
  err_code = fs_etpu_as_init(&as_instance, &pwmm_instance, &as_config);
if(err_code != FS_ETPU_ERROR_NONE) return(err_code + (ETPU_AS_CH<<16));
  err_code = fs_etpu_resolver_init(&resolver_instance, &resolver_config);
if(err_code != FS_ETPU_ERROR_NONE) return(err_code +
 (ETPU_RESOLVER_EXC_CHAN<<16));
return(0);
}
```

4.3.3.1 eTPU PWMM: Center-aligned PWM mode

The Motor Control PWM eTPU function (PWMM) uses either three eTPU channels to generate three PWM output signals or six eTPU channels to generate three complementary PWM output signal pairs to drive a 3-phase electrical motor.

Complementary pairs of PWM are used with center-aligned configuration, frame update only (parameters are updated once per PWM period, on a frame), and signed voltage modulation. The period is configured for 100 µs, dead time configuration is 1 µs, and minimum pulse width. Update time is configured for 7 µs, ensuring enough time to perform updates of all three phases for the next period.

PWMM function configuration is performed within the etpu_gct.c file. Part of the configuration is held in pwmm_instance where PWMM channels are assigned, complementary channel mode, function priority, and polarity of the PWM outputs are configured. Other parameters that can be changed during runtime are held in the pwmm_config structure and can be changed using the fs_etpu_pwmm_config() function call. Parameters are applied in the upcoming PWMM update. The change of parameters is then visible in the next period after the PWMM update. See the timing of the updates shown in Figure 21.

```
struct pwmm_config_t pwmm_config =
{
FS_ETPU_PWMM_FM1_FRAME_UPDATE_ONLY, /**< selection of PWM update position. */
FS_ETPU_PWMM_MODULATION_SIGNED,     /**< Selection of modulation */
FS_ETPU_PWMM_MODE_CENTER_ALIGNED,   /**< PWM Mode selection */
NSEC2TCR1(100000),                  /**< PWM period as a number of TCR1 cycles.
 */
USEC2TCR1(1),                       /**< PWM dead-time as a number of TCR1
 cycles.*/
USEC2TCR1(1),                       /**< Minimum pulse width as number of TCR1
 cycles. */
USEC2TCR1(7)                        /**< A time period (number of TCR1 cycles)
 that is needed to
                                                perform an update of
 all PWM phases. */
};
```

Note: *Use predefined macros to configure the intended functionality. All the possible options are listed in etpu_pwmm.h in the comments and described in the PWMM-doxydoc.chm file. Note that parameters in pwmm_instance can be configured only once at the eTPU initialization and cannot be changed during runtime.*

4.3.3.2 eTPU resolver configuration

The resolver function is designed to integrate with the SDADC modules seamlessly. SDADCs are configured to provide 32 per period of the resolver feedback signal. Considering the resolver excitation frequency to be 10 kHz means a 320 kHz output data rate is required for SDADC. With the given limitations of SDADC clocking and oversampling rate configurations, the closest possible output data rate that can be configured is 320.512 kHz (see Table 2) and output data rate determines the resolver excitation frequency to be 10.016 kHz, 99.84 µs period, respectively. Resolver bandwidth configuration can be changed by modifying ATO P, and I gains. The current configuration of ATO is for 1 kHz bandwidth. Excitation P and I gains are tuned for a balanced phase-shift reaction. Therefore, it is recommended to use this configuration. See eTPU RDC and RDC Checker User Guide (References), for more information about eTPU Resolver implementation and its configuration.

```
struct resolver_config_t resolver_config =
{
  FS_ETPU_RESOLVER_SEMAPHORE_0,
  FS_ETPU_RESOLVER_OPTIONS_CALCULATION_ON +
  FS_ETPU_RESOLVER_OPTIONS_DIAG_MEASURES_ON +
  FS_ETPU_RESOLVER_OPTIONS_EXC_ADAPTATION_ON +
  FS_ETPU_RESOLVER_OPTIONS_EXC_GENERATION_ON, /* Resolver function options */
  NSEC2TCR1(99840),                           /* Excitation signal period */
  SFRACT24(0.070597541),                      /* ATO P-gain */
  SFRACT24(0.002492006),                      /* ATO I-gain */
  SFRACT24(0.00000),                          /* Excitation  P-gain */
  SFRACT24(0.00012),                          /* Excitation  I-gain */
  SFRACT24(0.9)                               /* Speed EWMA filter coefficient
 */
};
```

4.3.3.3 eTPU AS: triggering output pulse

AS function generates periodical trigger pulses, one or two per period, Frame and Center pulse. The position of the pulses (relative to AS period frame time and center time) and the length are configurable.

AS function is configured to work in synchronization mode adopting the period of PWMM function. In that case, AS period is configured to zero, and a pointer to the PWMM period is provided in etpu_as.c file. The polarity of the AS signal is configured as high for the pulse. Those configurations are held in the *as_instace* structure (defined similarly like PWMM in *etpu_gct.c*) and are not a subject of change during the runtime. AS is configured to output one Frame pulse at the beginning of the PWMM period (on a Frame time), and length is configured to be 4 µs which is enough for all the feedback sampling and DMA transfer to eTPU data RAM. The AS pulse is internally gated to eQADC to trigger conversion. On rising edge of the AS pulse link to eTPU Resolver Sample channel is generated to trigger position sampling at the same time as feedback current measurement. IQR is generated at the falling edge of the trigger pulse *eTPU_AS_Isr()*, where all the feedback values are processed, and the state machine is executed.

Also, configuration for measured signals triggered by AS function is present in *etpu_gct.c* in a structure called *as_signal_config*. Structure *as_signal_config*, holds one sub-configuration structure per signal. The number of the signals to be processed by AS is defined in *as_instance* at the AS function initialization. The signal configuration structure holds parameters like gain, DC offset, and filter factor. The processing of the signals may and may not be used. Within this application, the processing of feedback signals is performed by the CPU, not the eTPU AS function.

See the following example code with AS configuration and AS signal configuration structures.

```
/** A structure to represent a single AS signal processing configuration.
 *  It includes both static and dynamic values. */
struct as_signal_config_t as_signal_config[4] =
{
{
0,          /* Result queue offset of the signal */
1,          /* Gain bit */
0x2000,     /* DC offset */
SFRACT24(0) /* Forget factor of the EWMA filter */
},
…
};
/** A structure to represent a configuration of AS.
 *  It includes AS configuration items which can mostly be changed in
 *  run-time. */
struct as_config_t as_config =
{
0,                              /* Start offset */
0,                              /* Period */
USEC2TCR1(4),                   /* Pulse width */
FS_ETPU_AS_FM0_FRAME_PULSE_ON,  /* Pulse selection */
0,                              /* Frame pulse adjustment */
0,                              /* Center pulse adjustment */
FS_ETPU_AS_IRQ_FRAME_PULSE_END,  /* IRQ and DMA options */
FS_ETPU_AS_LINK_FRAME_PULSE_START + FS_ETPU_AS_LINK_FRAME_PULSE_END, /* Link
 options */
FS_ETPU_CHANNEL_TO_LINK(ETPU_RESOLVER_SAMPLE_CHAN), /* Link channels numbers at
 frame pulse start */
0,                                                  /* Link channels numbers at
 frame pulse end */
0,                                                  /* Link channels numbers at
 center pulse start */
0,                                                  /* Link channels numbers at
 center pulse end */
```

```
(uint32_t *)(0xC3FC8000 + 0x1000),      /* Pointer to eTPU data RAM where
 signals from ADC are stored */
&as_signal_config[0],                   /* Pointer to a signal configuration
 structure */
0xF,                                    /* Mask determining signals to be
 processed at frame pulse end */
0x0,                                    /* Mask determining signals to be
 processed at center pulse end */
FS_ETPU_AS_PHASE_CURRENTS_OFF,          /* Phase current processing option */
0,    /* Phase A current index in result queue */
1,    /* Phase B current index in result queue */
2,    /* Phase C current index in result queue */
   …
};
```

4.3.4 SDADC configuration

```
};};} power_manager_callback_user_config_t * powerStaticCallbacksConfigsArr[] =
 {(void *)0};move.f (r0),d1 mpy d4,d0,d15
mpy d4,d1,d13 mac d5,d3,d15
move.f (r0),d2 mac d5,d0,d13 mac d4,d2,d15
move.f (r0),d3 mpy d4,d2,d11 mac d4,d3,d13 mac -d7,d11,d15
CodeC_Ind1jump j then move I to p
CodeC_Ind1jump j then move I to p
CodeC_Ind1jump j then move I to p
CodeC_Ind2jump j then move I to p
CodeC_Ind2jump j then move I to p
CodeC_Ind2jump j then move I to p
CodeC_Ind2jump j then move I to p
CodeC_Ind3jump j then move I to p
CodeC_Ind3jump j then move I to p
CodeC_Ind3jump j then move I to p
Codeleft aligned
Codeleft aligned
Codeleft aligned
```

The Sine and Cosine, analog feedback signals must be converted to a digital representation and transferred to eTPU data RAM for Resolver function processing. Conversion and transfer should be done independently of the CPU using an on-chip ADC and eDMA. Although any of the ADC modules can be used, the described configuration adopts the Sigma-Delta ADC (SDADC).

Two SDADC modules continuously sample the Sine and Cosine signals in parallel. They are configured to obtain 32 samples of each signal per period. The following table describes the configuration for a 10 kHz excitation signal and a 320 kHz sampling frequency.

Table 2. SDADC configuration for Resolver

Configuration item	Value
SDADC clock	200 MHz / 13 = 15.38 MHz (available range 4 – 16 MHz)
ADC decimation rate	24
Resulting output data rate	200 MHz / 13 / (2 * 24) = 320,512.8 Hz
Input mode	Differetial
High-pass filter	Enabled
FIFO size	16 words

Table 2. SDADC configuration for Resolver*...continued*

Configuration item	Value
FIFO threshold	8 words
DMA request on FIFO full	Selected and enabled

Note: *Differential mode is configured for SDADC to sense resolver feedback signals (sine and cosine). Thanks to differential resolver feedback signal sensing eTPU brings the advantage of noise rejection.*

Using the function call sdadc_init(…), two instances of SDADC, namely SDADC1, and SDADC2, are configured for continuous sampling with a given output data rate, as listed in Table 2.

SDADC1 is started by software. SDADC1 trigger output is selected as an HW trigger to start SDADC2. The described configuration enables to start both modules of SDADC synchronously. Both SDADC modules are configured to generate DMA requests when the FIFO sample count reaches the value 8. It means that a quarter of the sine wave is moved by DMA from SDADC result FIFO to eTPU data RAM at once, four times per period. See the following code example and the DMA configuration in Table 4.

```
/* SDADC init*/
void sdadc_init(uint8_t pdr, uint8_t gain, uint8_t fifo_thld, uint8_t inp_mode)
{
/*####################################
    * SD ADC 1 CONFIGURATION
    ####################################*/
SDADC_1.SFR.R = 0x1B;// clear all the status flags
SDADC_1.FCR.B.FRST = 1;// generate reset to flush FIFO
SDADC_1.FCR.B.FTHLD = fifo_thld;// FIFO threshold - overcome generates FIFO full
 event flag
SDADC_1.FCR.B.FSIZE = 3;// FIFO size: 0 - 1 word, 1 - 4 words, 2 - 8 words, 3 -
 16 words
SDADC_1.FCR.B.FOWEN = 0;// disable FIFO overwrite
SDADC_1.FCR.B.FE = 1;// enable FIFO
SDADC_1.RSER.B.DFFDIRS = 1;// DMA request on data FIFO full
SDADC_1.RSER.B.DFFDIRE = 1; // data FIFO full request enable
SDADC_1.MCR.B.MODE = inp_mode;// input mode: 0 - differential, 1 - single-ended
SDADC_1.MCR.B.PDR = pdr;// Oversampling rate
SDADC_1.MCR.B.PGAN = gain;// Gain
SDADC_1.MCR.B.HPFEN = 1;// High-pass filter enabled
SDADC_1.MCR.B.EN = 1;// internal SDADC modulator enabled
SDADC_1.OSDR.B.OSD = 1;// set the output settling delay
SDADC_1.CSR.B.ANCHSEL = 0;// analog input channel AN[x] selection
SDADC_1.MCR.B.TRIGEN = 1;// enable HW trigger
/*####################################
    * SD ADC 2 CONFIGURATION
    ####################################*/
SDADC_2.SFR.R = 0x1B;// clear all the status flags
SDADC_2.FCR.B.FRST = 1;// generate reset to flush FIFO
SDADC_2.FCR.B.FTHLD = fifo_thld;// FIFO threshold - overcome generates FIFO full
 event flag
SDADC_2.FCR.B.FSIZE = 3;// FIFO size: 0 - 1 word, 1 - 4 words, 2 - 8 words, 3 -
 16 words
SDADC_2.FCR.B.FOWEN = 0;// disable FIFO overwrite
SDADC_2.FCR.B.FE = 1;// enable FIFO
// DMA settings
SDADC_2.RSER.B.DFFDIRS = 1;// DMA request on data FIFO full
SDADC_2.RSER.B.DFFDIRE = 1; // data FIFO full request enable
SDADC_2.MCR.B.MODE = inp_mode;// input mode: 0 - differential, 1 - single-ended
SDADC_2.MCR.B.PDR = pdr;// Oversampling rate
SDADC_2.MCR.B.PGAN = gain;// Gain
```

```
SDADC_2.MCR.B.HPFEN = 1;// High-pass filter enabled
SDADC_2.MCR.B.EN = 1;// internal SDADC modulator enabled
SDADC_2.OSDR.B.OSD = 1;// set the output settling delay
SDADC_2.CSR.B.ANCHSEL = 0;// analogue input channel AN[x] selection
SDADC_2.MCR.B.TRIGSEL = 0;// SDADC_1 trigger output selected as an input trigger
 for SDADC
SDADC_2.MCR.B.TRIGEDSEL = 1;// rising edge of trigger input selected
SDADC_2.MCR.B.TRIGEN = 1;// enable HW trigger
}
void sdadc_start(void)
{
// start the SDADC_1 by SW trigger
SDADC_1.STKR.R = 0xFFFF;
}
```

4.3.5 eQADC configuration

MPC5775E features two instances of Enhanced Queued ADCs (eQADC), each module having two independent ADCs, total four independent on-chip ADC modules. The eQADC transfers commands from multiple Command FIFOs (CFIFOs) to the on-chip ADCs. The multiple Result FIFOs (RFIFOs) can receive data from the on-chip ADCs or from an on-chip DSP module. The eQADC supports software and external hardware triggers from other blocks to initiate transfers of commands from the CFIFOs to the on-chip ADCs. It also monitors the fullness of CFIFOs and RFIFOs, and accordingly generates DMA or interrupt requests to control data movement between the FIFOs and the system memory, which is external to the EQADC.

All the four on-chip ADCs are utilized to sample simultaneously Phase A, Phase B and Phase C currents and DC bus voltage. Simultaneous sampling is achieved by utilizing of four command FIFOs, one per ADC channel. External HW trigger is configured to trigger CFIFOs, namely eTPUA_28 channel is selected as a trigger input for all CFIFOs. eTPUA_28 channel is assigned to AS function. ADCs are configured for single scan mode. Results are stored in Result FIFOs, one RFIFO per ADC channel.

FIFO Drain DMA requests are enabled for all the CFIFOs as well as RFIFOs. On a CFIFO Drain DMA request respective eQADC command is transferred from system memory to CFIFO. Commands for all the channels have the end Of Queue bit (EOQ) configured. Described configuration results in repeated write of a single command to the CFIFO after CFIFO is triggered. Result FIFO drain DMA request evokes respective DMA channel that performs transfer from RFIFO to eTPU data RAM. See the configuration of the DMA channels in Table 6 and Table 7. See the code example of the EQADC initialization below.

```
void eqADC_Init() {
/*Enable converter*/
EQADC_A.CFPR[0].R = CFPR_EOQ| CFPR_RFIFO(0)|CPFR_ADC_REG_VALUE(((T_U32)1<<15)|
1)|CFPR_ADC_REG_CTRL;
EQADC_A.CFPR[1].R =CFPR_EOQ| CFPR_RFIFO(1)|CFPR_BN|
CPFR_ADC_REG_VALUE(((T_U32)1<<15)|1)|CFPR_ADC_REG_CTRL;
EQADC_B.CFPR[0].R = CFPR_EOQ| CFPR_RFIFO(0)|CPFR_ADC_REG_VALUE(((T_U32)1<<15)|
1)|CFPR_ADC_REG_CTRL;
EQADC_B.CFPR[1].R =CFPR_EOQ| CFPR_RFIFO(1)|CFPR_BN|
CPFR_ADC_REG_VALUE(((T_U32)1<<15)|1)|CFPR_ADC_REG_CTRL;
EQADC_A.CFCR0.R |= (((T_U32) 1) << 26) | (((T_U32) 1) << 10); /*Trigger
 two cfifo at the same time.*/ //VDCbus and IA
EQADC_B.CFCR0.R |= (((T_U32) 1) << 26) | (((T_U32) 1) << 10); /*Trigger
 two cfifo at the same time.*/ //Ib and Ic
/*Init channels*/
eqADC_Set_Mode(&EQADC_A, 0, CFIFO_CONTINUOUS_SCAN_MODE |
 CFIFO_RISING_MODE,CFIFO_ETPUB28_TRIGGER);
eqADC_Set_Mode(&EQADC_A, 1, CFIFO_CONTINUOUS_SCAN_MODE |
 CFIFO_RISING_MODE,CFIFO_ETPUB28_TRIGGER);
```

```
eqADC_Set_Mode(&EQADC_B, 0, CFIFO_CONTINUOUS_SCAN_MODE |
 CFIFO_RISING_MODE,CFIFO_ETPUB28_TRIGGER);
eqADC_Set_Mode(&EQADC_B, 1, CFIFO_CONTINUOUS_SCAN_MODE |
 CFIFO_RISING_MODE,CFIFO_ETPUB28_TRIGGER);
eqADC_Set_Mode(&EQADC_A, 3, CFIFO_SINGLE_SCAN_MODE | CFIFO_SOFTWARE_MODE, 0);
eqADC_Set_Mode(&EQADC_B, 3, CFIFO_SINGLE_SCAN_MODE | CFIFO_SOFTWARE_MODE, 0);
//RTD1 and RTD2
eqADC_Set_Mode(&EQADC_A, 4, CFIFO_CONTINUOUS_SCAN_MODE | CFIFO_RISING_MODE,
 CFIFO_PIT3_TRIGGER);
eqADC_Set_Mode(&EQADC_A, 5, CFIFO_CONTINUOUS_SCAN_MODE | CFIFO_RISING_MODE,
 CFIFO_PIT3_TRIGGER);
/*Enable DMA*/
eqADC_Enable_DMA();
}
```

4.3.6 DSPI configuration

The DSPI modules serve as a communication interface connecting the MPC5775E MCU, gate drivers GD31xx, and system basis chip FS65.

The following table shows the summary of the DSPI modules configuration:

Table 3. DSPI modules configuration

Module	Interface	Input clock	Baud rate
DSPI_A	FS65	100 MHz	5Mbit/s
DSPI_B	GDs High side	100 MHz	4.2Mbit/s
DSPI_C	GDs Low side	100 MHz	4.2Mbit/s

```
void SPI_Init() {
/*Config MCR*/
DSPI_A.MCR.B.MSTR = 1; /*m mode*/
DSPI_A.MCR.B.PCSIS = 0b111111; /*CS active low*/
DSPI_A.MCR.B.DIS_RXF = 1; /*enable rx fifo*/
DSPI_A.MCR.B.DIS_TXF = 1; /*enable tx fifo*/
DSPI_B.MCR.B.MSTR = 1; /*m mode*/
DSPI_B.MCR.B.PCSIS = 0b111111;
; /*CS active low*/
DSPI_B.MCR.B.DIS_RXF = 1; /*enable rx fifo*/
DSPI_B.MCR.B.DIS_TXF = 1; /*enable tx fifo*/
DSPI_C.MCR.B.MSTR = 1; /*m mode*/
DSPI_C.MCR.B.PCSIS = 0b111111; /*CS active low*/
DSPI_C.MCR.B.DIS_RXF = 1; /*enable rx fifo*/
DSPI_C.MCR.B.DIS_TXF = 1; /*enable tx fifo*/
DSPI_D.MCR.B.MSTR = 1; /*m mode*/
DSPI_D.MCR.B.PCSIS = 0b111111;
; /*CS active low*/
DSPI_D.MCR.B.DIS_RXF = 1; /*enable rx fifo*/
DSPI_D.MCR.B.DIS_TXF = 1; /*enable tx fifo*/
/*Config CTAR*/
SBC_SPI.MODE.CTAR[0].R = SBC_CTAR_0;
/*Start*/
SBC_SPI.MCR.B.HALT = 0;
/*Configured M mode */
//GDLS_SPI.MCR.R = 0x80010001;
GDLS_SPI.MCR.B.MSTR = 1;      // Set DSPIx in mode S-0 or M-1
GDLS_SPI.MCR.B.MDIS=0;              // Enable clock
```

```
GDLS_SPI.MCR.B.HALT=0;              // Allow transfer
GDLS_SPI.MCR.B.PCSIS=0b111111;       // CS0 active Low
//FIFOs
GDLS_SPI.MCR.B.DIS_TXF=0;      // FIFOs enabled
GDLS_SPI.MCR.B.DIS_RXF=0;
GDLS_SPI.MCR.B.FRZ = 0;
//GDLS_SPI.CTAR[0].R = 0x78021004;
GDLS_SPI.MODE.CTAR[0].B.DBR=0;   // Double baud rate
GDLS_SPI.MODE.CTAR[0].B.FMSZ= 7; // Frame size (value+1) 4 bit
GDLS_SPI.MODE.CTAR[0].B.CPOL=0;   // Polarity CPOL=0=>sck is valid @1
GDLS_SPI.MODE.CTAR[0].B.CPHA=0;   // Phase CPHA=0=>data captured on leading edge
GDLS_SPI.MODE.CTAR[0].B.LSBFE=0;   // MSB first if 0
GDLS_SPI.MODE.CTAR[0].B.PBR = 1;
GDLS_SPI.MODE.CTAR[0].B.BR = 3;
//GDLS_SPI.CTAR[0].B.BR = 6;
GDLS_SPI.MODE.CTAR[0].B.PCSSCK = 2; // tcsc
GDLS_SPI.MODE.CTAR[0].B.CSSCK = 4;
GDLS_SPI.MODE.CTAR[0].B.PDT = 2;    // tdt
GDLS_SPI.MODE.CTAR[0].B.DT = 5;
GDLS_SPI.MODE.CTAR[0].B.PASC = 2;   // tasc
GDLS_SPI.MODE.CTAR[0].B.ASC = 4;
/*Receive FIFO Drain Request enable*/
GDLS_SPI.RSER.B.RFDF_RE = 0;
GDLS_SPI.RSER.B.TCF_RE = 0;
/* Exit HALT mode: go from STOPPED to RUNNING state*/
GDLS_SPI.MCR.B.HALT = 0x0;
//GDHS_SPI.MCR.R = 0x80010001;
GDHS_SPI.MCR.B.MSTR = 1;      // Set DSPIx in mode S-0 or M-1
GDHS_SPI.MCR.B.MDIS=0;              // Enable clock
GDHS_SPI.MCR.B.HALT=0;              // Allow transfer
GDHS_SPI.MCR.B.PCSIS=0b111111;       // CS0 active Low
//FIFOs
GDHS_SPI.MCR.B.DIS_TXF=0;      // FIFOs enabled
GDHS_SPI.MCR.B.DIS_RXF=0;
GDHS_SPI.MCR.B.FRZ = 0;
//GDHS_SPI.CTAR[0].R = 0x78021004;
GDHS_SPI.MODE.CTAR[0].B.DBR=0;   // Double baud rate
GDHS_SPI.MODE.CTAR[0].B.FMSZ= 7; // Frame size (value+1) 4 bit
GDHS_SPI.MODE.CTAR[0].B.CPOL=0;   // Polarity CPOL=0=>sck is valid @1
GDHS_SPI.MODE.CTAR[0].B.CPHA=0;   // Phase CPHA=0=>data captured on leading edge
GDHS_SPI.MODE.CTAR[0].B.LSBFE=0;   // MSB first if 0
GDHS_SPI.MODE.CTAR[0].B.PBR = 1;
GDHS_SPI.MODE.CTAR[0].B.BR = 3;
//GDHS_SPI.CTAR[0].B.BR = 6;
GDHS_SPI.MODE.CTAR[0].B.PCSSCK = 2; // tcsc
GDHS_SPI.MODE.CTAR[0].B.CSSCK = 4;
GDHS_SPI.MODE.CTAR[0].B.PDT = 2;    // tdt
GDHS_SPI.MODE.CTAR[0].B.DT = 5;
GDHS_SPI.MODE.CTAR[0].B.PASC = 2;   // tasc
GDHS_SPI.MODE.CTAR[0].B.ASC = 4;
/*Receive FIFO Drain Request enable*/
GDHS_SPI.RSER.B.RFDF_RE = 0;
GDHS_SPI.RSER.B.TCF_RE = 0;
/* Exit HALT mode: go from STOPPED to RUNNING state*/
GDHS_SPI.MCR.B.HALT = 0x0;
}
```

4.3.7 DMA transfer configuration

Eleven DMA channels are utilized to speed up the control loop operation and offload the CPU. DMA transfers the following:

- Samples of Resolver feedback sine and cosine signals from SDADC to eTPU data RAM.
- eQADC commands to command FIFO for phase current and DC bus voltage measurement.
- eQADC measurement from result FIFO to eTPU data RAM.

All the channels used in the application are listed in Table 4, together with respective DMA request sources.

Table 4. eDMA channel usage in application

Module	DMA channel	Requesting source	Description
eDMA A	0	EQADC_A_FISR0[CFFF0]	EQADC_A Command FIFO 0 Fill Flag (DC bus voltage measurement commands)
	1	EQADC_A_FISR0[RFDF0]	EQADC_A Receive FIFO 0 Drain Flag (DC bus voltage values)
	2	EQADC_A_FISR1[CFFF1]	EQADC_A Command FIFO 1 Fill Flag (Phase A current measurement commands)
	3	EQADC_A_FISR1[RFDF1]	EQADC_A Receive FIFO 1 Drain Flag (Phase A current values)
	6	EQADC_A_FISR3[CFFF3]	EQADC_A Command FIFO 3 Fill Flag (low speed DC bus voltage and phase A current measurement commands)
	7	EQADC_A_FISR3[RFDF3]	EQADC_A Receive FIFO 3 Drain Flag (low speed DC bus voltage and phase A current values)
	8	EQADC_A_FISR4[CFFF4]	EQADC_A Command FIFO 4 Fill Flag (100ms external temperature measurement 1 commands)
	9	EQADC_A_FISR4[RFDF4]	EQADC_A Receive FIFO 4 Drain Flag (100ms external temperature 1 values)
	10	EQADC_A_FISR5[CFFF5]	EQADC_A Command FIF5 0 Fill Flag (100ms external temperature measurement 2 commands)
	11	EQADC_A_FISR5[RFDF5]	EQADC_A Receive FIFO 5 Drain Flag (100ms external temperature 2 values)
eDMA B	0	EQADC_B_FISR0[CFFF0]	EQADC_B Command FIFO 0 Fill Flag (Phase B current measurement commands)
	1	EQADC_B_FISR0[RFDF0]	EQADC_B Receive FIFO 0 Drain Flag (Phase B current values)
	2	EQADC_B_FISR1[CFFF1]	EQADC_B Command FIFO 1 Fill Flag (Phase C current measurement commands)
	3	EQADC_B_FISR1[RFDF1]	EQADC_B Receive FIFO 1 Drain Flag (Phase C current values)
	6	EQADC_B_FISR3[CFFF3]	EQADC_B Command FIFO 3 Fill Flag (low speed Phase B and C currents measurement commands)
	7	EQADC_B_FISR3[RFDF3]	EQADC_B Receive FIFO 3 Drain Flag (low speed phase B and C currents values)
	36	SDADC1	SDADC_1 result ready (sine positive resolver feedback signal)
	37	SDADC2	SDADC_2 result ready (cosine positive resolver feedback signal)
	48	-	Channel is linked by eDMA B 36

Note: *The numbering of SDADC instances differs in the MPC5775E Reference manual in various chapters. SDADC instances are numbered 1-4, whereas the source signals from SDADCs to trigger eDMA operation are numbered 0-3. Note that the eDMA trigger signal SDADC_0 result ready originates in the SDADC1 module, and similarly, SDADC_1 result ready originates in SDADC2 module.*

For the eTPU Resolver function operation, three eDMA channels are used to ensure Resolver feedback signals are delivered to eTPU data RAM and to trigger eTPU processing once all the data are transferred. Two eDMA B channels, 36 and 37, are configured to transfer SDADC results after SDADC result FIFO count reaches 8. Channel 36 is configured to link another eDMA B channel (channel 48) on major loop completion (after all the results are transferred into eTPU data RAM), which causes a subsequent transfer of constants into the eTPU Resolver ATO channel HSR register. The array of the constants is defined in edma.c source file and is as follows:

```
const uint32_t link_cnst[] = {0,
FS_ETPU_RESOLVER_HSR_UPDATE_1ST,
0,
 FS_ETPU_RESOLVER_HSR_UPDATE_2ND};
```

After the first quarter of sine wave samples is transferred into eTPU data, RAM zero is written into Resolver ATO eTPU channel HSR register. Writing a zero does not initiate any eTPU operation. On the second quarter transfer end, the `FS_ETPU_RESOLVER_HSR_UPDATE_1ST` constant is written to the HSR register. Change in HSR register write initiates eTPU service request for processing the first half period of the sampled sine and cosine wave. Similar applies for the second half period with `FS_ETPU_RESOLVER_HSR_UPDATE_2ND` that evokes processing of the second half period. The following table contains the detailed configuration for DMA channels the eTPU Resolver function uses.

Table 5. DMA configuration for eTPU Resolver

Configuration item	Sine ADC FIFO DMA channel	Cosine ADC FIFO DMA channel	Linked HSR DMA channel
Source address	&SDADC_x.CDR.R	&SDADC_y.CDR.R	&link_const[0]
Destination address	resolver_instance. . signals_pba	resolver_instance. . signals_pba + 64	&ETPU.CHAN[resolver_ instance.chan_ num_exc].HSRR.R
Source transfer size / modulo	32-bits / 0 bytes	32-bits / 0 bytes	32-bits / 0 bytes
Destination transfer size / modulo	32-bits / 0 bytes	32-bits / 0 bytes	32-bits / 0 bytes
Source address offset	0 bytes	0 bytes	4 bytes
Destination address offset	4 bytes	4 bytes	0 bytes
Minor loop byte count	32 bytes	32 bytes	4 bytes
Major loop iteration count	4	4	4
Last source address adjustment	0 bytes	0 bytes	-16 bytes
Last destination address adjustment	-128 bytes	-128 bytes	0 bytes
Channel to channel linking	Enabled	Disabled	Disabled
Linked channel	HSR DMA channel	-	-

See the following code example for the SDADC DMA configuration described in Table 5.

```
void rsvlr_edma_init(void)
{
uint8_t hsr_ch = 48;
```

```
// DMA B ch 36 - SDADC0 result ready
// DMA B ch 37 - SDADC1 result ready
// DMA channels for eQADC:
// DMA settings for SDADC_0 request
DMA_B.TCD[36].SADDR.R = (uint32_t)&SDADC_1.CDR.R;
DMA_B.TCD[36].ATTR.B.SMOD = 0;/* source address modulo*/
DMA_B.TCD[36].ATTR.B.SSIZE = 2; /* source data size: 32bit */
DMA_B.TCD[36].ATTR.B.DMOD = 0;/* destination address modulo: 32 datawords each 4
 bytes */
DMA_B.TCD[36].ATTR.B.DSIZE = 2;/* destination data size: 32bit */
DMA_B.TCD[36].SOFF.R = 0;/* source address signed offset */
DMA_B.TCD[36].NBYTES.MLNO.R = 32;/* inner "minor" byte count */
DMA_B.TCD[36].SLAST.R = 0;/* TCD Last Source Address Adjustment */
DMA_B.TCD[36].DADDR.R = (vuint32_t)resolver_instance.signals_pba;/* destination
 address */
DMA_B.TCD[36].CITER.ELINKYES.B.ELINK = 1;/* Enable channel-to-channel linking on
 minor-loop complete */
DMA_B.TCD[36].CITER.ELINKYES.B.LINKCH = hsr_ch;/* Minor Loop Link Channel Number
 */
DMA_B.TCD[36].CITER.ELINKYES.B.CITER = 4;/* current major iteration count */
DMA_B.TCD[36].DOFF.R = 4;/* signed destination address offset */
DMA_B.TCD[36].DLASTSGA.R = -128; /* last destination address adjustment /scatter
 gather  address */
DMA_B.TCD[36].BITER.ELINKYES.B.ELINK = 1;/* enable channel-to-channel linking on
 minor loop complete */
DMA_B.TCD[36].BITER.ELINKYES.B.LINKCH = hsr_ch;/* Minor Loop Link Channel Number
 */
DMA_B.TCD[36].BITER.ELINKYES.B.BITER = 4;/* beginning major iteration count */
DMA_B.TCD[36].CSR.B.MAJORELINK = 1;/* enable channel-to-channel linking on major
 loop complete */
DMA_B.TCD[36].CSR.B.MAJORLINKCH = hsr_ch;/* Major Loop Link Channel Number */
// DMA settings for SDADC_1 request
DMA_B.TCD[37].SADDR.R = (uint32_t)&SDADC_2.CDR.R;
DMA_B.TCD[37].ATTR.B.SMOD = 0;/* source address modulo*/
DMA_B.TCD[37].ATTR.B.SSIZE = 2; /* source data size: 32bit */
DMA_B.TCD[37].ATTR.B.DMOD = 0;/* destination address modulo: 32 datawords each 4
 bytes */
DMA_B.TCD[37].ATTR.B.DSIZE = 2;/* destination data size: 32bit */
DMA_B.TCD[37].SOFF.R = 0;/* source address signed offset */
DMA_B.TCD[37].NBYTES.MLNO.R = 32;/* inner "minor" byte count */
DMA_B.TCD[37].SLAST.R = 0;/* TCD Last Source Address Adjustment */
DMA_B.TCD[37].DADDR.R = (vuint32_t)resolver_instance.signals_pba + 0x80;/*
 destination address */
DMA_B.TCD[37].CITER.ELINKNO.B.ELINK = 0;/* Disable channel-to-channel linking on
 minor-loop complete */
DMA_B.TCD[37].CITER.ELINKNO.B.CITER = 4;/* current major iteration count */
DMA_B.TCD[37].DOFF.R = 4;/* signed destination address offset */
DMA_B.TCD[37].DLASTSGA.R = -128; /* last destination address adjustment /scatter
 gather  address */
DMA_B.TCD[37].BITER.ELINKNO.B.ELINK = 0;/* Disable channel-to-channel linking on
 minor loop complete */
DMA_B.TCD[37].BITER.ELINKNO.B.BITER = 4;/* Beginning major iteration count */
DMA_B.TCD[37].CSR.B.MAJORELINK = 0;/* Disable channel-to-channel linking on
 major loop complete */
// DMA settings for hsr link
DMA_B.TCD[hsr_ch].SADDR.R = (uint32_t)&link_cnst[0];
DMA_B.TCD[hsr_ch].ATTR.B.SMOD = 0;/* source address modulo*/
DMA_B.TCD[hsr_ch].ATTR.B.SSIZE = 2; /* source data size: 32bit */
DMA_B.TCD[hsr_ch].ATTR.B.DMOD = 0;/* destination address modulo: 32 datawords
 each 4 bytes */
```

```
DMA_B.TCD[hsr_ch].ATTR.B.DSIZE = 2;/* destination data size: 32bit */
DMA_B.TCD[hsr_ch].SOFF.R = 4;/* source address signed offset */
DMA_B.TCD[hsr_ch].NBYTES.MLNO.R = 4;/* inner "minor" byte count */
DMA_B.TCD[hsr_ch].SLAST.R = -16;/* TCD Last Source Address Adjustment */
DMA_B.TCD[hsr_ch].DADDR.R = (uint32_t)&eTPU-
>CHAN[resolver_instance.chan_num_exc].HSRR.R;/* destination address */ //ext
DMA_B.TCD[hsr_ch].CITER.ELINKNO.B.ELINK = 0;/* Disable channel-to-channel
 linking on minor-loop complete */
DMA_B.TCD[hsr_ch].CITER.ELINKNO.B.CITER = 4;/* current major iteration count */
DMA_B.TCD[hsr_ch].DOFF.R = 0;/* signed destination address offset */
DMA_B.TCD[hsr_ch].DLASTSGA.R = 0; /* last destination address adjustment /
scatter gather  address */
DMA_B.TCD[hsr_ch].BITER.ELINKNO.B.ELINK = 0;/* Disable channel-to-channel
 linking on minor loop complete */
DMA_B.TCD[hsr_ch].BITER.ELINKNO.B.BITER = 4;/* Beginning major iteration count
 */
DMA_B.TCD[hsr_ch].CSR.B.MAJORELINK = 0;/* Disable channel-to-channel linking on
 major loop complete */
}
```

Another four eDMA channels transfer eQADC commands from system memory to eQADC command FIFO. The DMA transfer is requested when the command FIFO is empty. eQADC commands can be seen below.

```
/*ADC conversion command*/
#define ADC_CMD_VDCCFPR_EOQ|CFPR_CAL|CFPR_RFIFO(0)|CFPR_CHANNEL(28)
#define ADC_CMD_IACFPR_EOQ|CFPR_BN|CFPR_CAL|CFPR_RFIFO(1)|CFPR_CHANNEL(24)
#define ADC_CMD_IBCFPR_EOQ|CFPR_CAL|CFPR_RFIFO(0)|CFPR_CHANNEL(25)
#define ADC_CMD_ICCFPR_EOQ|CFPR_BN|CFPR_CAL|CFPR_RFIFO(1)|CFPR_CHANNEL(26)
```

The table below provides a detailed configuration of the DMA channels used by eQADC.

Table 6. eDMA configuration for eQADC command FIFO

Configuration Item	EQADC_A Command FIFO 0 DMA channel	EQADC_A Command FIFO 1 DMA channel	EQADC_B Command FIFO 0 DMA channel	EQADC_B Command FIFO 1 DMA channel
Source address	&ADC_CMD_T0 [0]	&ADC_CMD_T0 [1]	&ADC_CMD_T0 [2]	&ADC_CMD_T0 [3]
Destination address	&EQADC_A. CFPR[0].R	&EQADC_A. CFPR[1].R	&EQADC_B. CFPR[0].R	&EQADC_B. CFPR[1].R
Source transfer size/ modulo	32-bits/0 bytes	32-bits/0 bytes	32-bits/0 bytes	32-bits/0 bytes
Destination transfer size/ modulo	32-bits/0 bytes	32-bits/0 bytes	32-bits/0 bytes	32-bits/0 bytes
Source address offset	0 bytes	0 bytes	0 bytes	0 bytes
Destination address offset	0 bytes	0 bytes	0 bytes	0 bytes
Minor loop byte count	4 bytes	4 bytes	4 bytes	4 bytes
Major loop iteration count	1	1	1	1
Last source address adjustment	0 bytes	0 bytes	0 bytes	0 bytes
Last destination address adjustment	0 bytes	0 bytes	0 bytes	0 bytes

Table 6. eDMA configuration for eQADC command FIFO*...continued*

Configuration Item	EQADC_A Command FIFO 0 DMA channel	EQADC_A Command FIFO 1 DMA channel	EQADC_B Command FIFO 0 DMA channel	EQADC_B Command FIFO 1 DMA channel
Channel to channel linking	Disabled	Disabled	Disabled	isabled
Linked channel	-	-	-	-

Last set of four eDMA channels transfer results from eQADC Result FIFO to eTPU data RAM for further processing. Table 7 shows the detailed configuration for DMA channels used by eQADC result FIFO.

Table 7. eDMA channel configuration for eQADC Result FIFO

Configuration Item	EQADC_A Receive FIFO 0 DMA channel	EQADC_A Receive FIFO 1 DMA channel	EQADC_B Receive FIFO 0 DMA channel	EQADC_B Receive FIFO 1 DMA channel
Source address	&EQADC_A. RFPR[0].R	&EQADC_A. RFPR[1].R	&EQADC_B. RFPR[0].R	&EQADC_B. RFPR[1].R
Destination address	0xC3FC9000 (free eTPU data RAM space)	0xC3FC9004 (free eTPU data RAM space)	0xC3FC9008 (free eTPU data RAM space)	0xC3FC900C (free eTPU data RAM space)
Source transfer size/ modulo	32-bits / 0 bytes	32-bits / 0 bytes	32-bits / 0 bytes	32-bits / 0 bytes
Destination transfer size/ modulo	32-bits / 0 bytes	32-bits / 0 bytes	32-bits / 0 bytes	32-bits / 0 bytes
Source address offset	0 bytes	0 bytes	0 bytes	0 bytes
Destination address offset	0 bytes	0 bytes	0 bytes	0 bytes
Minor loop byte count	4 bytes	4 bytes	4 bytes	4 bytes
Major loop iteration count	1	1	1	1
Last source address adjustment	0 bytes	0 bytes	0 bytes	0 bytes
Last destination address adjustment	0 bytes	0 bytes	0 bytes	0 bytes
Channel to channel linking	Disabled	Disabled	Disabled	Disabled
Linked channel	-	-	-	-

The eQADC DMA configuration is stored in the array, and the common function *DMA_Init()* configures DMA channels with this configuration.

```
DMA_Cfg DMA_A_Cfg_Array[64] = {
{ (T_U32) &ADC_CMD_T0[0], (T_U32) &EQADC_A.CFPR[0].R, DMA_SIZE_32BIT, 4, 1, 0,
 0, 0, 0, 0, 0, 0, 1, 0 }, /*Group 0*/
{ (T_U32) &EQADC_A.RFPR[0].R, (T_U32)0xC3FC8000 + 0x1000, DMA_SIZE_32BIT, 4, 1,
 0, 0, 0, 0, 0, 0, 0, 1, 1 },
{ (T_U32) &ADC_CMD_T0[1], (T_U32) &EQADC_A.CFPR[1].R, DMA_SIZE_32BIT, 4, 1, 0,
 0, 0, 0, 0, 0, 0, 1, 2 },
{ (T_U32) &EQADC_A.RFPR[1].R, (T_U32)0xC3FC8000 + 0x1004, DMA_SIZE_32BIT, 4, 1,
 0, 0, 0, 0, 0, 0, 0, 1, 3 },
...
}
```

```
DMA_Cfg DMA_B_Cfg_Array[64] = {
{ (T_U32) &ADC_CMD_T0[2], (T_U32) &EQADC_B.CFPR[0].R, DMA_SIZE_32BIT, 4, 1, 0,
 0, 0, 0, 0, 0, 0, 1, 0 }, /*Group 0*/
{ (T_U32) &EQADC_B.RFPR[0].R, (T_U32)0xC3FC8000 + 0x1008, DMA_SIZE_32BIT, 4, 1,
 0, 0, 0, 0, 1, 0, 0, 1, 1 },
{ (T_U32) &ADC_CMD_T0[3], (T_U32) &EQADC_B.CFPR[1].R, DMA_SIZE_32BIT, 4, 1, 0,
 0, 0, 0, 0, 0, 0, 1, 2 },
{ (T_U32) &EQADC_B.RFPR[1].R, (T_U32)0xC3FC8000 + 0x100C, DMA_SIZE_32BIT, 4, 1,
 0, 0, 0, 0, 0, 0, 0, 1, 3 },
…
}
```

4.3.8 Port control and pin multiplexing

PMSM FOC motor control application requires the following on-chip pins assignment. Refer to the following table.

Table 8. Pins assignment for MPC5775E PMSM FOC control

Module	Signal name	Pin name/Functionality	Description
eQADC A	M_AN24	ANA24/ANB24	Phase A current
	M_AN28	ANA28/ANB28	DC bus voltage
eQADC B	M_AN25	ANA25/ANB25	Phase B current
	M_AN26	ANA26/ANB26	Phase C current
SDADC 1	M_ANA0_SDA0	ANA0/SDA0	Sine positive resolver feedback signal (differential)
	M_ANA1_SDA1	ANA1/SDA1	Sine negative resolver feedback signal (differential)
SDADC 2	M_ANA16_SDB0	ANA16/SDB0	Cosine positive resolver feedback signal (differential)
	M_ANA17_SDB1	ANA17/SDB1	Cosine negative resolver feedback signal (differential)
CAN_A (FreeMASTER)	TXA	TXA/GPIO83	CAN transmit data
	RXA	RXA/GPIO84	CAN receive data
DSPI A (FS 65)	DSPI_SCKA	SCKA/GPIO93	DSPI A clock output
	DSPI_SINA	SINA/GPIO94	DSPI A serial data input
	DSPI_SOUTA	SOUTA/GPIO95	DSPI A serial data output
	DSPI_CSA0	PCSA0/GPIO96	DSPI A chip select 0 output signal for
DSPI B (GD high side)	DSPI_SCKB	SCKB/GPIO102	DSPI B clock output
	DSPI_SINB	SINB/GPIO103	DSPI B serial data input
	DSPI_SOUTB	SOUTB/GPIO104	DSPI B serial data output
	DSPI_CSB1	PCSB1/GPIO106	DSPI B chip select 1 output signal for
DSPI C (GD low side)	DSPI_SCKC	SCKC/GPIO235	DSPI C clock output
	DSPI_SINC	SINC/GPIO236	DSPI C serial data input
	DSPI_SOUTC	SOUTC/GPIO237	DSPI C serial data output
	DSPI_CSC0	PCSC0/GPIO105	DSPI C chip select 0 output signal for

Table 8. Pins assignment for MPC5775E PMSM FOC control*...continued*

Module	Signal name	Pin name/Functionality	Description
eTPU A	M_eTPUA9	ETPUA9 /GPIO188	PWMM master channel
	M_eTPUA0	ETPUA0 /GPIO179	PWMM Phase A Base channel (high-side driver)
	M_eTPUA1	ETPUA1 /GPIO180	PWMM Phase A Complementary channel (low-side driver)
	M_eTPUA2	ETPUA2 /GPIO181	PWMM Phase B Base channel (high-side driver)
	M_eTPUA3	ETPUA3 /GPIO182	PWMM Phase B Complementary channel (low-side driver)
	M_eTPUA6	ETPUA6 /GPIO185	PWMM Phase C Base channel (high-side driver)
	M_eTPUA7	ETPUA7 /GPIO186	PWMM Phase C Complementary channel (low-side driver)
	M_eTPUA5	ETPUA5 /GPIO119	Resolver excitation output
eTPU B	M_eTPUB8	ETPUB8	Resolver ATO channel (no input/output)
	M_eTPUB27	ETPUB27 /GPIO174	Resolver sample (extrapolation) input
	M_eTPUB28	ETPUB28 /GPIO175	Analog sensing trigger output
	M_eTPUB0	ETPUB0/GPIO147	Resolver diagnostic channel
GPIO GPIO	FSENB	GPIO79	Fail safe enable (output)
	FSS_HS	GPIO221	Fail safe state high site (output)
	FSS_LS	GPIO222	Fail safe state low site (output)
	INTB_GD_HSn	GPIO194	High site GD interrupt (input)
	INTB_GD_LSn	GPIO193	Low site GD interrupt (input)
	EN_FLYBK_HS	GPIO432	High site GD power supply enable (output)
	EN_FLYBK_LS	GPIO204	Low site GD power supply enable (output)
	PPC_GPIO90	GPIO90	Green indication LED
	PPC_GPIO89	GPIO89	Yellow indication LED

Pin configuration is stored in the array and function *GPIO_Init()* executes the configuration.

```
GPIO_Cfg GPIO_Cfg_Array[] = {
#if defined(HW_VER_PIMrB)
{ 79, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 0 }, /*FSEn*/
{ 221, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 0 }, /*FSH*/
{ 222, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 0 }, /*FSL*/
// read backs
{ 223, FUNCTION_GPIO | DIRECTION_IN | PULL_DISABLE, 0 }, /*FSEn read back*/
{ 75, FUNCTION_GPIO | DIRECTION_IN | PULL_DISABLE, 0 }, /*FSH*/
{ 76, FUNCTION_GPIO | DIRECTION_IN | PULL_DISABLE, 0 }, /*FSL*/
{ 77, FUNCTION_GPIO | DIRECTION_IN | PULL_DISABLE, 0 }, /*FSb0*/
{ 78, FUNCTION_GPIO | DIRECTION_IN | PULL_DISABLE, 0 }, /*FSb1*/
{ 220, FUNCTION_GPIO | DIRECTION_IN | PULL_DISABLE, 0 }, /*KRAM*/
{ 193, FUNCTION_GPIO | DIRECTION_IN | PULL_UP, 0 }, /*GD_INTB_L*/
{ 194, FUNCTION_GPIO | DIRECTION_IN | PULL_UP, 0 }, /*GD_INTB_H*/
{ 235, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 0 }, /*SCKC*/
{ 236, FUNCTION_PRIM | DIRECTION_IN | PULL_DISABLE,  0 }, /*SINC*/
{ 237, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 0 }, /*SOUTC*/
```

```
{ 105, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 1 }, /*GD LS driver uses
 GPIO as CS*/
{ 106, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 1 }, /*GD HS driver uses
 GPIO as CS*/
{ 448, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 0 }, /*Test output sync
 pulse for RDSON testing*/
#elif defined(HW_VER_PIMrA)
{ 75, FUNCTION_GPIO | DIRECTION_IN | PULL_UP, 0 }, /*GD_INTB_L*/
{ 76, FUNCTION_GPIO | DIRECTION_IN | PULL_UP, 0 }, /*GD_INTB_H*/
{ 105, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*PCS0B*/
{ 106, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*PCS1B*/
{ 107, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*PCS2B*/
{ 108, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*PCS3B*/
{ 109, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*PCS4B*/
{ 110, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*PCS5B*/
{ 223, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 0 }, /*FSEn*/
#else
#error "!no hw version is defined!"
#endif
{ 83, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*TXA*/
{ 84, FUNCTION_PRIM | DIRECTION_IN | PULL_DISABLE, 1 }, /*RXA*/
{ 89, FUNCTION_GPIO | DIRECTION_OUT | DIRECTION_IN | PULL_DISABLE, 1 }, /*LED*/
{ 90, FUNCTION_GPIO | DIRECTION_OUT | DIRECTION_IN | PULL_DISABLE, 1 }, /*LED*/
{ 93, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*SCKA*/
{ 94, FUNCTION_PRIM | DIRECTION_IN | PULL_DISABLE, 1 }, /*SINA*/
{ 95, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*SOUTA*/
{ 96, FUNCTION_PRIM | DIRECTION_OUT | PULL_ENABLE, 1 }, /*PCSA0*/
{ 102, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 0 }, /*SCKB*/
{ 103, FUNCTION_PRIM | DIRECTION_IN | PULL_DISABLE,  0 }, /*SINB*/
{ 104, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 0 }, /*SOUTB*/
{ 119, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 1 }, /*ADC trigger*/
{ 174, FUNCTION_PRIM | DIRECTION_IN | PULL_DISABLE, 0 }, /* resolver sample B27
 */
{ 175, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 0 }, /* resolver trigger
 B28 */
{ 147, FUNCTION_PRIM | DIRECTION_OUT | PULL_DISABLE, 0 }, /* resolver diag */
{ 188, FUNCTION_ALTER1 | DIRECTION_OUT | PULL_DISABLE, 1 }, /*PWM m
*/
{ 179, FUNCTION_ALTER1 | DIRECTION_OUT | PULL_DISABLE, 0 }, /*AU*/
{ 180, FUNCTION_ALTER1 | DIRECTION_OUT | PULL_DISABLE, 0 }, /*AL*/
{ 181, FUNCTION_ALTER1 | DIRECTION_OUT | PULL_DISABLE, 0 }, /*BU*/
{ 182, FUNCTION_ALTER1 | DIRECTION_OUT | PULL_DISABLE, 0 }, /*BL*/
{ 185, FUNCTION_ALTER1 | DIRECTION_OUT | PULL_DISABLE, 0 }, /*CU*/
{ 186, FUNCTION_ALTER1 | DIRECTION_OUT | PULL_DISABLE, 0 }, /*CL*/
{ 204, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 1 }, /*Flyback En Bottom */
{ 432, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 1 }, /*Flyback En Top */
{ 117, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 0 }, /*sim fccu0*/
{ 125, FUNCTION_GPIO | DIRECTION_OUT | PULL_DISABLE, 1 }, /*sim fccu1*/
};
```

4.4 Software architecture

4.4.1 Introduction

Section 4.4.1 describes the software design of the PMSM Field Oriented Control framework application. The application overview and description of software implementation are provided. The aim of section 4.4.2 is to help users understand software design.

4.4.2 Application data flow overview

The application software is interrupt-driven, running in real-time. One periodic interrupt service routine is associated with the eTPU analog sensing interrupt, executing all motor control tasks. One periodic interrupt service routine includes fast current and slow speed loop control. All tasks are performed in an order described by the application state machine shown in Figure 24. Application flowcharts are shown in Figure 22 and Figure 23.

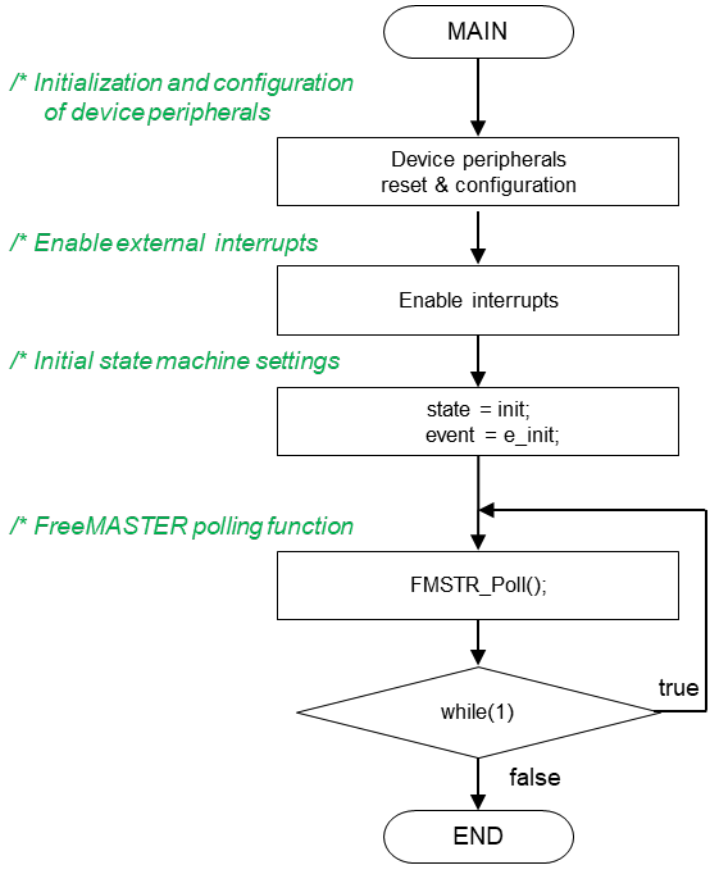

Figure 22. Flow chart diagram of main function with background loop

The state machine functions are called within a periodic interrupt service routine to achieve precise and deterministic sampling of analog quantities and execute all necessary motor control calculations. Hence, in order to call state machine functions, the peripheral causing this periodic interrupt must be properly configured and the interrupt should be enabled. As described in MPC5775E Device initialization, all peripherals are initially configured, and all interrupts are enabled after a RESET of the device. When interrupts are enabled, and all MPC5775E peripherals are correctly configured, the state machine functions are called from the eTPU analog sensing interrupt service routine. The background loop handles non-critical timing tasks, like the FreeMASTER communication polling.

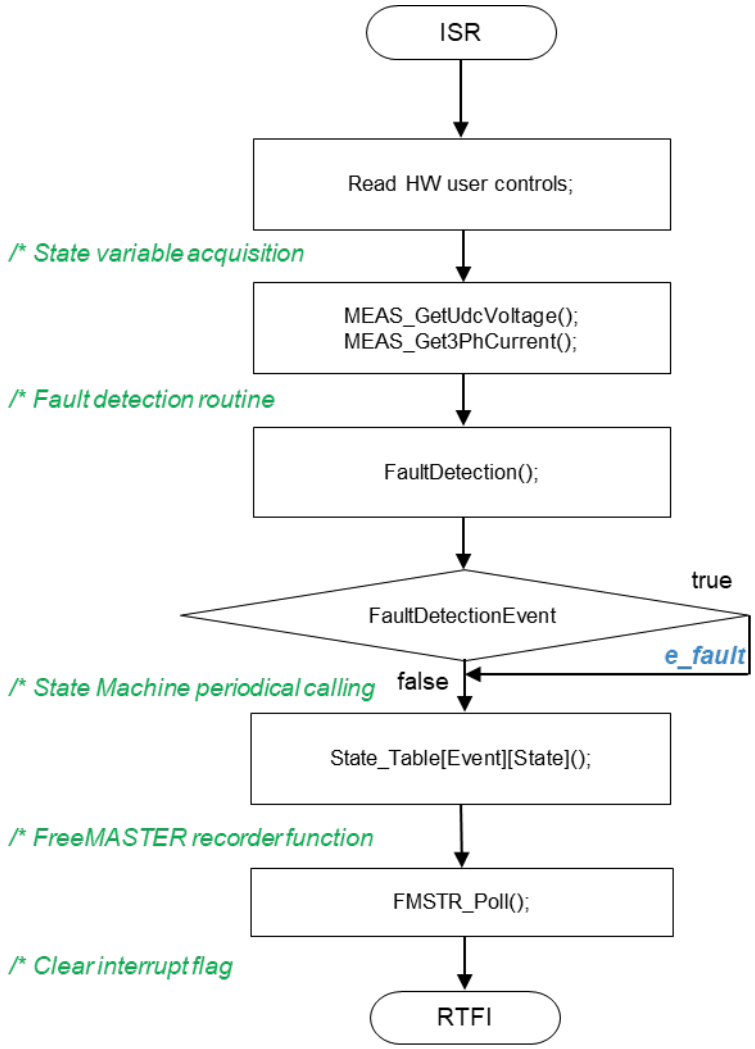

Figure 23. Flow chart diagram of periodic interrupt service routine

4.4.3 State machine

The application state machine is implemented using a two-dimensional array of pointers to the functions using a variable called *StateTable[Event][State]()*. 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 the state machine function, which causes a function call whenever *StateTable[Event][State]()* is called.

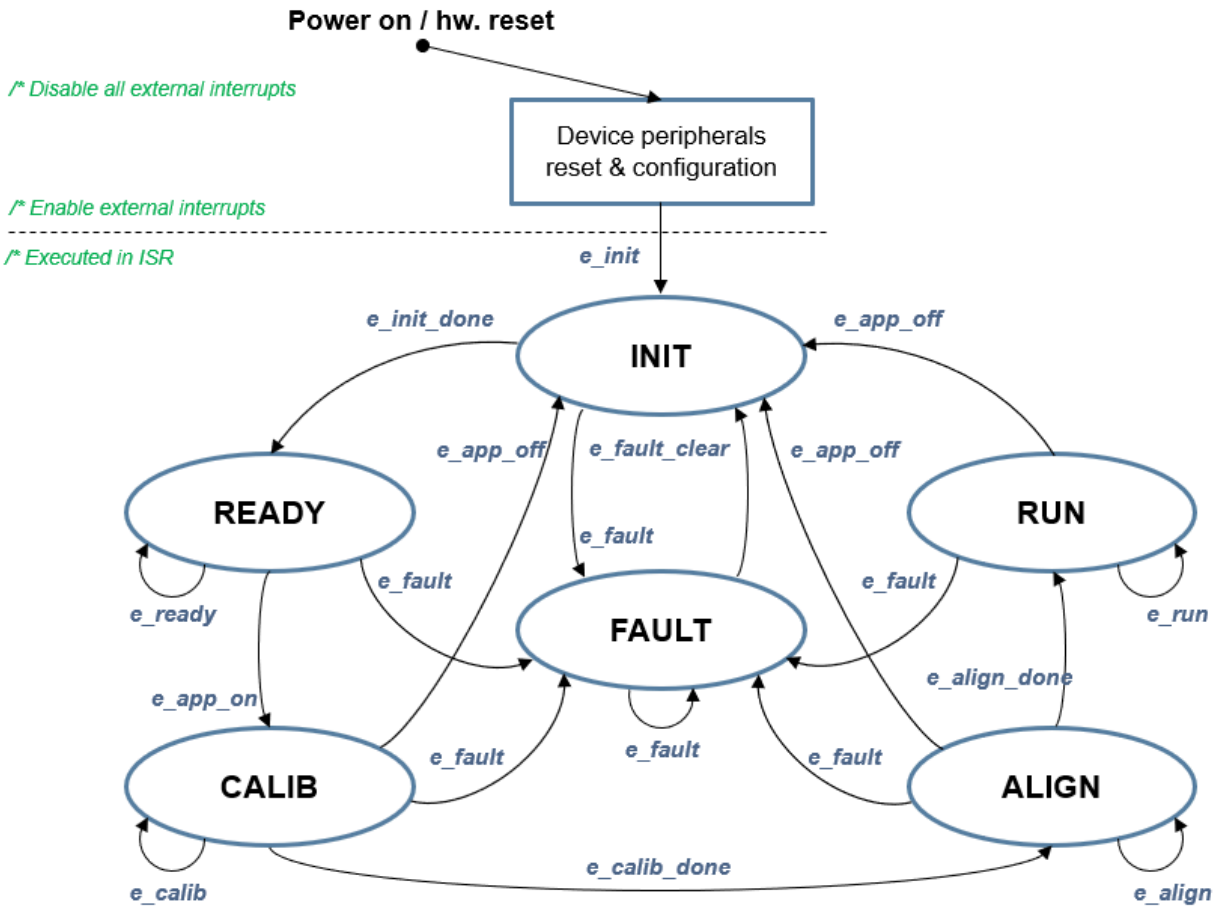

Figure 24. Application state machine

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

AppStates:

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

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

AppEvents:

- e_fault - event = 0
- e_fault_clear - event = 1
- e_init - event = 2
- e_init_done - event = 3
- e_ready - event = 4

- e_app_on - event = 5
- e_app_off - event = 11
- e_calib - event = 6
- e_calib_done - event = 7
- e_align - event = 8
- e_align_done - event = 9
- e_run - event = 10

4.4.4 State – FAULT

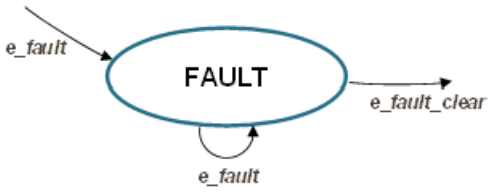

Figure 25. FAULT state with transitions

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 the 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 sets and puts the application immediately into the fault state. Even if the 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 a short glitch on monitored pins/variables caused the fault. State FAULT can only be left when the application variable *switchFaultClear* is manually set to *true* (using FreeMASTER) or by simultaneously pressing the user buttons (BTN0 and BTN1) on the MCSPTR2A5775E inverter board. 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 the following code:

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

Setting an event to *cntrState.event* = *e_fault_clear* when in FAULT state represents a new request to proceed to INIT state. The described 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* to true. However, according to the interrupt data flow diagram shown in Figure, function *faultDetection()* is called before state machine function *state_table[event][state]()*. Therefore, all faults will be rechecked, and if any fault condition remains 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()* modifies 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* manually sets to true.

The application is scanning for the 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. See chapter MCAT settings and tuning for further information on how to set these thresholds using the MCAT. In addition, fault state is entered if following errors are detected:

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

4.4.5 State – INIT

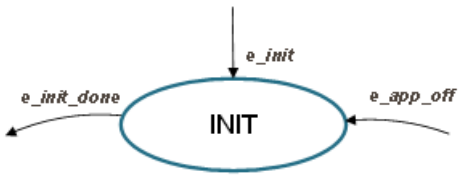

Figure 26. INIT state with transitions

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. 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.

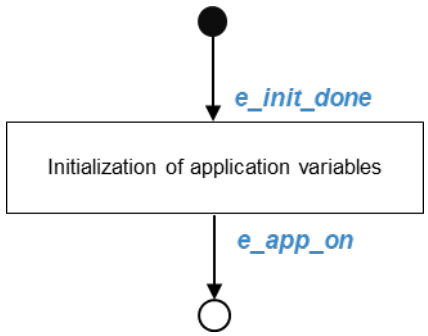

Figure 27. Flow chart of state INIT

4.4.6 State – READY

Figure 28. READY state with transitions

In READY state, the application waits for the user command to start the motor. The application is released from waiting mode by pressing the onboard button BTN0 or BTN1 or by FreeMASTER interface setting the variable *switchAppOnOff = true* (see flow chart in Figure 29).

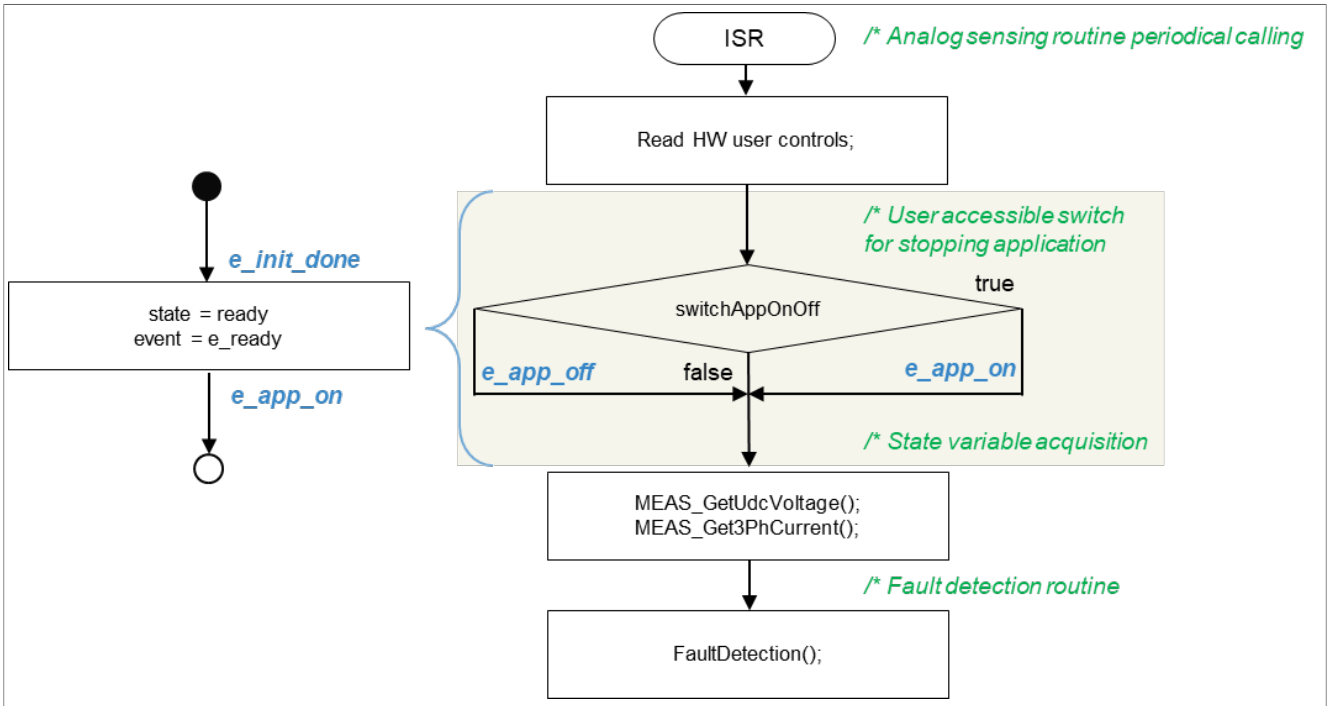

Figure 29. Flow chart of state READY

4.4.7 State – CALIB

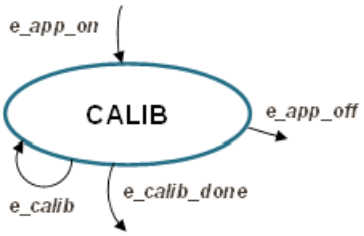

Figure 30. CALIB state with transitions

Once the state machine enters CALIB state, all PWM outputs are enabled. State calib is a reserver for ADC modules calibration. MCSPTR2A5775E provides eTPU-based motor control. All analog quantities are sampled and post processed by eTPU. Calibration, offset calibration, and filtering are part of the analog sensing function from the eTPU function selector.

State CALIB is a state that allows transition back to itself, provided no faults are present, the user does not request a stop of the application (by *switchAppOnOff=true*), and the calibration process has not finished. When the application event sets automatically to *cntrState.event=e_calib_done,* state machine can proceed to state ALIGN.

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 the falling edge of *switchAppOnOff=false* using FreeMASTER.

4.4.8 State – ALIGN

Figure 31. ALIGN state with transitions

State Align shows the alignment of the rotor and stator flux vectors to mark zero position. The zero position is not known when using a model-based approach for position estimation. The zero position is obtained at ALIGN state, where a DC voltage is applied to the d-axis voltage for a certain period. DC voltage causes the rotor to rotate to an "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 wait for the rotor to stabilize in an aligned position, a certain time period is selected during which the DC voltage is constantly applied. The period of time and the amplitude of DC voltage can be modified in INIT state. Timing is implemented using a software counter that counts from a pre-defined value to zero. During this time, the event remains set to *cntrState.event=e_align*. When the counter reaches zero, the counter resets back to the pre-defined value, and the event automatically sets to *cntrState.event=e_align_done*. Event change to *e_align_done* enables a transition to RUN state see flow chart in Figure 32.

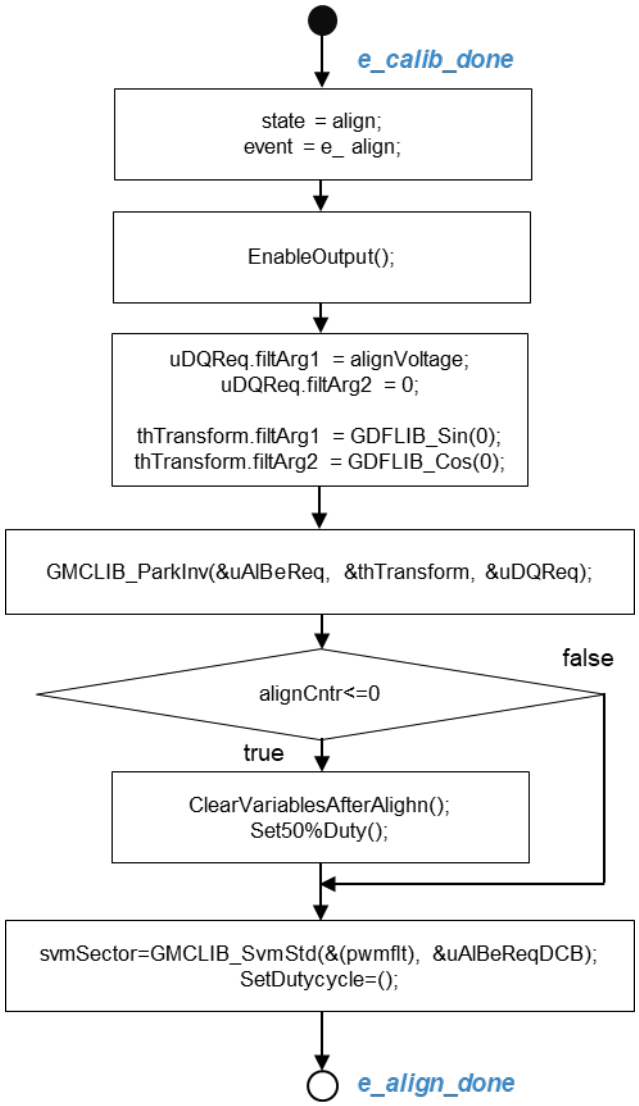

Figure 32. Flow chart of state ALIGN

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 the falling edge of *switchAppOnOff=false* using FreeMASTER or using the switch.

4.4.9 State – RUN

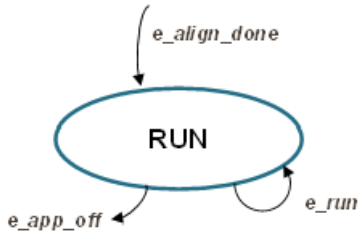

Figure 33. RUN state with transitions

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

Figure 34 shows module interconnection and used functions. Figure 35 shows implementation of FOC algorithm and used functions and variables. As shown in the diagram, Rotor position and speed are estimated by ATO observer. ATO generates default rotor position and speed feedback for FOC.

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

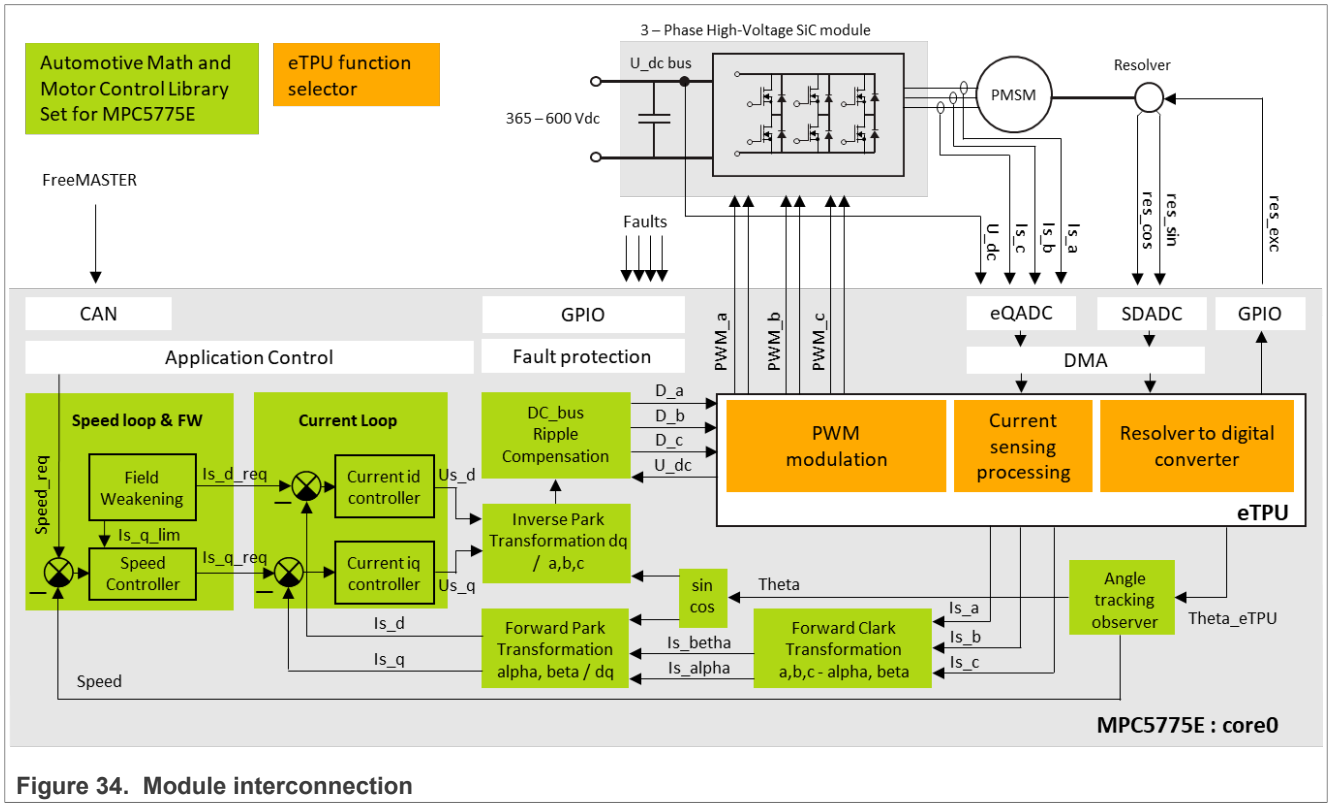

Figure 34. Module interconnection

4.5 AMMCLIB integration

The application software of the FOC control with resolver position sensing and 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. Figure 35 shows the essential blocks of the FOC structure. AMMCLIB supports all available data type implementations: 32-bit fixed-point, 16-bit fixed-point, and single-precision floating-point. To achieve high performance of the MPC5775E core, floating point arithmetic is used as a reference for motor control applications.

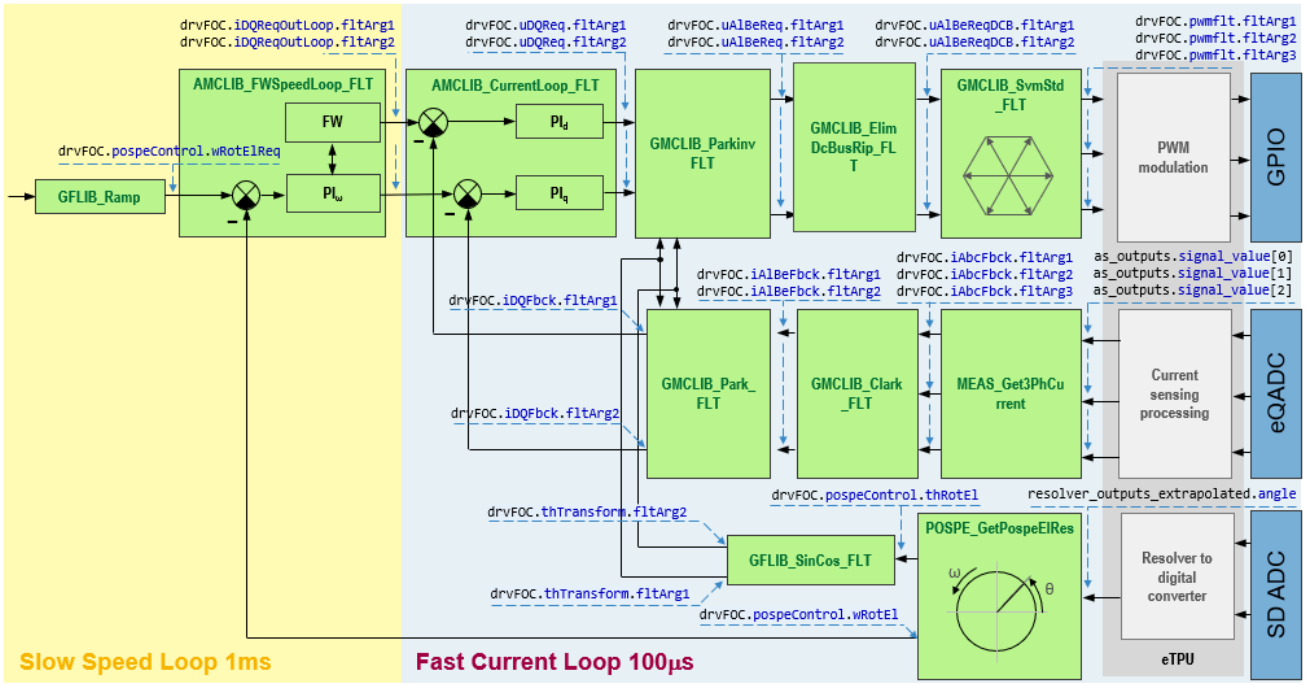

Figure 35. Variables/function name convention of implemented Sensor-based FOC with FW on MPC5775E

Current Loop function AMCLIB_CurrentLoop unites and optimizes most inner loop of the FOC cascade structure (see Figure 35). 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. Figure 36 shows all functions and data structures.

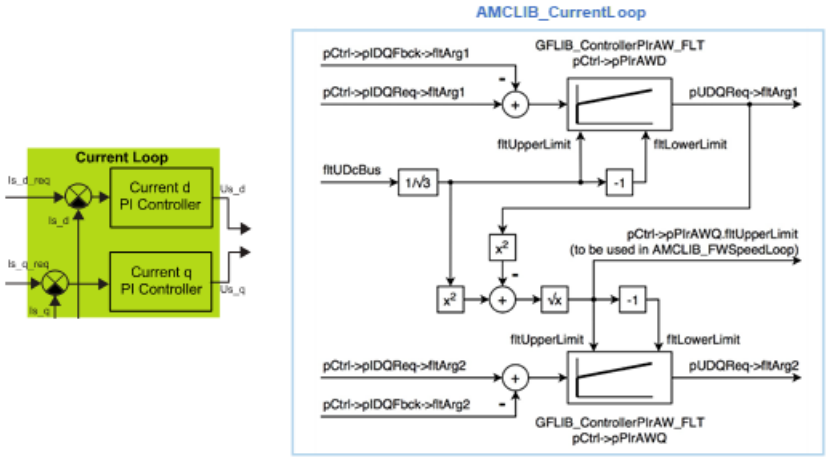

Figure 36. Functions and data structures in AMCLIB_CurrentLoop

Required d- and q-axis stator currents can be manually modified or generated by the outer loop of the cascade structure consisting of Speed Loop and Field Weakening (FW), as shown in Figure 33. To achieve a highly optimized level, AMCLIB_FWSpeedLoop merges two functions of the AMMCLIB, namely speed control loop AMCLIB_SpeedLoop and field weakening control AMCLIB_FW, Figure 34. AMCLIB_SpeedLoop consists of speed PI controller GFLIB_ControllerPIpAW, speed ramp GFLIB_Ramp placed in the 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 Field weakening. Figure 37 shows all functions and data structures used in the AMCLIB_FW function.

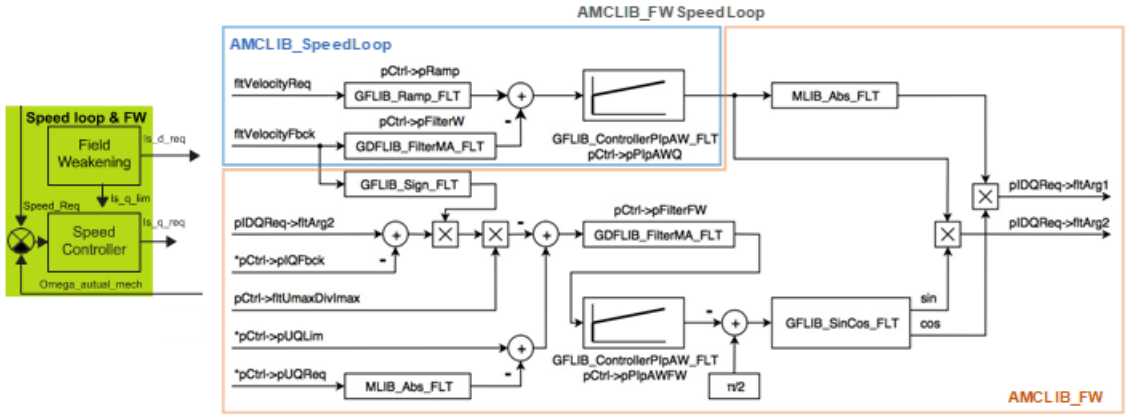

Figure 37. Functions and data structures in AMCLIB_FWSpeedLoop

AMCLIB_FW key advantages:

- Fully utilize the drive capabilities (speed range and 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

Angle tracking observer AMCLIB_TrackObsrv constitute important blocks in application, ATO estimate rotor position and speed based on the inputs from eTPU.

AMCLIB_TrackObsrv is an adopted phase-locked-loop algorithm that estimates rotor speed and position, keeping $\theta_{err} = 0$. Negligible position error is ensured by a loop compensator that is the PI controller. While the PI controller generates the estimated rotor speed, the integrator used in the phase-locked-loop algorithm serves the estimated rotor position.

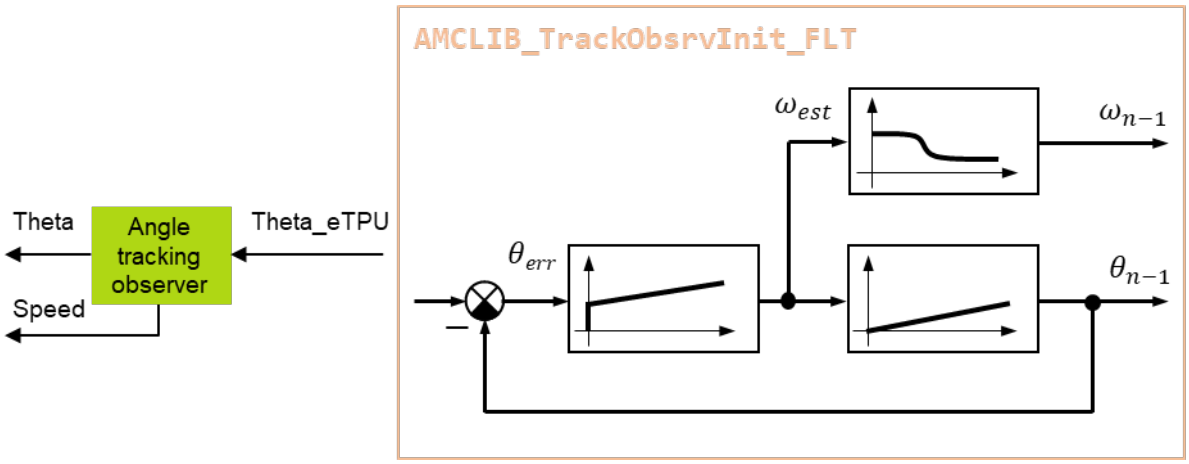

Figure 38. Structure of the AMCLIB_TrackObsrver

See MPC5775E AMMCLIB User's manual (see section References) for more information related to AMMCLIB FOC functions. Parameters of the PI controllers placed in the speed control loop, current control loop, 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 described in AN4912, AN4642 (see section References).

4.6 MCAT integration

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

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

The MCAT tool generates a set of constants to the dedicated header file (for example “{Project Location}\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 controls the application and monitors variables during run time. Communication with the host PC passes via USB. However, because FreeMASTER supports RS232 communication, there must be a driver for the physical USB interface, OpenSDA, installed on the host PC that creates a virtual COM port from the USB. The driver shall be installed automatically, plugging MPC5775E-EVB into the USB port. Alternatively, it can be downloaded from www.pemicro.com/opensda/. The application configures the eSCI module of the MPC5775E for a communication speed of 115200 bps. Therefore, the FreeMASTER user interface should also be configured respectively.

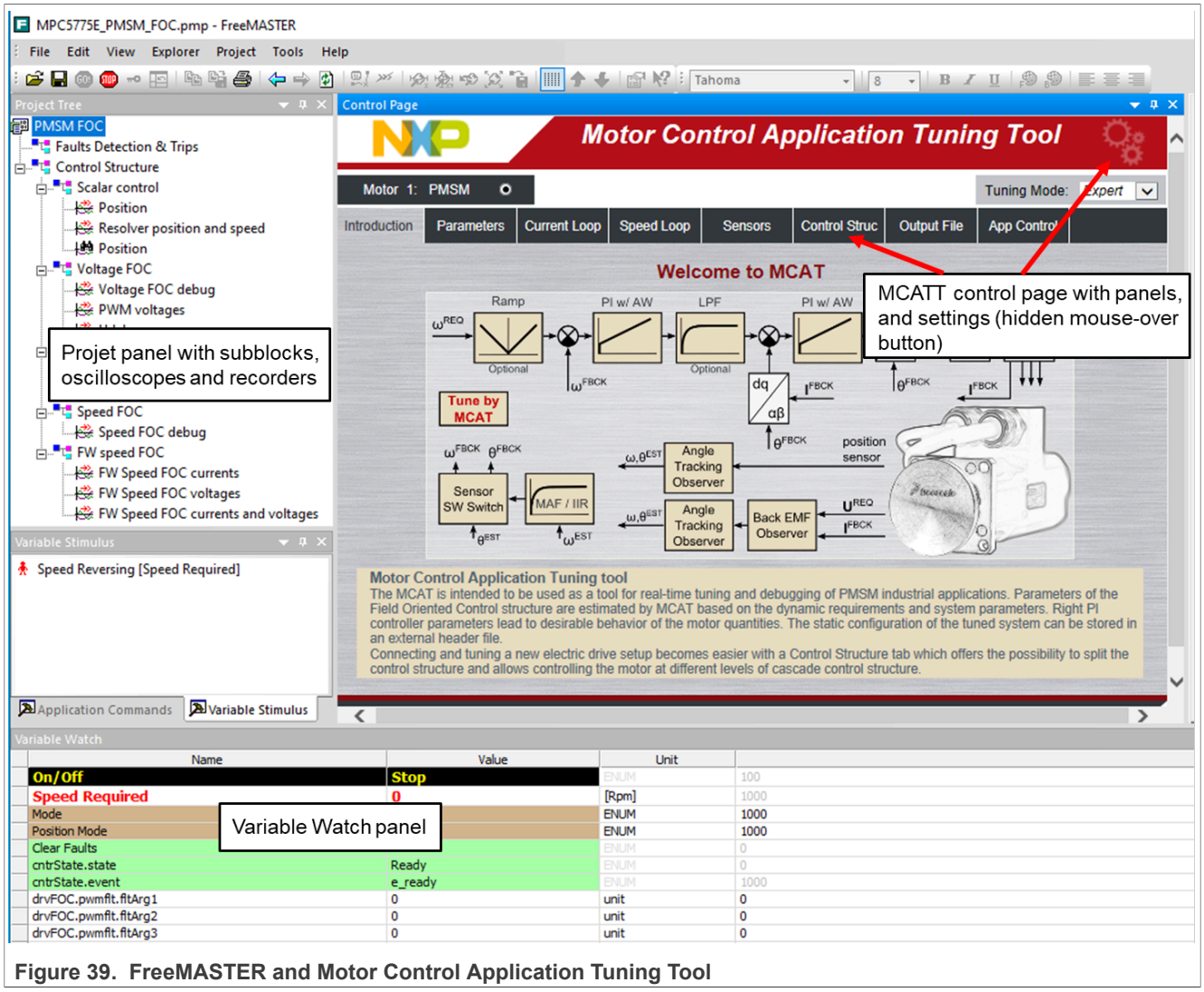

Figure 39. FreeMASTER and Motor Control Application Tuning Tool

5.1 MCAT settings and tuning

5.1.1 Application configuration and tuning

FreeMASTER and MCAT interface (Figure 39) 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). However, some of them (especially the fault limit thresholds) must be set using the configuration header file generation, which can be done on the “Output File” panel by clicking the “Generate Configuration File” (see figure).

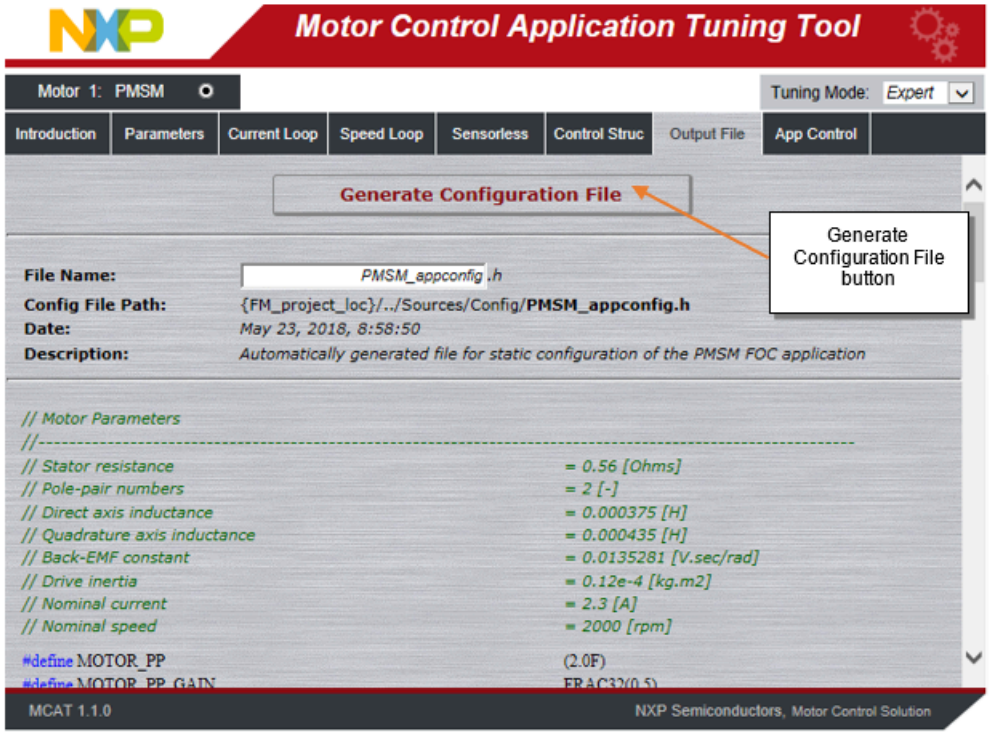

Figure 40. Output File panel and "Generate Configuration File" button

Parameters runtime update is done using the "Update Target" button. Changes can also be saved using the "Store Data" button or reloaded to previously saved configurations using the "Reload Data" button.

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

Note: *MCAT tool can be configured using hidden mouse-over "Settings" button. See Figure 36, 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 because it forces the MCAT to look for different variable names and to generate a 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 shown in Figure 38. The following list of settings pages is based on the PMSM sensor-based 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

Changes can be tested using MCAT “Control Struc” page Figure 42, where the following control structures can be enabled:

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

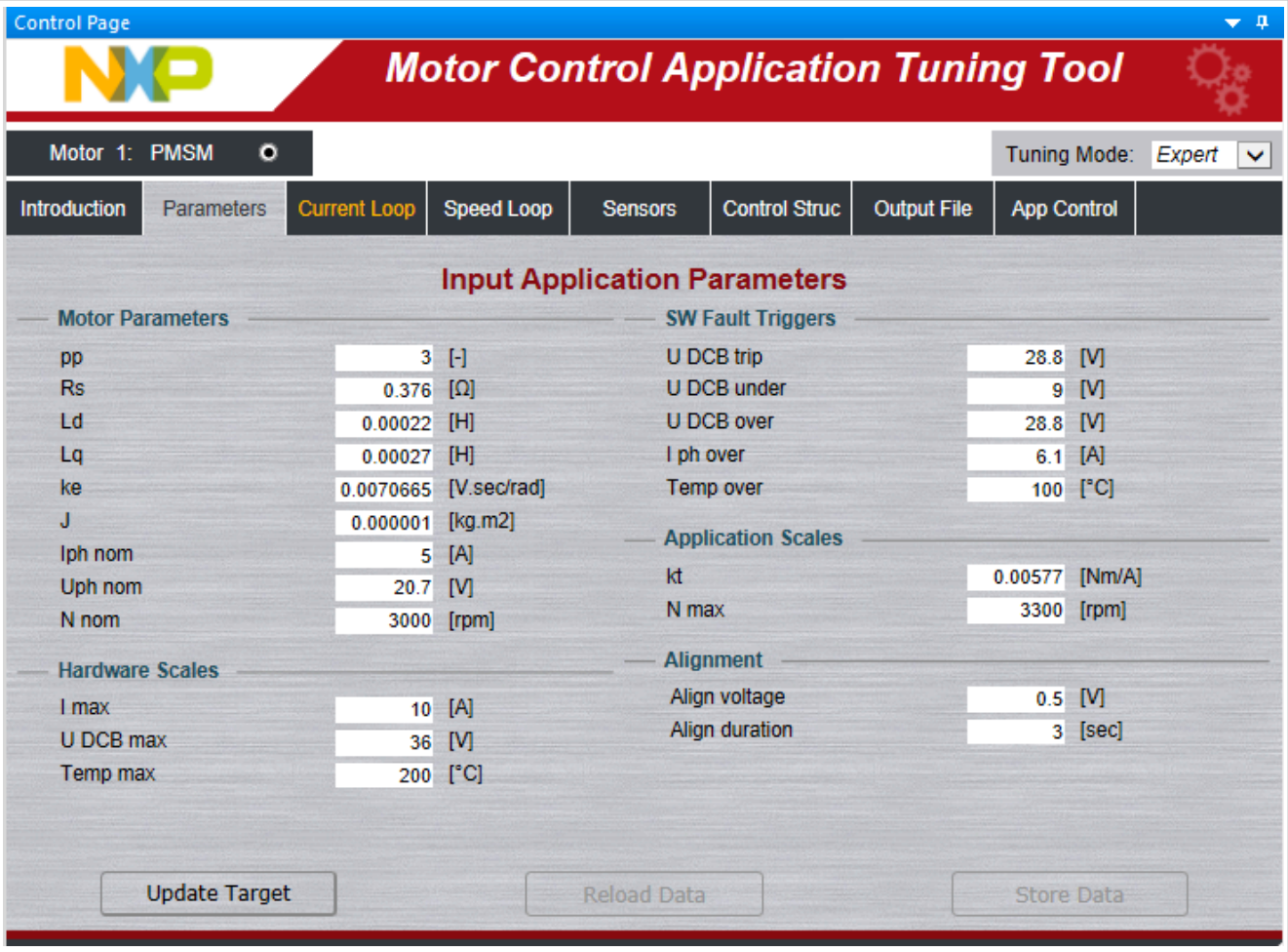

Figure 41. MCAT input application parameters page

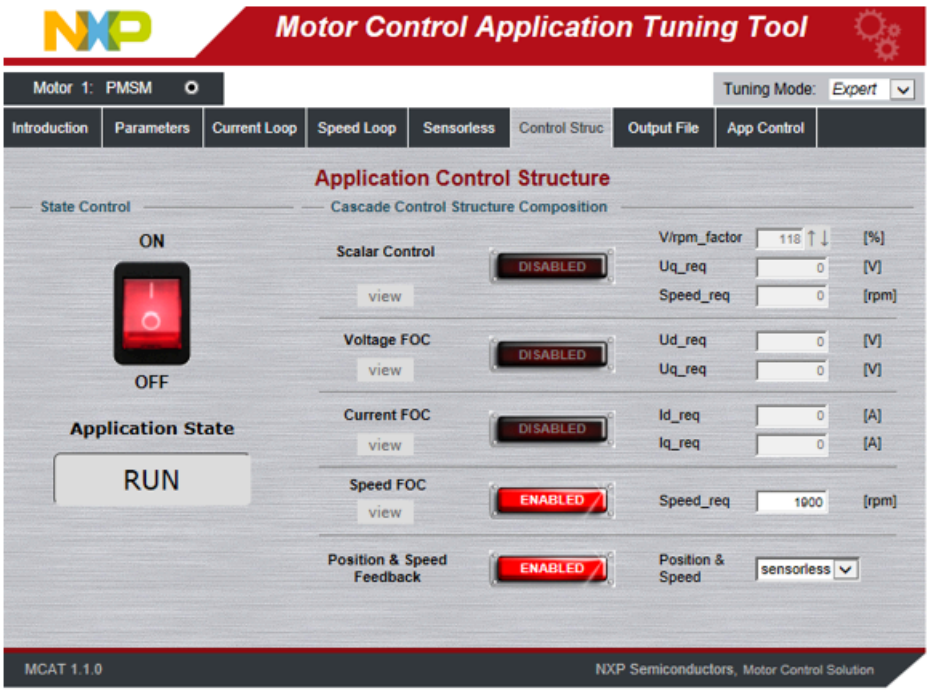

Figure 42. MCAT application control structure page

5.2 MCAT application control

All application state machine variables can be seen on the FreeMASTER MCAT App control page, as shown in Figure 43. Warnings and faults are signaled by a highlighted red color bar with the name of the fault source. The warnings are signaled by a round LED-like indicator, 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 43, for example, there is a pending fault flag, and one warning indicated ("U_{dcb} 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 exceeds 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. No warning indicators are highlighted after all actual fault sources have been removed, but the fault indicators remain highlighted. Pressing the "FAULT" button clears all pending faults and enables transition of the state machine into INIT and then READY state. After the application faults have been cleared and the application is in READY state, all variables should be set to their default values. The application can be started by selecting APP_ON on the application On/Off switch. Successful selection is indicated by highlighting the On/Off button in green.

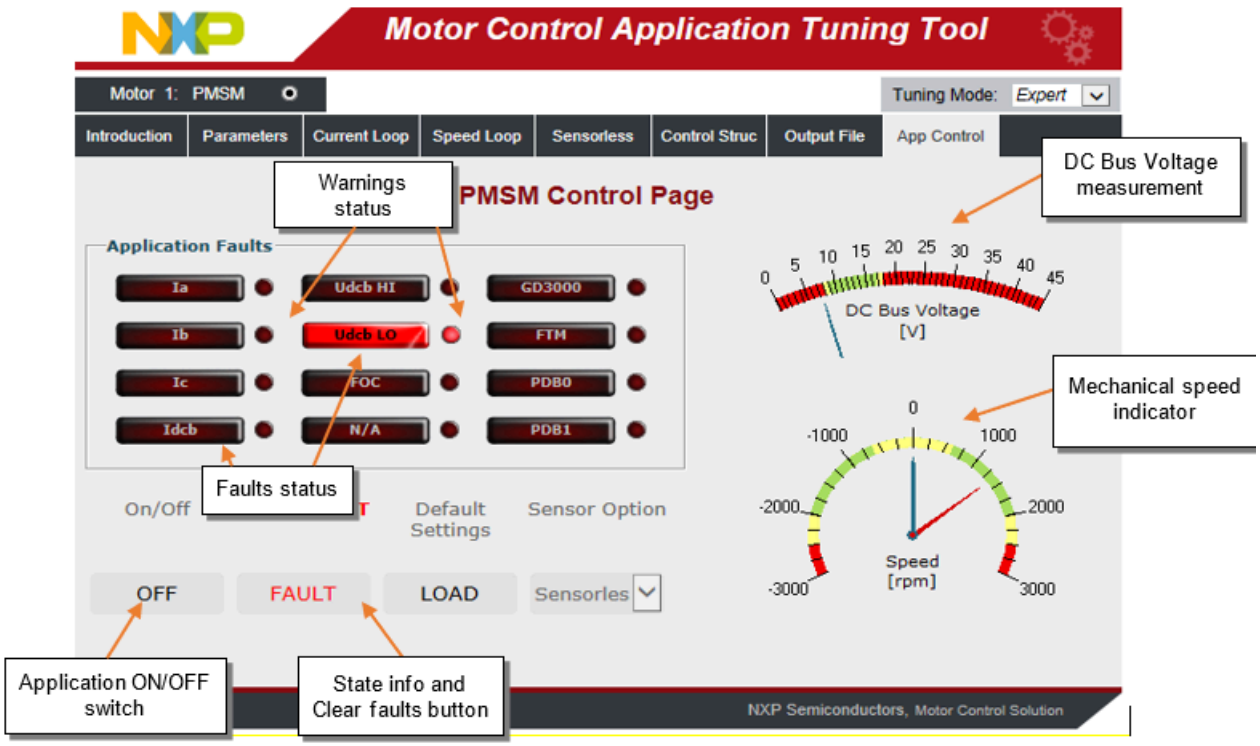

Figure 43. FreeMASTER MCAT control page for controlling the application

6 Testing and validation

Testing and validating of the power inverter unit is done in certified automotive laboratories using the dynamometer. The main target of all test runs is to analyze unit behavior and efficiency mapping over the full speed and torque range. Testing and validation targeted the first quadrant of PMSM machine operation. The test bench is visible in Figure 44.

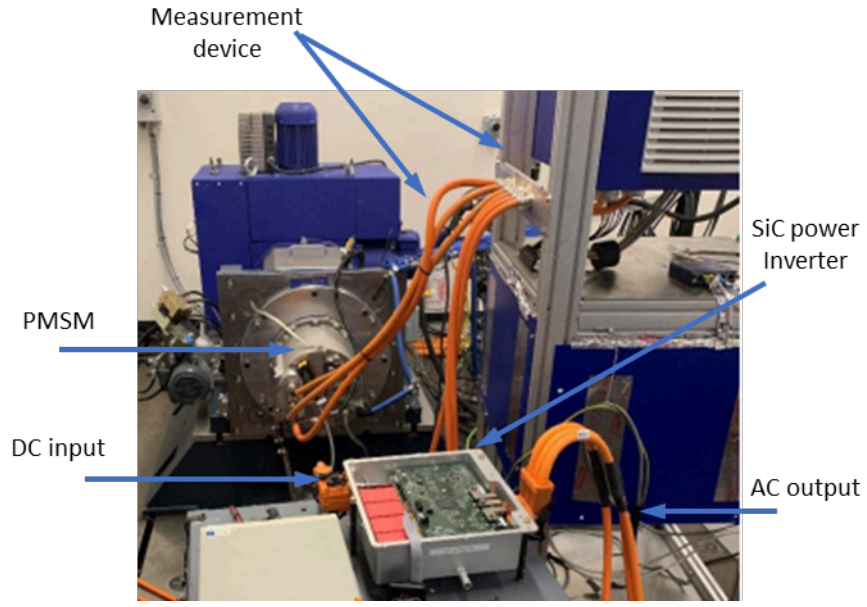

Figure 44. Test bench for testing and validation

The inverter unit was equipped with a standardized dq current loop picture. The power inverter module equipped with a standardized current loop used an IPMSM machine from VEPCO company. See Table 9, all-important parameters are in the table. Use a bi-directional Dc bus voltage source for all experiments. It is necessary to use bidirectional power supplies for all experiments.

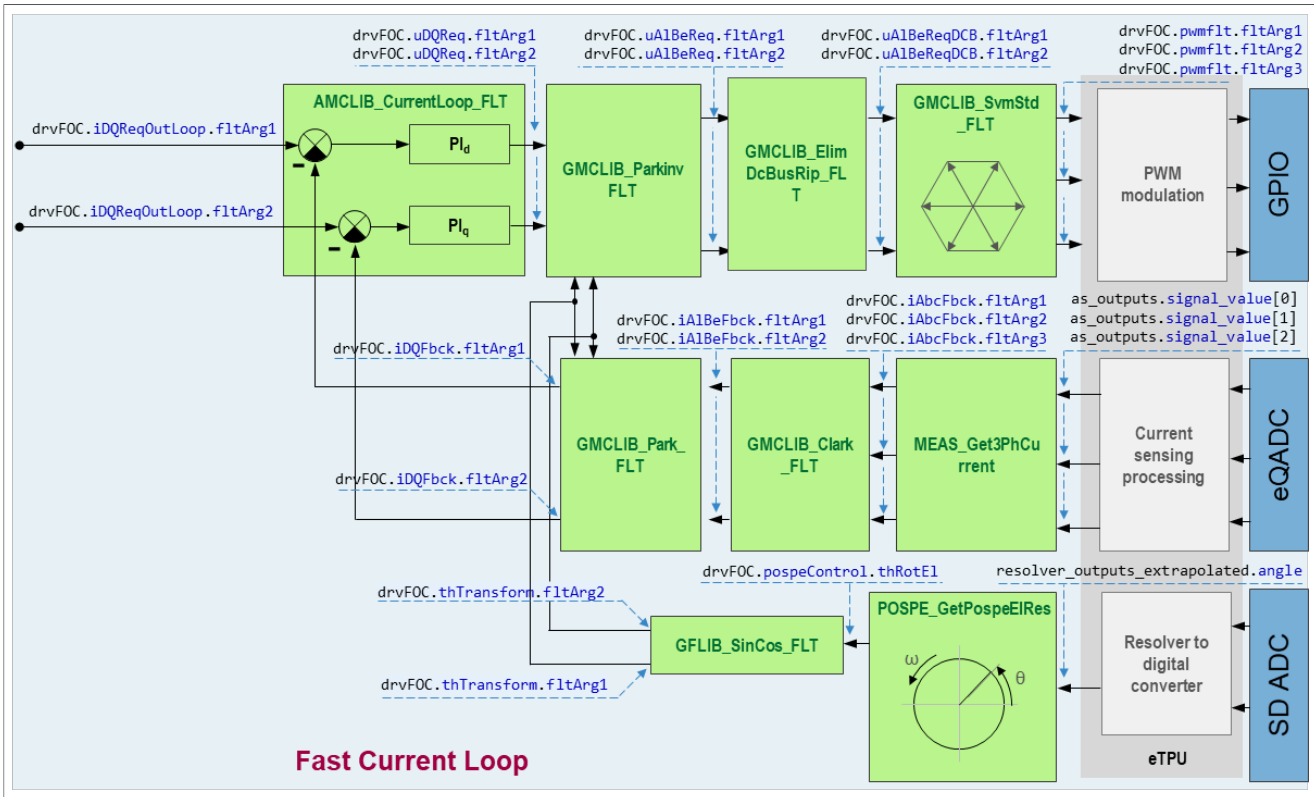

Figure 45. dq current control loop for efficiency mapping

The inverter control algorithm during validation procedures consists of a standardized FOC current loop. Testing vectors and torque commands were calculated on the host PC as shown in Figure 46. The host PC algorithm controls the overall dynamometer application and sends commands to the inverter via the CAN communication line. FreeMASTER provided data extension between Host PC and inverter. FreeMASTER as the gateway between the host PC and inverter, is also used for data collection and post-processing, targeting future analysis. Calculate the testing vectors offline. All vectors were targeted advanced control algorithms, especially MTPA, MTPV, and FW, based on PMSM machine mathematical model.

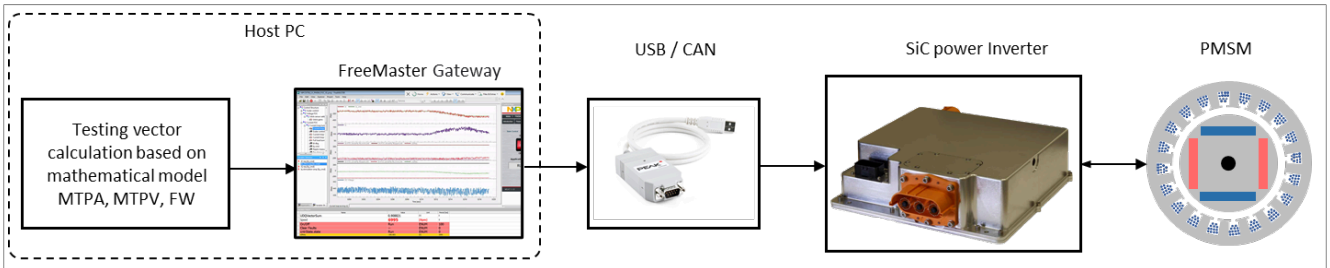

Figure 46. Block diagram for inverter efficiency mapping

Table 9. VEPCO IPMSM machine parameters

Nominal bus voltage	500	Max speed rpm	14000
Insulation class	200C	Base speed @nom Vbus/rpm	5750
Max Currents/Arms	425	Peak torque @max Current/Nm	320
Nom Current/Arms	230	Nom Torque/Nm	150
Peak Power/kW	192	Back EMF	736V @max speed
Nom power/kW	75	IP class	IP 67
Cool/Inlet Temp	Water/glycol 65C	pp	4

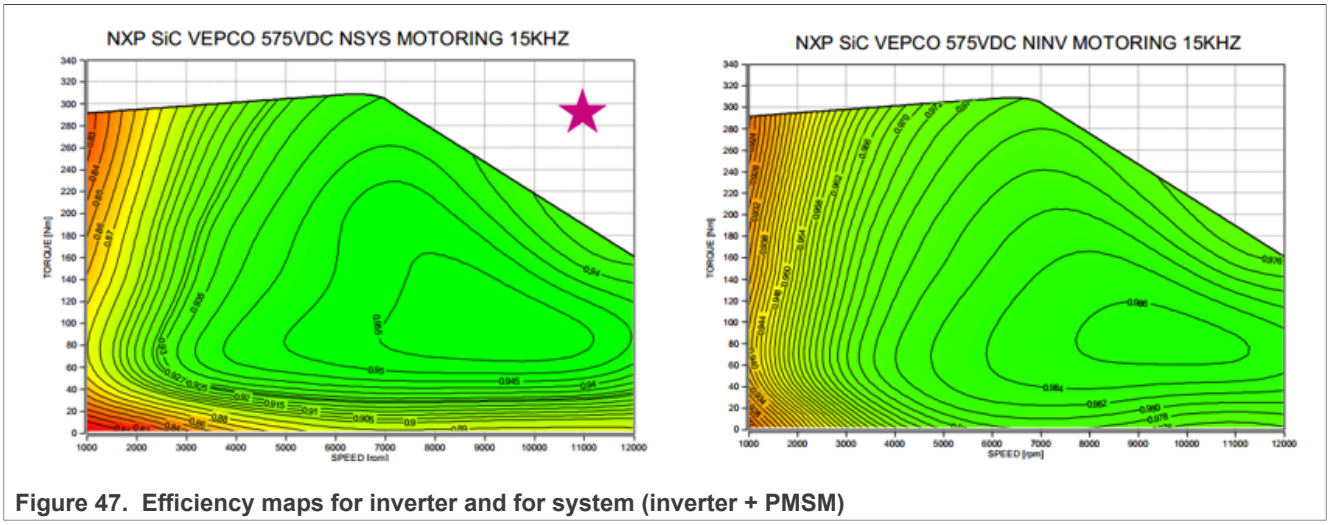

Figure 47. Efficiency maps for inverter and for system (inverter + PMSM)

The resulting efficiency maps are visible in the above figure. The data was measured in AVL laboratories. The following results represent efficiency maps for the whole system marked with a star and separately for the inverter. Data shows the following peak values under these specific conditions.

Table 10. Efficiency mapping results

Type of machine	VEPCO IPMSM
Switching frequency	15 kHz
DC bus voltage	575 V_{DC}
Speed range	0-12000 rpm
Peak Torque	307 Nm
Peak power mechanical	223 kW
Peak System Efficiency (PMSM + inverter)	96.1 %
Peak PMSM machine Efficiency	97.8 %
Peak Inverter efficiency	99.2 %

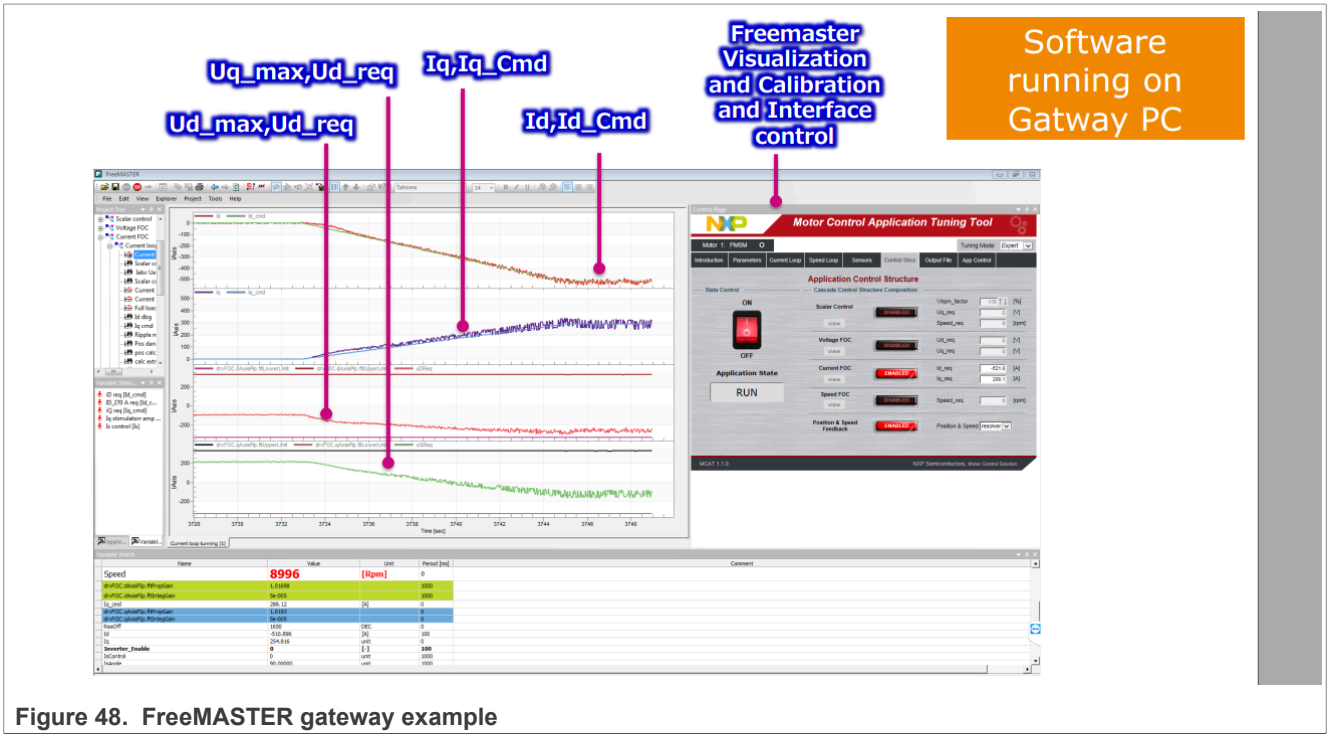

Figure 48. FreeMASTER gateway example

7 Conclusion

The design described in AN13879 application note shows the simplicity and efficiency of using the MPC5775E microcontroller and advanced gate driver GD3160 for PMSM motor control and introduces it as an appropriate candidate for various applications in the automotive area. MCAT tool provides an interactive online tool that makes the PMSM drive application tuning friendly and intuitive. The power inverter module design in application note AN13879 represents a starting point for future motor control development. Use the standardized FOC for getting system mapping. Collecting data for postprocessing and analyzing systems on-the-fly is made easy with the help of the online monitor, FreeMASTER. FreeMASTER can also serve as a gateway for data extension between the traction inverter and the host application.

8 References

- S32 Design Studio IDE for ARM® based MCUs

- FreeMASTER Run-Time Debugging Tool
- Automotive Math and Motor Control Library Set for MPC577xC
- MPC5777C Reference Manual
- Rashid, M. H. Power Electronics Handbook, 2nd Edition. Academic Press
- Motor Control Application Tuning (MCAT) Tool
- eTPU RDC and RDC Checker User Guide

9 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

10 Revision history

This table summarizes the revisions to this document.

Table 11. Revision history

Document ID	Release date	Description
AN13879 v.1.0	16 January 2024	Initial release

11 Legal information

11.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

11.2 Disclaimers

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.

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 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.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Suitability for use in automotive applications — This NXP product has been qualified for use in automotive applications. If this product is used by customer in the development of, or for incorporation into, products or services (a) used in safety critical applications or (b) in which failure could lead to death, personal injury, or severe physical or environmental damage (such products and services hereinafter referred to as "Critical Applications"), then customer makes the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, safety, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. As such, customer assumes all risk related to use of any products in Critical Applications and NXP and its suppliers shall not be liable for any such use by customer. Accordingly, customer will indemnify and hold NXP harmless from any claims, liabilities, damages and associated costs and expenses (including attorneys' fees) that NXP may incur related to customer's incorporation of any product in a Critical Application.

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.

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

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. 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.

NXP B.V. - NXP B.V. is not an operating company and it does not distribute or sell products.

11.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Tables

Figures

Contents

For more information, please visit: http://www.nxp.com

Date of release: 16 January 2024
Document identifier: AN13879