NXP Semiconductor
Application Notes

Document Number: AN5327
Rev. 0, 08/2016

3-phase Sensorless Single-Shunt Current-Sensing PMSM Motor Control Kit with MagniV MC9S12ZVM

Featuring Motor Control Application Tuning (MCAT) Tool

Contents

1 Introduction

This application note is targeted for automotive applications and describes the design of a 3-phase Permanent Magnet Synchronous Motor (PMSM) vector control drive with single shunt current sensing without a position sensor.

This cost-effective solution benefits from MC9S12ZVM device dedicated for motor control. The system is designed to drive a 3-phase PM synchronous motor. Following are the supported features:

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

2 System concept

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

- Targeted at the MC9S12ZVM Evaluation Board (refer to dedicated user manual for MC9S12ZVM available at www.nxp.com) See section References for more information.
- Control technique incorporating:
  - Field Oriented Control of 3-phase PM synchronous motor without position sensor
  - Closed-loop speed control
  - Bi-directional rotation
  - Close-loop current control
  - Flux and torque independent control
  - Position and speed is estimated by Extended BEMF observer.
  - Start up with alignment
  - Reconstruction of three-phase motor currents from a single shunt resistor
  - 100 µs sampling period with FreeMASTER recorder
- FreeMASTER software control interface (motor start/stop, speed setup)
- FreeMASTER software monitor
- FreeMASTER embedded Motor Control Application Tuning (MCAT) tool (motor parameters, current loop, sensorless parameters, speed loop)
- FreeMASTER software MCAT graphical control page (required speed, actual motor speed, start/stop status, DC-Bus voltage level, motor current, system status)
- FreeMASTER software speed scope (observes actual and desired speeds, DC-Bus voltage and motor current)
- FreeMASTER software high-speed recorder (reconstructed motor currents, vector control algorithm quantities)
- DC-Bus over-voltage and under-voltage, over-current, overload and start-up fail protection.

3 PMSM field oriented control

3.1 Fundamental principle of PMSM FOC

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

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

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

Equation 1

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

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

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

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

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

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

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

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

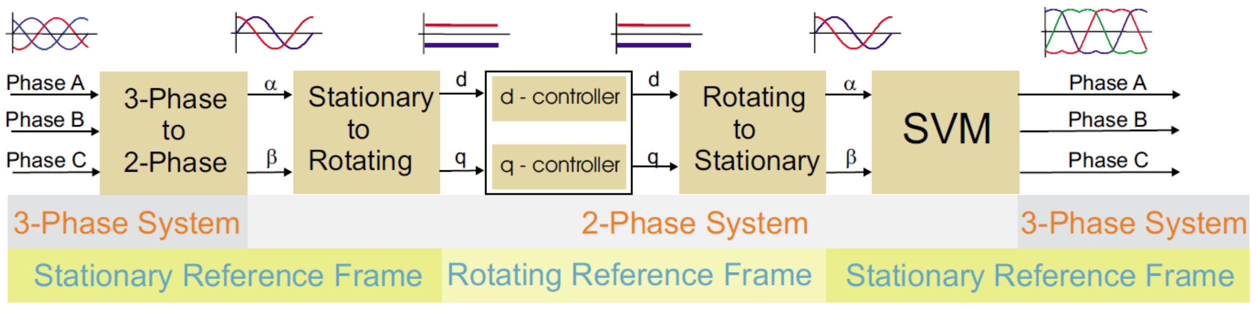


Figure 1. Field oriented control transformations

3.2 PMSM model in quadrature phase synchronous reference frame

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

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

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

Equation 2

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

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

Equation 3

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

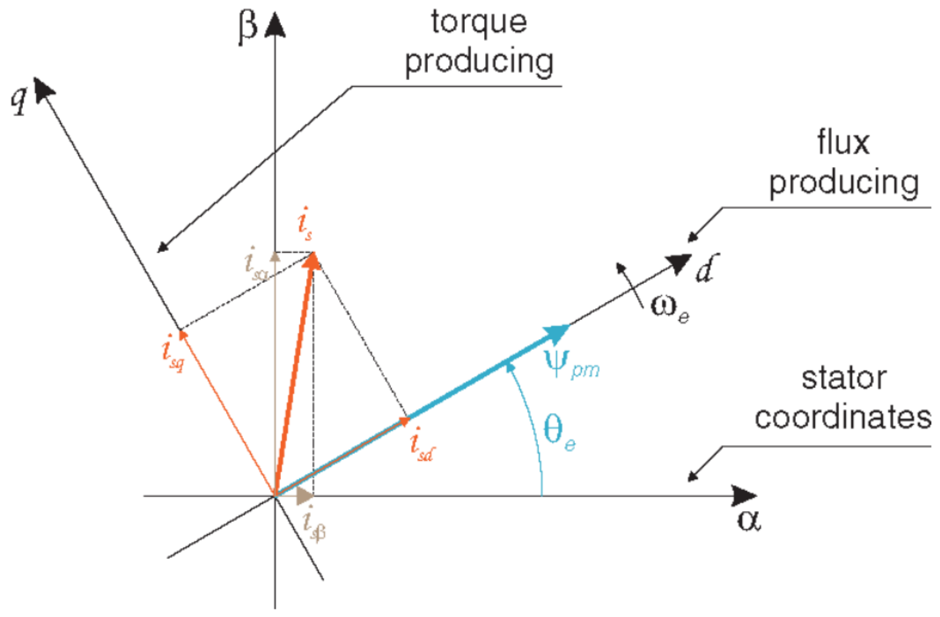


Figure 2. Orientation of stator (stationary) and rotor (rotational) reference frames, with current components transformed into both frames

The voltage equation of the quadrature phase synchronous reference frame model can be obtained by transforming the three phase voltage equations (*Equation 1*, and *Equation 2*) into a two phase rotational frame which is aligned and rotates synchronously with the rotor as shown in *Figure 2*. Such transformation, after some mathematical corrections, yields the following set of equations:

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

Equation 4

In the above equation ω_e is electrical rotor speed. It can be seen that Equation 4

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

after compensation:

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

Equation 5

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

Equation 6

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

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

Equation 7

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

Equation 8

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

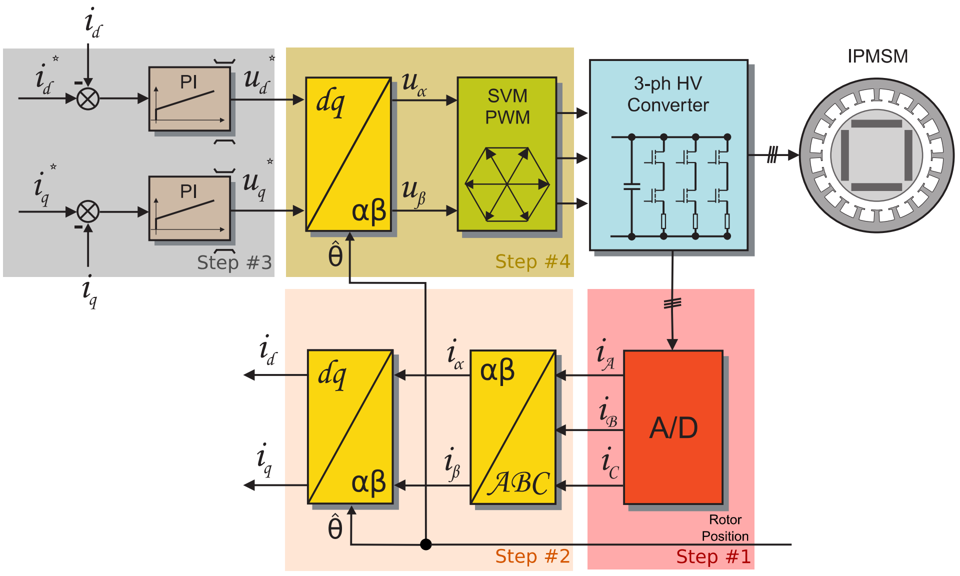


Figure 3. FOC Control Structure

3.3 Output voltage actuation and phase current measurement

The 3-phase voltage source inverter shown in *Figure 4* uses single DC-bus shunt resistor (R71, other shunt resistors are not used). DC-bus current which flows through the shunt resistor produces a voltage drop which is interfaced to the AD converter of microcontroller through conditional circuitry (refer to MC9S12ZVML128 Evaluation Board User Manual available at nxp.com).

3.3.1 DC-link current sensing

Since the only current being measured is the DC-bus current, stator phase currents must be reconstructed based on at least two DC-bus current samples captured at a different time based on the combination of transistors shown in *Figure 6.*

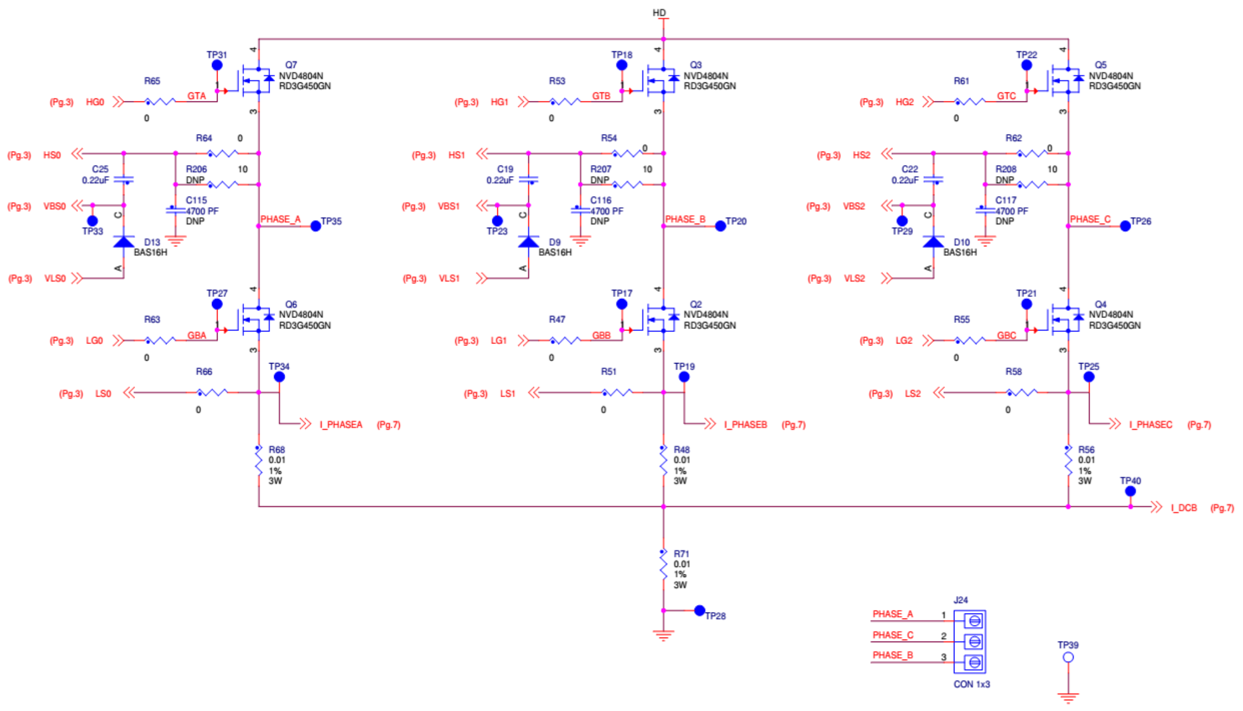


Figure 4. 3-phase DC/AC inverter with shunt resistors for current measurement

Figure 5 shows gain setup and input signal filtering circuit for two internal operational amplifier integrated on MC9S12ZVM which provides the conditional circuitry and adjusts voltages to fit into the ADC input voltage range.

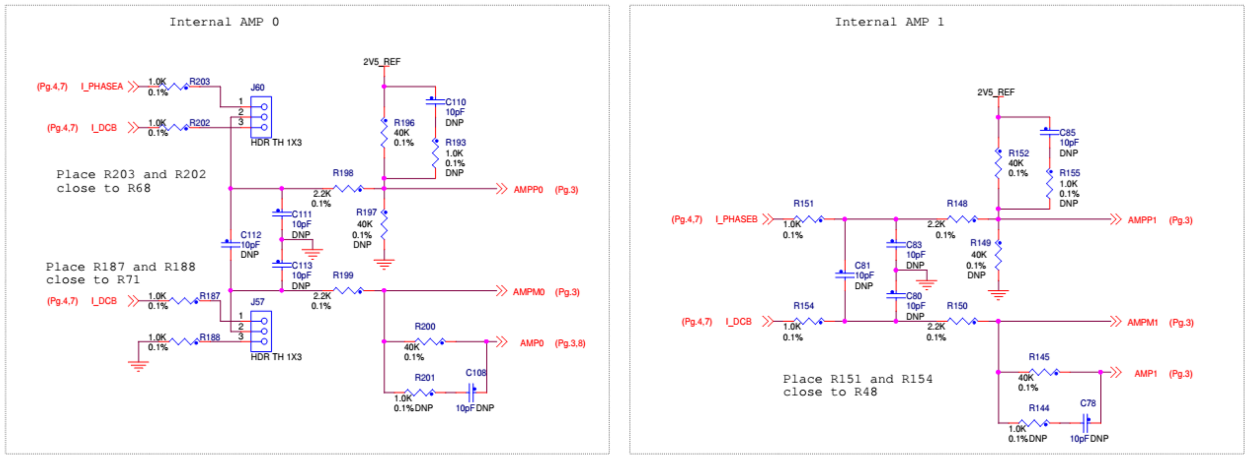

Internal AMP 0
Place R203 and R202 close to R68
Place R187 and R188 close to R71
Internal AMP 1
Place R151 and R154 close to R48

Figure 5. Phase current measurement conditional circuitry

3.3.2 Phase currents reconstruction

The phase current sampling technique is a critical issue for detection of phase current differences and for acquiring full three phase information of stator current by its reconstruction. Phase current flowing through a shunt resistor produces a voltage drop which needs to be appropriately sampled by the AD converter when the DC bus voltage is connected to the motor, thus in six of eight (non-zero) voltage vectors (see *Figure 6*).

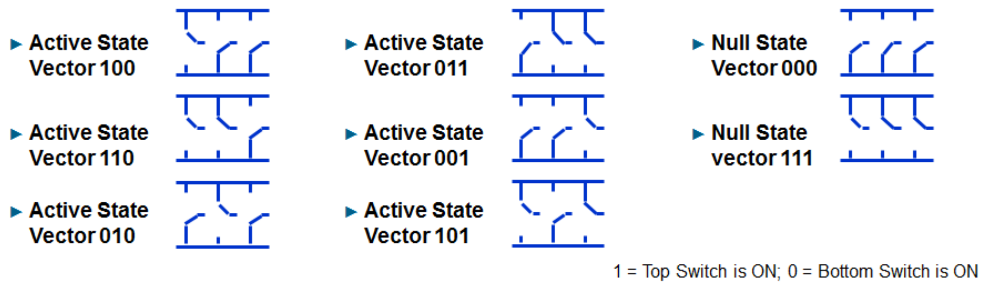


Figure 6. Voltage vector states in terms of transistor states

Figure 7 shows an example of a current measurement during vector 101, in which i_{SB} sample can be taken. Considering a symmetrical 3-phase system, Kirchhoff law can be used at any time, thus

$$i_{SA} + i_{SB} + i_{SC} = 0$$

Equation 9

Based on the *Equation 9*, at least two currents in a single PWM period are needed to have all the three currents available for the vector control. Thanks to the modulation of the voltage vector, two different combinations of non-zero voltage vectors are available during a single PWM period. Thus, two currents can be sensed as shown in *Figure 8*. The third current is then calculated based on *Equation 9*.

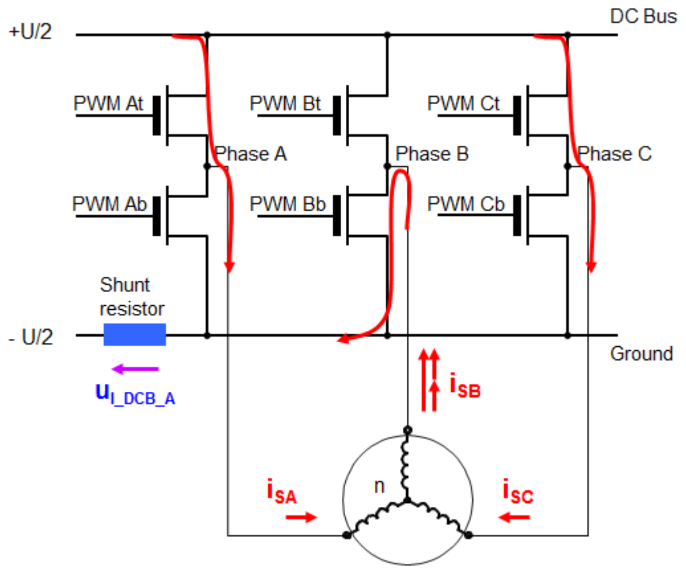


Figure 7. Single-shunt three-phase current reconstruction

Single-shunt 3-phase current reconstruction is available only if two voltage vectors are active for a sufficient time period to capture the current. As two PWM edges come close to each other, phase current signal pulse on the DC-link becomes too short to be captured or “disappears” at all, as shown in *Figure 9*. This makes that portion of 3-phase current information invisible for sensing circuit and can eventually disturb the phase current feedback. If all three phases come close enough, no phase current information can be recovered from the DC-link current sensor.

There are two main techniques to make the 3-phase current reconstruction available at any time. The first one, so called “phase shifting PWM” shifts one of the overlapping phases from another to make the DC-link current visible for a sufficient period of time. This method is described e.g. in Design reference manual DRM102 available at www.nxp.com.

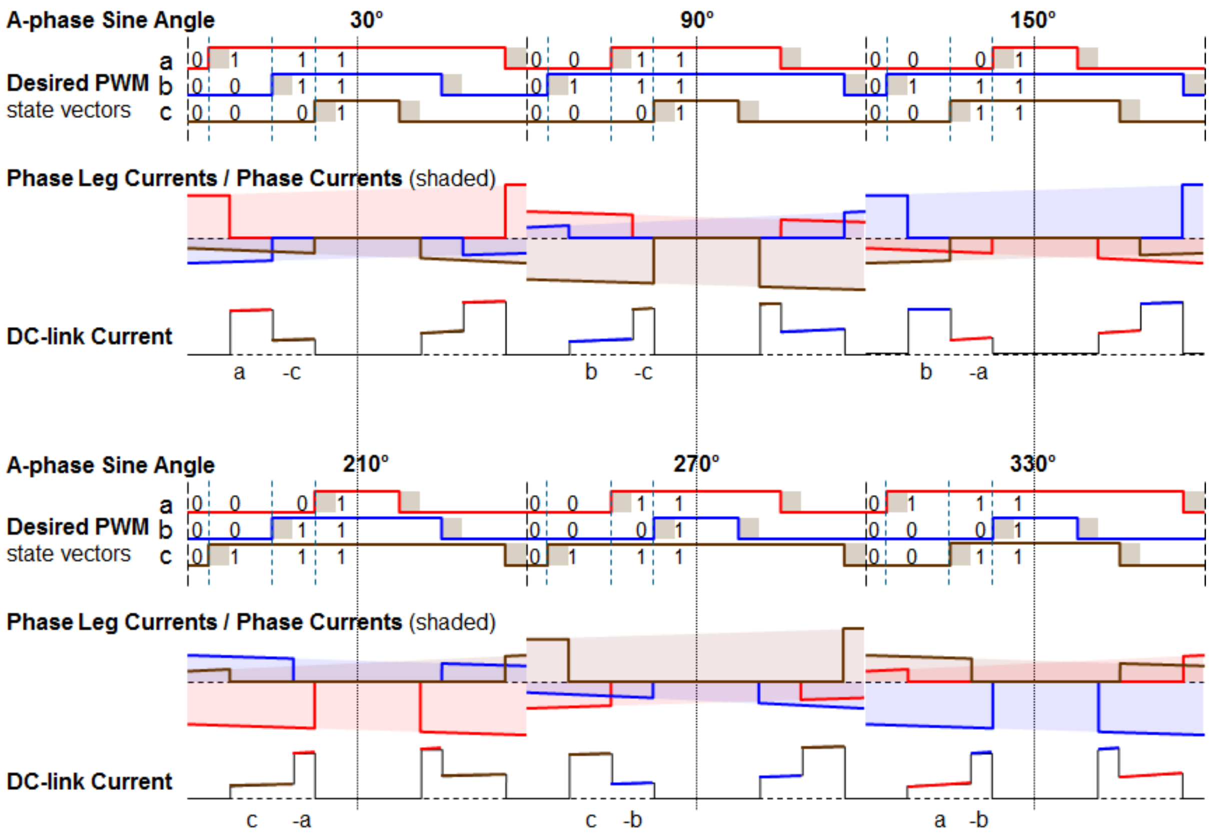


Figure 8. DC-link current and phase currents connection

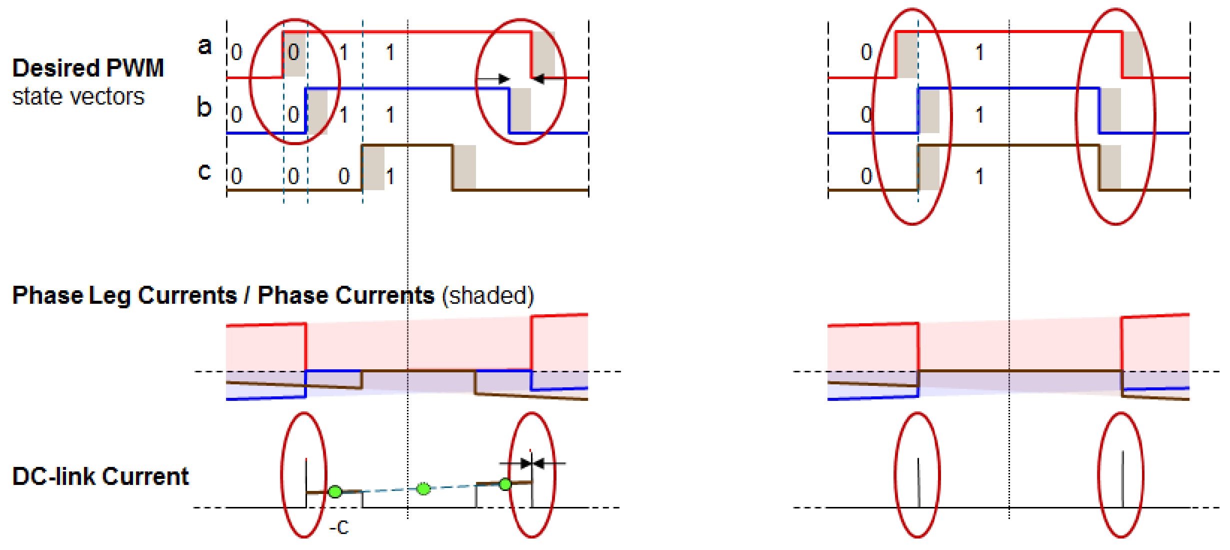


Figure 9. Single-shunt current sensing limits during two phase edges overlapping

Another option is to split one of the overlapped signals in two parts, thus insert a zero pulse in the middle of the pulse (see *Figure 10*, blue signal on the left, blue and brown signals on the right), so called "double switching PWM". Instead of shifting one signal from another, one of the overlapped signals is split in two symmetrical signals and these two parts are shifted apart (*Figure 10* on the left, the blue signal) so the total length of the signal is the same (in comparison to phase A). However, there is a different number of switching operations in phase B. Considering a different number of dead-times inserted, the output voltage of the double-switched phase is lower. Another impact of such double switching is a different voltage vector being injected into the motor. These disturbances may cause a harmonic distortion to the flux and a sound noise.

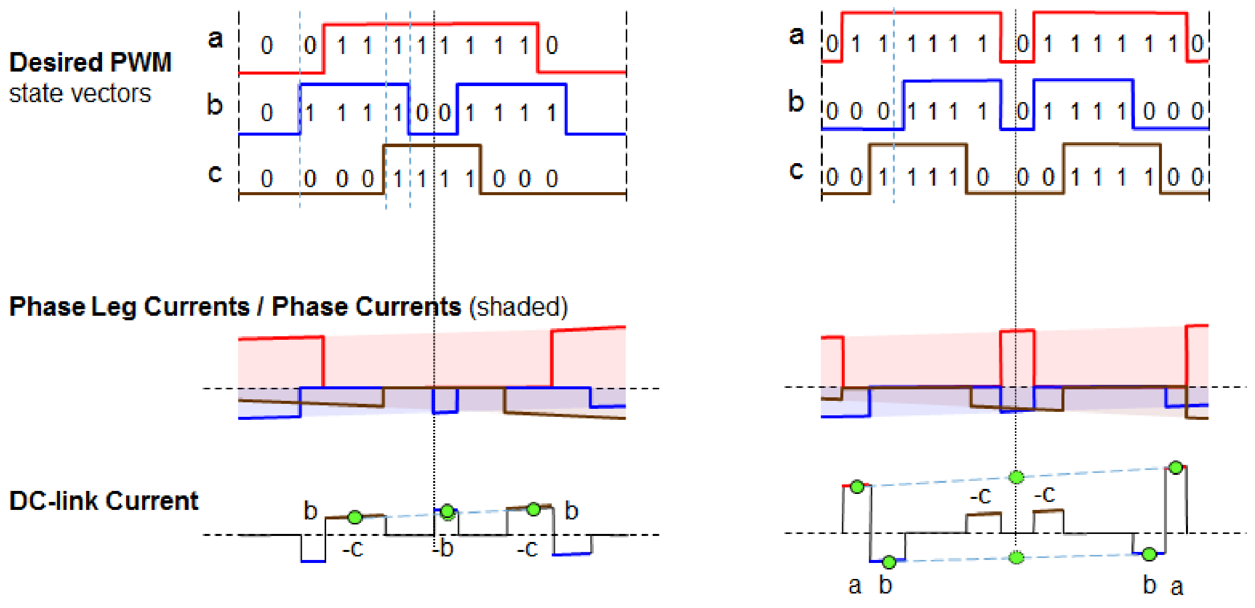


Figure 10. Double switching PWM and 3-phase sensing opportunities

To lower the noise and losses during double switching, all three signals are split in two parts, whilst one of the signals has the two parts shifted apart by a longer time span (*Figure 10* on the right, the brown signal is shifted from the blue signal). The unnecessary voltage vector (110) is switched for two short periods of time including a zero-voltage vector and the negative impact of double switching is reduced. The main disadvantage of such concept is the duty cycle is limited to approximately 93%.

Thanks to double switching, two samples for each current are available, an average value can be calculated (see *Figure 10*).

3.3.3 Output voltage actuation

Generated phase signals based on duty cycles (phase A, phase B, phase C) of two PWM periods are shown in *Figure 11*. These phase voltage waveforms correspond to a center-aligned double switched PWM with a space vector modulation input.

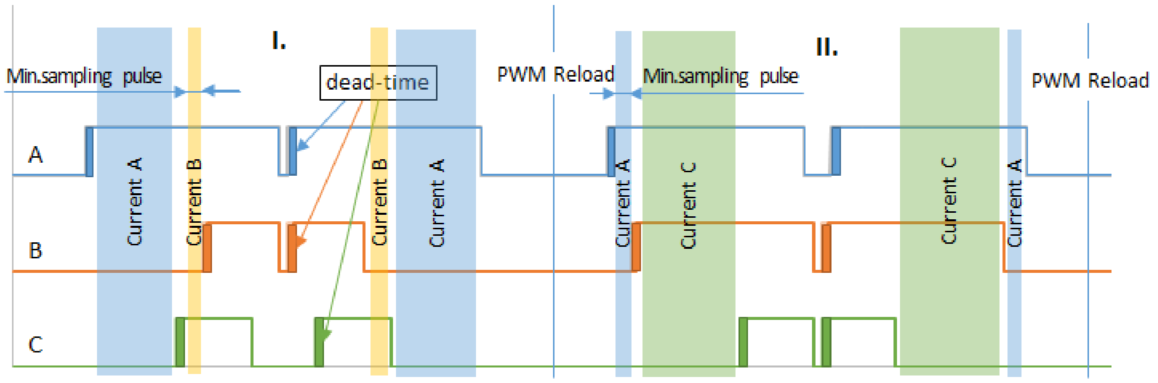


Figure 11. Double-switching PWM and phase currents visibility

Figure 11 shows two periods of PWM. In period I., phase B and phase C are of the same duty cycle, but shifted to enable three phase current reconstruction from a single shunt current signal. The width of the space between two pulses of phase C impacts the minimal sampling pulse width for current B sensing. Period II. introduces similar situation with phase A and B.

To benefit from the double switching feature of the PMF module, timings of the PWM signals edges are calculated. The algorithm (SetDutyCycle (SWLIBS_3Syst_F16 *f16pwm, tU16 sector) function in “actuate_s12zvm.c” module) calculates the edges based on 3-phase SVM generated reference voltage (f16pwm) and current sector within the voltage hexagon (shown in *Figure 12*, SVM duty cycles are black dashed lines, double switched signals are blue, orange and green).

First of all, for actual sector, zero pulses are calculated based on the difference between reference voltages (diffUV, diffVW, diffWU) using minZeroPulse and minSamplingPulse. These voltages can be calculated as a difference of the duty cycles.

$$diffUV = pwm_U - pwm_V$$

$$diffVW = pwm_V - pwm_W$$

$$diffUW = pwm_U - pwm_W$$

Equation 10

If one of the differences is less than two minimal sampling pulses, that means there is no space for standard sensing of two currents between these two phases and the middle zero pulse is calculated. Otherwise a minimal zero pulse is added (given by minZeroPulse, considering all three phases will be double-switched). The duty cycle extension of each phase is stored in pwmB structure.

Following set of equations considers sector 1 of the voltage hexagon. The first duty cycle extension goes for the phase C, adding the minimal zero pulse only.

$$pwmB.f16Arg3 = minZeroPulse$$

Equation 11

The second duty cycle extension of the phase B is calculated based on the condition below.

$$diffVW < 2 \cdot minSamplingPulse$$

$$\begin{cases} \boldsymbol{true}: \ pwmB.f16Arg2 = (2 \cdot minSamplingPulse + minZeroPulse - diffVW) \\ \boldsymbol{false}: pwmB.f16Arg2 = minZeroPulse \end{cases}$$

Equation 12

Now if there is not enough space for taking two samples of the current, in other words, if the difference between V and W reference voltages is less than two sampling pulses, the pwmB.f16Arg2 is calculated to extend the phase B duty cycle by 2 minimal sampling pulses and the minimal zero pulse. That means, the pulse will be split in two pulses of the same length and shifted apart with minimal zero pulse in between. Phase C is affected by the minZeroPulse only.

The third duty cycle adds two additional minimal sampling pulses to the previous calculation if necessary to ensure the second phase current can be sampled. That means, the first rising edge of the phase A will be shifted by one minimal sampling pulse from the first rising edge of the phase B. The second sampling pulse is put between the last falling edges of the A and B phase signals.

$$temp = 2 \cdot minSamplingPulse + pmwB.f16Arg2$$

$$diffUV < (temp - minZeroPulse) \begin{cases} \boldsymbol{true}: \ pwmB.f16Arg1 = (temp - diffUV) \\ \boldsymbol{false}: pwmB.f16Arg1 = minZeroPulse \end{cases}$$

Equation 13

NOTE

Minimal zero pulse should allow the transistors to fully switch off and on. Minimal sampling pulse is given by typical time between two ADC conversions (given by minSamplingPulse). For S12ZVM ADC device running at 8.33 MHz frequency the minimal sampling pulse is 2.5 µs.

```
tFrac16      minZeroPulse        = FRAC16(3.8/100.0); // ~ 1.9 us
tFrac16      minSamplingPulse    = FRAC16(5.0/100.0); // ~ 2.5 us

//switch(sector) example for sector 1
// ...
case 1:
      diffUV   = MLIB_Sub_F16(f16pwm->f16Arg1,f16pwm->f16Arg2);
      diffVW   = MLIB_Sub_F16(f16pwm->f16Arg2,f16pwm->f16Arg3);
```

```
        pwmB.f16Arg3 = minZeroPulse;
        pwmB.f16Arg2 = (diffVW)<minSamplingPulse_x2) ? MLIB_SubSat_F16(pulseSum,diffVW)
                                                     : minZeroPulse;
        temp         = MLIB_AddSat_F16(pwmB.f16Arg2,minSamplingPulse_x2);
        pwmB.f16Arg1 = ((diffUV)<MLIB_Sub_F16(temp,minZeroPulse)) ?
                                                MLIB_SubSat_F16(temp,diffUV)
                                              : minZeroPulse;
        CalcEdges(&pwm3PhEdges, f16pwm, &pwmB);
```

The edges for double-switched PWM are calculated in CalcEdges() by adding pwmB values to the SVM generated duty cycles and stored to pwmA structure. Four edges for each PWM channel are calculated; pwmA and pwmB values are divided by 2 and added to or subtracted from the half of the PWM period respectively. The structure pwm3PhEdges contains three phase structures phA to phC, which contain two modulus structures modA and modB of two edges each. The PMF module value registers are given by first edges of the pulses only. Finally, the PMF double switching logic generates the signals including a deadtime. More detailed information on the double switching feature can be found in MC9S12ZVMRMV1, MC9S12ZVM-Family Reference Manual available at nxp.com.

```
void CalcEdges(PMF_3PH_MODULATOR_T *pwm3PhEdges, SWLIBS_3Syst_F16 *duty,
SWLIBS_3Syst_F16 *pwmB)
{
        SWLIBS_3Syst_F16    pwmAhalf,pwmBhalf;
        SWLIBS_3Syst_F16    pwmA;

        pwmA.f16Arg1 = MLIB_AddSat_F16(duty->f16Arg1,pwmB->f16Arg1);
        pwmA.f16Arg2 = MLIB_AddSat_F16(duty->f16Arg2,pwmB->f16Arg2);
        pwmA.f16Arg3 = MLIB_AddSat_F16(duty->f16Arg3,pwmB->f16Arg3);

        pwmAhalf.f16Arg1 = pwmA.f16Arg1>>1;
        pwmAhalf.f16Arg2 = pwmA.f16Arg2>>1;
        pwmAhalf.f16Arg3 = pwmA.f16Arg3>>1;

        pwmBhalf.f16Arg1 = pwmB->f16Arg1>>1;
        pwmBhalf.f16Arg2 = pwmB->f16Arg2>>1;
        pwmBhalf.f16Arg3 = pwmB->f16Arg3>>1;

        /* ph A */
        pwm3PhEdges->phA.modA.firstEdge   = 0x3FFF - pwmAhalf.f16Arg1;      //1st rising
        pwm3PhEdges->phA.modA.secondEdge  = 0x3FFF + pwmAhalf.f16Arg1;      //4th falling
        pwm3PhEdges->phA.modB.firstEdge   = 0x3FFF - pwmBhalf.f16Arg1;      //2nd falling
        pwm3PhEdges->phA.modB.secondEdge  = 0x3FFF + pwmBhalf.f16Arg1;      //3rd rising

        /* ph B */
        pwm3PhEdges->phB.modA.firstEdge   = 0x3FFF - pwmAhalf.f16Arg2;
        pwm3PhEdges->phB.modA.secondEdge  = 0x3FFF + pwmAhalf.f16Arg2;
        pwm3PhEdges->phB.modB.firstEdge   = 0x3FFF - pwmBhalf.f16Arg2;
        pwm3PhEdges->phB.modB.secondEdge  = 0x3FFF + pwmBhalf.f16Arg2;

        /* ph C */
        pwm3PhEdges->phC.modA.firstEdge   = 0x3FFF - pwmAhalf.f16Arg3;
        pwm3PhEdges->phC.modA.secondEdge  = 0x3FFF + pwmAhalf.f16Arg3;
        pwm3PhEdges->phC.modB.firstEdge   = 0x3FFF - pwmBhalf.f16Arg3;
        pwm3PhEdges->phC.modB.secondEdge  = 0x3FFF + pwmBhalf.f16Arg3;
}
```

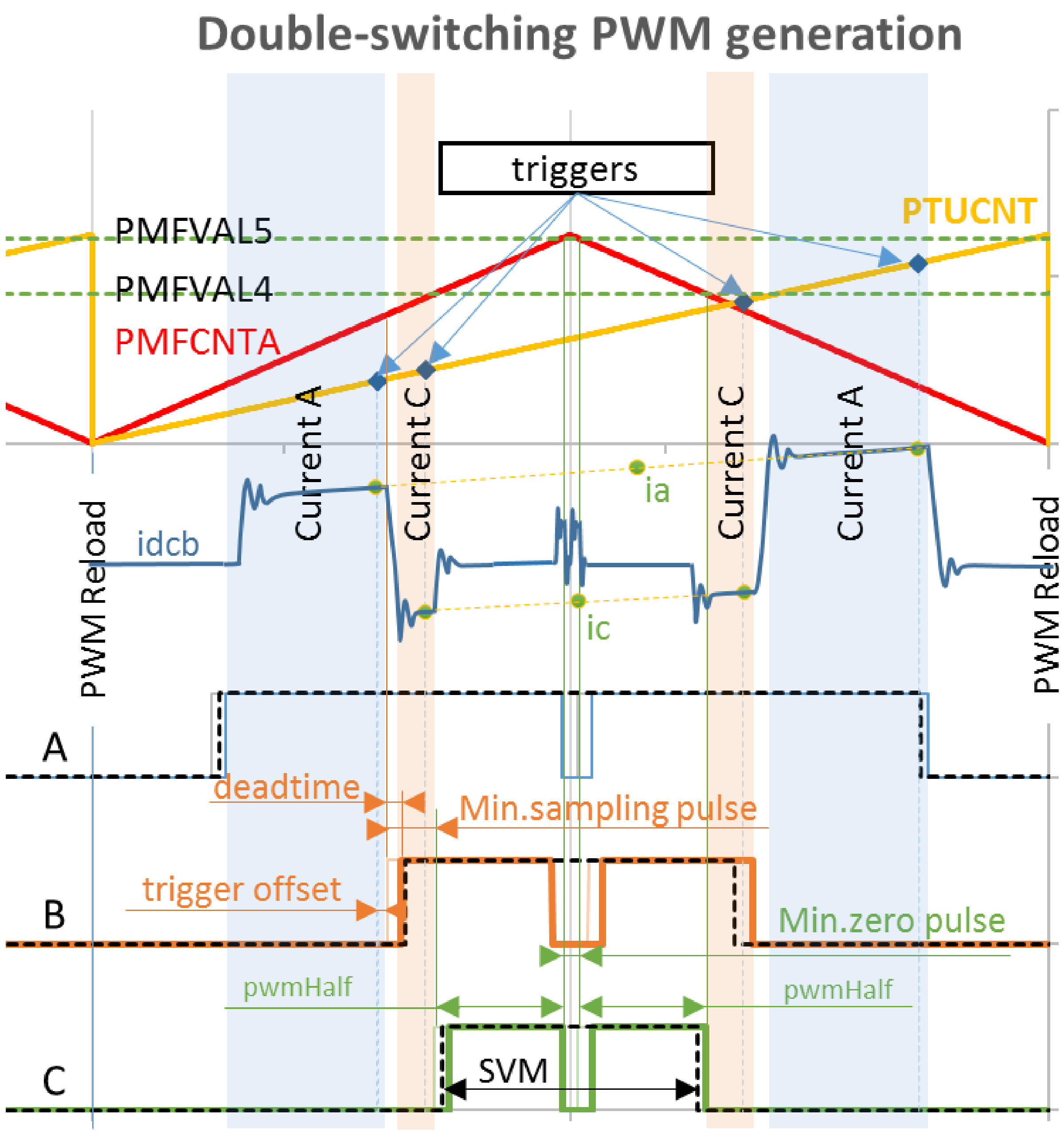


Figure 12. Double switching edges calculation

As shown in *Figure 12* and as explained in *3.3.2*, the best sampling instants of phase currents are given by the combination of active vectors, when the currents are "visible" by the DC-link shunt resistor (a window, blue and orange shaded areas of *Figure 12*). However, the currents cannot be measured at an arbitrary voltage shape. The current signal is distorted by switching of the transistors, therefore it is essential to capture the signal by the end of the window, when the signal is stable. The PTU trigger to the ADC module is based on the edges, but it also involves a delay of the system, which includes GDU delay, MOSFET switch on/off time and ADC conversion initiation time (triggerOffset1 to triggerOffset4 values are added to the triggers in SetPtuTriggers() in "actuate_s12zvm.c" module, set to -25).

Two phase currents in each PWM period are measured two times, thus four triggers are issued. Since the best time to start sampling is at the end of the falling edge of the PWM, the edges calculated in CalcEdges() module are used, based on current sector of the voltage vector. For example of sector 1, the triggers are calculated as follows.

```
       trg[1] = pwm3PhEdges.phB.modA.firstEdge;
       trg[2] = pwm3PhEdges.phC.modA.firstEdge;
       trg[3] = pwm3PhEdges.phC.modB.secondEdge;
       trg[4] = pwm3PhEdges.phB.modA.secondEdge;
       trg[5] = pwm3PhEdges.phA.modA.secondEdge;
…
       trigs.ph1Trg1        = (tU16) MLIB_Mul_F16(trg[1],PMFMODA);
       trigs.ph2Trg1        = (tU16) MLIB_Mul_F16(trg[2],PMFMODA);
       trigs.dcOffsetTrg    = (tU16) PMFMODA;
       trigs.ph2Trg2        = (tU16) MLIB_Mul_F16(trg[4],PMFMODA);
       trigs.ph1Trg2        = (tU16) MLIB_Mul_F16(trg[5],PMFMODA);
```

The first trigger (*Figure 12*) is set to the first rising edge of the phase B, when the current A can be measured with the lowest noise. Later on, the trigger is shifted left by triggerOffset1 to allow the ADC to sample and hold the value. The second trigger is the first rising edge of the phase C, when the current C is visible to the DC-link current sensor with the lowest noise.

The third trigger in the list is not used, but it is an example how to trigger the ADC at the point when the DC-link sensor offset can be sensed.

The fourth trigger is again the phase C sensing period, linked to the falling edge of the phase B. The fifth trigger connected to the phase A sensing period is linked to the falling edge of the phase A.

The final trigger values are multiplied with the PMFMODA value, the PWM modulus, to extend the PMF related values to the 16-bit value used by the PTU counter register.

All the triggers are shifted left by triggerOffsetX value to enable the ADC to sample and hold the value, as described before. The shift is performed in the SetPtuTriggers() module. The code snapshot below switches the list of PTU triggers first, then sets the trigger list values to the trigger values calculated above, shifted by the triggerOffsetX value. The last trigger value 0x00 ends the trigger sequence and prepares the PTU for the next reload. The triggerOffsetX values are set to -25, which means 0.5µs shift as the best option for the current setting of the development kit. Custom solutions should consider changing the value to achieve the best trigger placement.

```
void SetPtuTriggers(PTU_TRIGGERS_T *pTrg)
{
    writeToList = (*(volatile tU8 *)(0x0580 + 0x0006)) & 0x01;
    writeToList  ^= (1 << 0);

    ptuTriggerList0[writeToList][0] = pTrg->ph1Trg1 + triggerOffset1;
    ptuTriggerList0[writeToList][1] = pTrg->ph2Trg1 + triggerOffset2;

    //ptuTriggerList0[writeToList][2] = pTrg->dcOffsetTrg + triggerOffsetR;

    ptuTriggerList0[writeToList][2] = pTrg->ph2Trg2 + triggerOffset3;
    ptuTriggerList0[writeToList][3] = pTrg->ph1Trg2 + triggerOffset4;
    ptuTriggerList0[writeToList][4] = 0x00;                        // End Of List
}
```

Figure 12 shows the calculated average values of currents ia and ic (green dots). The average values are shifted from each other and from the center of PWM period. Since the phase currents are usually changing very slowly over single PWM period, the error is negligible. However, the right placement of the average value can be easily achieved by linear interpolation of the sampling points given by the PTU triggers and the sampled values.

3.4 Rotor position/speed estimation

The first stage of the proposed overall control structure is alignment algorithm of rotor PM to set an accurate initial position. The alignment algorithm applies DC voltage to phase A for a certain period. This will cause the rotor to move to "align" position, where stator and rotor fluxes are aligned. The rotor position in which the rotor stabilizes after applying DC voltage is set as zero position. The alignment algorithm allows applying a full startup torque to the motor.

In the second stage, the field-oriented control is in open-loop mode, in order to move the motor up to a speed value where the observer provides sufficiently accurate speed and position estimations. As soon as the observer provides appropriate estimates, the rotor speed and position calculation is based on the estimation of a BEMF in the stationary reference frame using a Luenberger type of observer.

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

This estimation method for the position and angular speed is based on the motor mathematical model with an extended electromotive force function. This extended BEMF model includes both position information from the conventionally defined BEMF and the stator inductance. This enables extraction of the rotor position and velocity information by estimating the extended BEMF only.

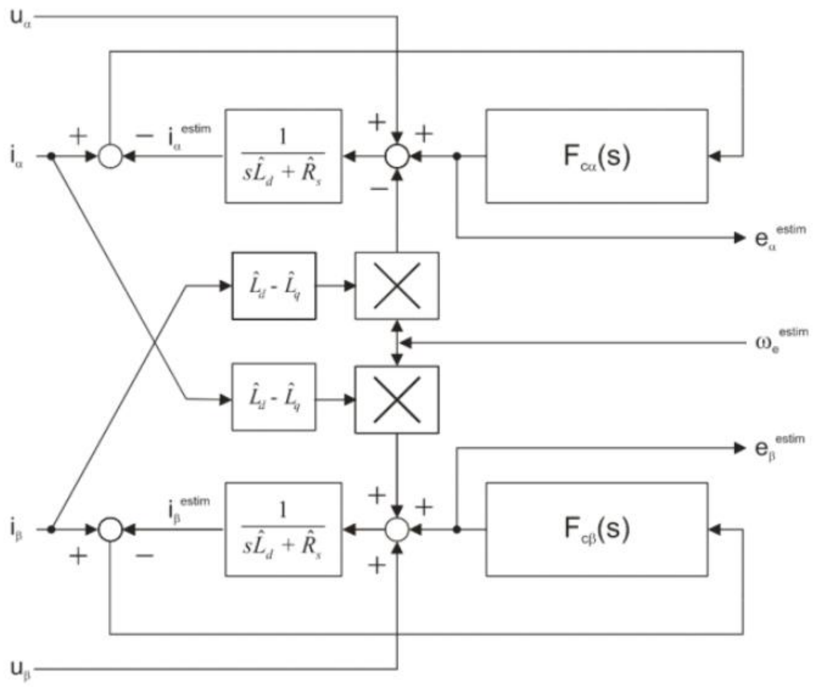


Figure 13. Luenberger type stator current observer acting as state filter for BEMF

The observer is applied to PMSM motor with an estimator model excluding the extended BEMF term. Then the extended BEMF term can be estimated using the observer as shown in *Figure 13*, which uses a simple observer of PMSM stator current. The BEMF observer presented here is realized within the rotating reference frame (*dq*). The estimator of dq-axis consists of the stator current observer based on RL motor circuit with estimated motor parameters. This current observer is fed by the sum of the actual applied motor voltage, cross-coupled rotational term, which corresponds to the motor saliency (*Ld-Lq*), and compensator corrective output. The observer provides BEMF signals as disturbance because BEMF is not included in the observer model.

To obtain the speed and position information from the position error, another algorithm 'A tracking observer' is used. This algorithm adopts the phase-locked loop mechanism. It requires a single input argument as phase error. Such phase tracking observer, with standard PI controller used as the loop compensator as shown in *Figure 14*.

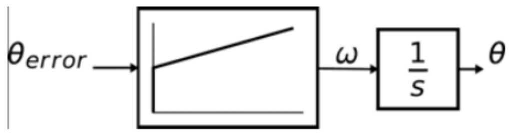


Figure 14. Block diagram of proposed PLL scheme for position estimation

4 Software implementation on the MC9S12ZVML128

4.1 MC9S12ZVML128 – Key modules for PMSM FOC control

The MC9S12ZVML128 device includes modules such as the Pulse Width Modulator with Fault Protection (PMF), a Programmable Trigger Unit (PTU), an Analogue-to-Digital Converter (ADC), and a Timer module (TIM) and a Gate Drive Unit (GDU) suitable for control applications, in particular, motor control applications. These modules are directly interconnected and can be configured to meet various motor control application requirements. *Figure 15* shows module interconnection for a typical PMSM Sensorless application using single shunt current sensing. The modules are described below and a detailed description can be found in the MC9S12ZVMRMV1, MC9S12ZVM-Family Reference Manual available at nxp.com.

4.1.1 Module interconnection

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

The PTU and ADC are based on list architecture; which means they operate using lists stored in memory. These lists define the trigger points for the PTU, commands for the ADC, and results from the ADC.

Each control loop cycle is started by a PMF reload event. The PMF reload event restarts the PTU time base. If the PTU is enabled, the reload is immediately passed on as a ptu_reload event to the ADC and GDU modules.

The PMF generates the reload event at the required PWM reload frequency. The PMF reload event causes the PTU time base to restart, to acquire the first trigger time from the list, and to generate the ptu_reload signal for the ADCx to start loading the ADC conversion command from the Command Sequence List (CSL).

When the trigger time is encountered, the corresponding PTU trigger generates the trigger_x signal for the associated ADC. For simultaneous sampling, the PTU generates two simultaneous trigger_x signals, one for each ADC. At the trigger_x signal assertion the ADC starts the first conversion of the next conversion sequence in the CSL (the first A/D command is already downloaded) (MC9S12ZVMRMV1, MC9S12ZVM-Family Reference Manual, available at nxp.com).

From the above mentioned, this implies that the PTU module serves as a delay block which can schedule several acquisitions of state variables relative to the start of the PWM period and within one PWM period.

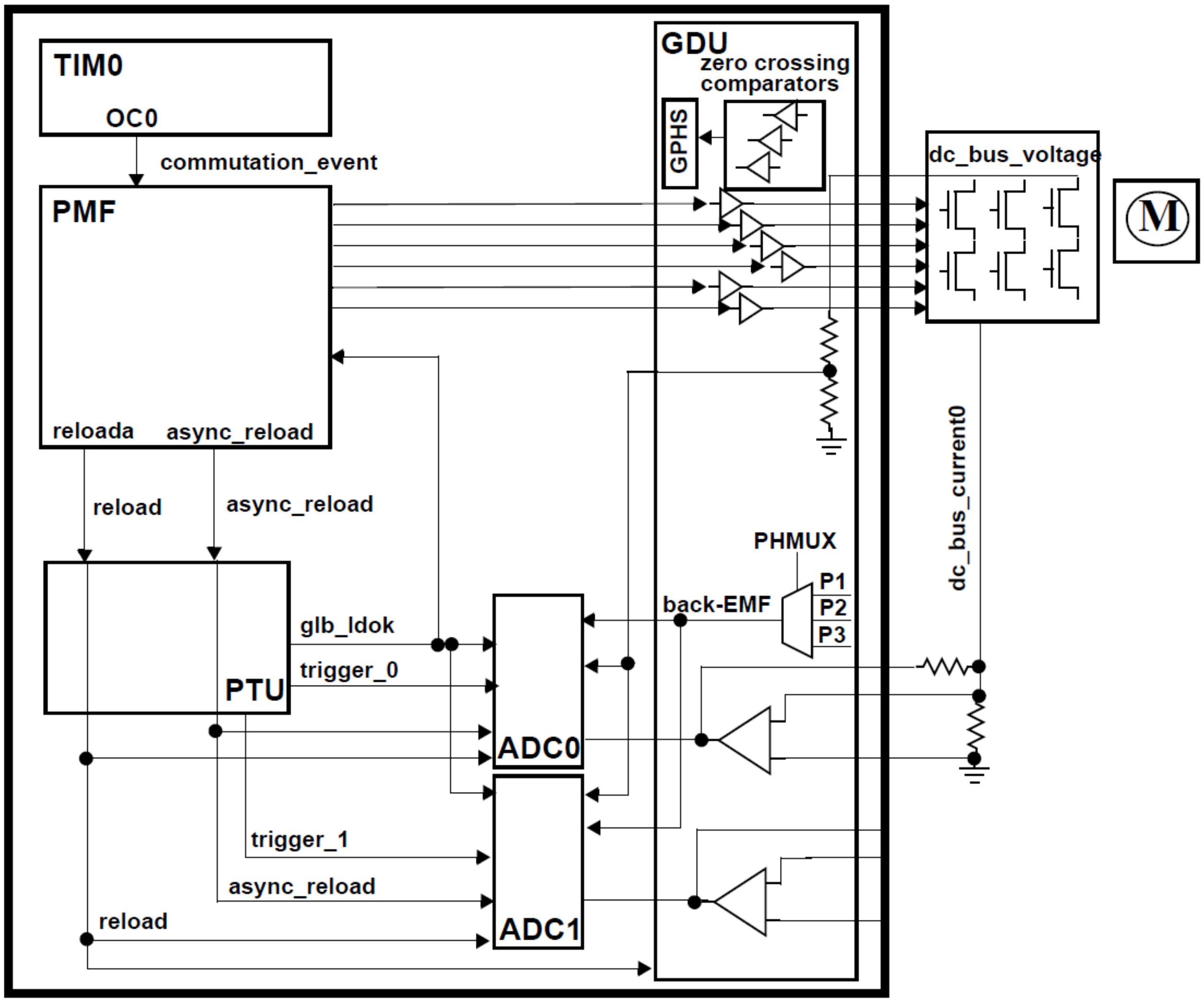


Figure 15. Module interconnections

4.1.2 Module involvement in digital PMSM Sensorless control loop

This section will discuss timing and modules synchronization to accomplish PMSM Sensorless Single-shunt FOC on the MC9S12ZVM128L and the internal hardware features.

The time diagram of the automatic synchronization between PWM and ADC in the PMSM application is shown in *Figure 16*.

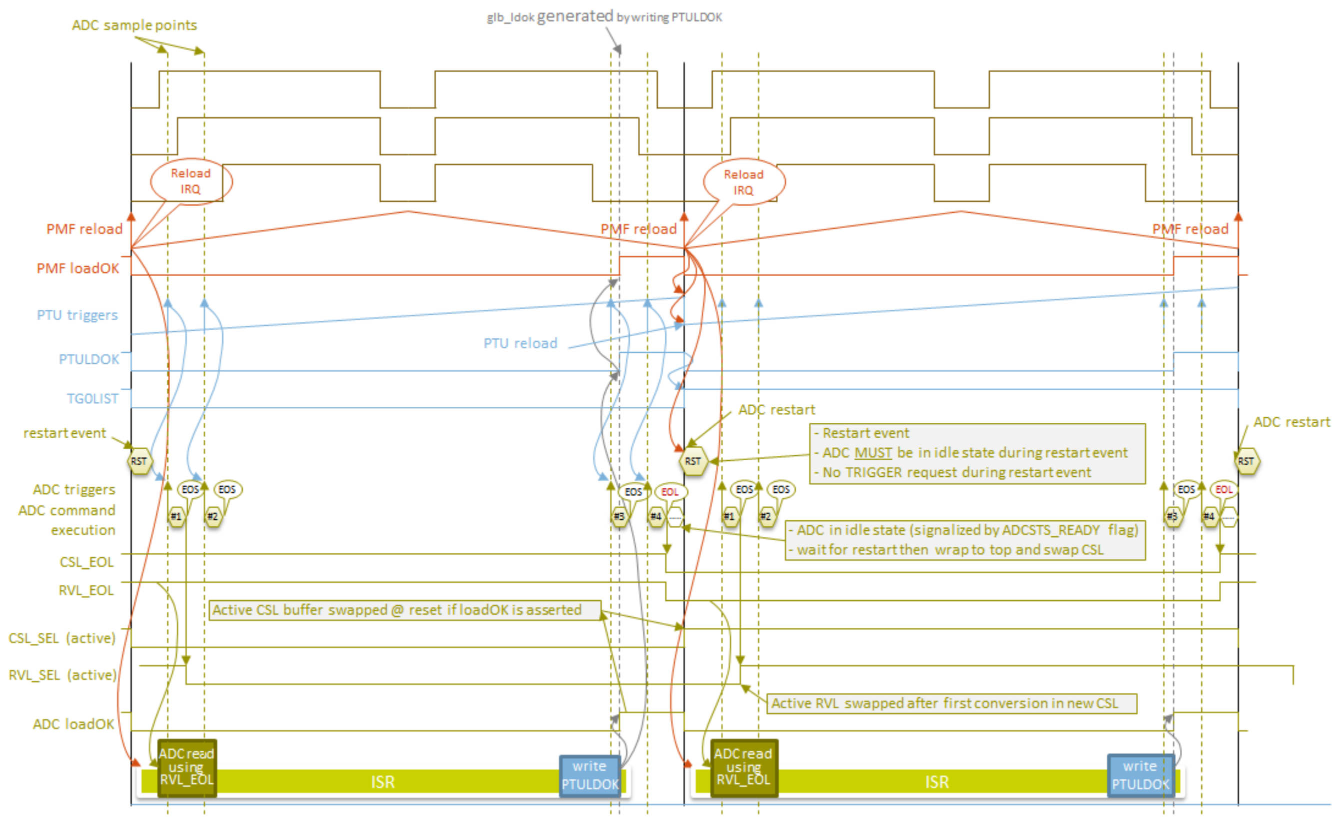

ADC sample points
glb_ldok generated by writing PTULDOK
Reload IRQ
PMF reload
PMF loadOK
PTU triggers
PTU reload
PTULDOK
TG0LIST
restart event
ADC restart
RST
EOS
EOL
#1
#2
#3
#4
ADC triggers
ADC command execution
CSL_EOL
RVL_EOL
CSL_SEL (active)
RVL_SEL (active)
ADC loadOK
- Restart event
- ADC MUST be in idle state during restart event
- No TRIGGER request during restart event
- ADC in idle state (signalized by ADCSTS_READY flag)
- wait for restart then wrap to top and swap CSL
Active CSL buffer swapped @ reset if loadOK is asserted
Active RVL swapped after first conversion in new CSL
ADC read using RVL_EOL
ISR
write PTULDOK

Figure 16. Time Diagram of PWM and ADC Synchronization

The PMSM Sensorless FOC control with single shunt current measurement is based on static timing; meaning the trigger point instances of the ADC conversions are located at same place within one control loop cycle. However, the trigger points vary from one control loop cycle to another, based on the PWM edges calculation.

Each control cycle starts with PWM reload signal (*Figure 16*). The reload signal is generated every PWM period. The PWM reload signal itself also triggers the reload of the Trigger List in the PTU module, as well as restarting the PTU counter. When the PTU counter reaches the predefined value in the trigger list (triggers #1, #2, #3 and #4), the PTU triggers ADC measurement. With each trigger, single measurement of a phase current in DC-link shunt resistor is triggered, getting two values of phase A and phase C current in this example. The ADC conversion results are automatically stored into a predefined queue in memory.

Another conversions are triggered with a fixed timing, dedicated to CPU Junction Temperature and DC-Bus Voltage measurement (not shown in *Figure 16*).

The CPU is triggered by the ADC conversion complete interrupt service routine. Based on the stored ADC values the current PI controllers calculate new PWM duty cycles, PWM edges for double switching and PTU triggers to set correct timings for the single shunt current sensing described above. These on/off edges will be updated to PMF module at next reload event.

The TIM, PTU, GDU and ADC peripherals are based on the bus clock. To achieve a higher resolution of PWM, the PMF module is supplied by a core clock. The core clock is double that of the bus clock.

4.2 MCS12ZVM Device initialization

4.2.1 CPMU

For proper operation the CPMU needs to have stable power supply. The power supply is stable when the GDUF_GLVLSF is cleared. The application is using Internal Reference which provides a 1MHz internal clock (CPMUOSC_OSCE = 1). Out of the 1MHz Internal clock the bus and core clock is derived by following settings:

```
        CPMUREFDIV_REFFRQ = 0;
        CPMUSYNR_SYNDIV = 49;
        CPMUSYNR_VCOFRQ = 3;
        CPMUPOSTDIV_POSTDIV = 0;
```

Set the bus and core clock to 50MHz and 100MHz respectively. The SW needs to wait until the PLL lock (CPMUIFLG_LOCK set to 1).

The CPMU module settings provide also possibility to enable the High Temperature Sensor which is routed than to ADC channel.

```
        CPMUHTCTL_VSEL = 0;
        CPMUHTCTL_HTE = 1;
```

4.2.2 PMF

The Pulse Width Modulator with Fault Protection (PMF) module is configured to generate a centre-aligned (PMFCFG0_EDGEx = 0) PWM with a frequency of 20 kHz (PMFMODA = 2500). In order to protect the MOSFET devices in the same leg of the inverter, deadtime is set to approximately 0.20 us (PMFDTMA = 20). PWM generator A runs as the master and generates the Reload Signal as a synchronization signal for the other submodules (PMFCFG2_REV[0:1] = 1). The reload signal is generated at every PWM opportunity (PMFFQCA = 1). Pair A, Pair B and Pair C PWMs are synchronized to PWM generator A (PMFCFG0_MTG = 0). PMF module generates an interrupt on every reload event (PMFENCA_PWMRIEA = 1) to perform application logic and calculate the PMSM FOC Sensorless algorithm.

A PWM pulse width PMFVAL registers are double buffered and are swapped when GLDOK is set and the PWM reload signal occurs. The GLDOK is an external signal generated by the PTU module. The GLDOK is enabled at the PWM module (PMFENCA_GLDOKA = 1).

Double switching feature enables to control two on/off edges of the PWM output signal symmetrically to the center of a PWM period. First ON edge of the phase A is given by PMFVAL0, followed by the first OFF edge based on PMFVAL1. Second ON edge is again given by PMFVAL0 and the second OFF edge is given by PMFVAL1. Phase B is controlled by PMFVAL2 and PMFVAL3; Phase C is controlled by PMFVAL4 and PMFVAL5 respectively. Deadtime correction is made for every switching of the transistors in the same leg of the inverter. Double switching is enabled at the PMF module (PMFICCTL_PECx = 0x7). Further information on the double switching can be found in MC9S12ZVMRMV1 available at www.nxp.com.

4.2.3 PTU

The Programmable Trigger Unit (PTU) is intended to completely avoid CPU involvement in the time acquisitions of state variables during the control cycle.

The PTU module consists of 2 trigger generators (TG). For each TG, a separate enable bit is available, so that both TGs can be enabled independently. Trigger generator 0 is connected to ADC0, and trigger generator 1 is connected to ADC1. The trigger generation of the PTU module is synchronized to the incoming reload event. This reload event resets and restarts the internal time base counter and makes sure that the first trigger value from the actual trigger list is loaded. Furthermore, the corresponding ADC is informed that a new control cycle has started.

If the counter value matches the current trigger value, then a trigger event is generated. In this way, the reload event is delayed by the number of bus clock cycles defined by the current trigger value. All acquisition time values are stored inside the global memory map, basically, inside the system memory as a two 2-dimensional arrays of integers (ptuTriggerList0[][] and ptuTriggerList1[][]). The exact location of the acquisition time values (ptuTriggerList0[][] and ptuTriggerList1[][]) in the system memory is given by the linker command file and linked to the PTU module during the initialization phase.

```
PTUPTR = ptuTriggerList0;
```

Each trigger generator uses only one list to load the trigger values from the memory. The pointers for the primary (TG0L0IDX/ TG1L0IDX) and alternate (TG0L1IDX/ TG1L1IDX) lists are equal.

```
TG0L1IDX = (unsigned char)(((long)&ptuTriggerList0[1][0] - (long)ptuTriggerList0) >> 1);
TG1L0IDX = (unsigned char)(((long)&ptuTriggerList1[0][0] - (long)ptuTriggerList0) >> 1);
TG1L1IDX = (unsigned char)(((long)&ptuTriggerList1[0][0] - (long)ptuTriggerList0) >> 1);
```

The trigger generator is using only one physical list of trigger events, even if the trigger generator logic is switching between both pointers. The PTU module generates the LDOK signal used to inform other modules that the double buffered registers were updated by software.

4.2.4 GDU

The Gate Drive Unit (GDU) is a Field Effect Transistor (FET) pre-driver designed for three-phase motor control applications. The following GDU features are used in PMSM FOC sensorless control

- **Charge Pump:** The charge pump is used to maintain the high-side driver gate source voltage VGS when PWM is running at a 100% duty cycle. The clock for the charge pump is set to be $\frac{f_{bus}}{64}$ (GDUCLK2_GCPCD = 4)
- **Desaturation Error:** The GDU integrates three desaturation comparators for the low-side FET pre-drivers and three desaturation comparators for the high-side FET pre-drivers. The desaturation level is set to be 1.35V (GDUDSLVL = 0x77) for both low-side and high-side FET. A blanking time during the FET transients needs to be employed. The blanking time is set to be approximately 8 us (GDUCTR = 0x13).
- **Current Sense Amplifiers:** Internal current sense amplifier 0 (GDUE_GCSE0 = 1) is used to measure DC-link current. The output of the current sense amplifier 0 is routed internally to ADC0 channel 0. Internal amplifier 1 is enabled as well (GDUE_GCSE1 = 1), but it is not used in the application. The output of the current sense amplifier 1 is routed internally to ADC1 channel 1.

4.2.5 ADC

The MC9S12ZVML128 uses two independent Analogue-to-Digital Converters (ADC). Both ADCs are n-channel multiplexed input successive approximation analogue-to-digital converters. The List Based Architecture (LBA) provides a flexible conversion sequence definition, as well as flexible oversampling. Both ADC conversion command lists are stored inside the global memory map, basically, inside the system memory as two dimensional arrays of bytes (ADC0CommandList[][], ADC1CommandList[][]). The exact location of the ADC conversion commands in the system memory is given by the linker command file and linked to the respective ADC module during the initialization phase. The same strategy is used for the ADC Results. The Conversion results are stored in an array of shorts (ADC0ResultList[][], ADC1ResultList[][]) located in system memory.

```
  // ADC0 Command Base Pointer
  ADC0CBP = ADC0CommandList;

  // ADC0 Result Base Pointer
  ADC0RBP = ADC0ResultList;


  // ADC1 Command Base Pointer
  ADC1CBP = ADC1CommandList;

  // ADC1 Result Base Pointer
  ADC1RBP = ADC1ResultList;
```

The ADC conversion clocks are set to be 8.33 MHz (ADC0TIM = 2; ADC1TIM = 2). The results are stored in memory as 12-bit (ADC0FMT_SRES = 4; ADC1FMT_SRES = 4) left-justified data (ADC0FMT_DJM = 0; ADC1FMT_DJM = 0).

Conversion flow of both ADCs is controlled by internal signals (generated by the PTU) and by the DataBus (ADC0CTL_0_ACC_CFG = 3; ADC1CTL_0_ACC_CFG = 3). The results are stored in system memory even if commutation occurs when conversion is ongoing (ADC0CTL_0_STR_SEQA = 1; ADC1CTL_0_STR_SEQA = 1).

The ADC1 schedules the end of list interrupt (ADC1CONIE_1_EOL_IE = 1).

The PMSM sensorless FOC algorithm uses ADC0 to measure the DC-link current at four time instants. The ADC1 is used to measure DC-Bus voltage and CPU junction temperature.

4.3 Software architecture

4.3.1 Introduction

This section describes the software design of the Sensorless PMSM Field Oriented Control framework application. First, the application overview and description of software implementation are provided. Then the numerical scaling in fixed-point fractional arithmetic of the controller is discussed. The aim of this chapter is to help in understanding of the designed software.

4.3.2 Application data flow overview

The application software is interrupt driven running in real time. There is one periodic interrupt service routine associated with the ADC end of sequence interrupt, executing all motor control tasks. These include both fast current and slow speed loop control. All tasks are performed in an order described by the application state machine shown in *Figure 19*, and application flowcharts shown in *Figure 17* and *0*.

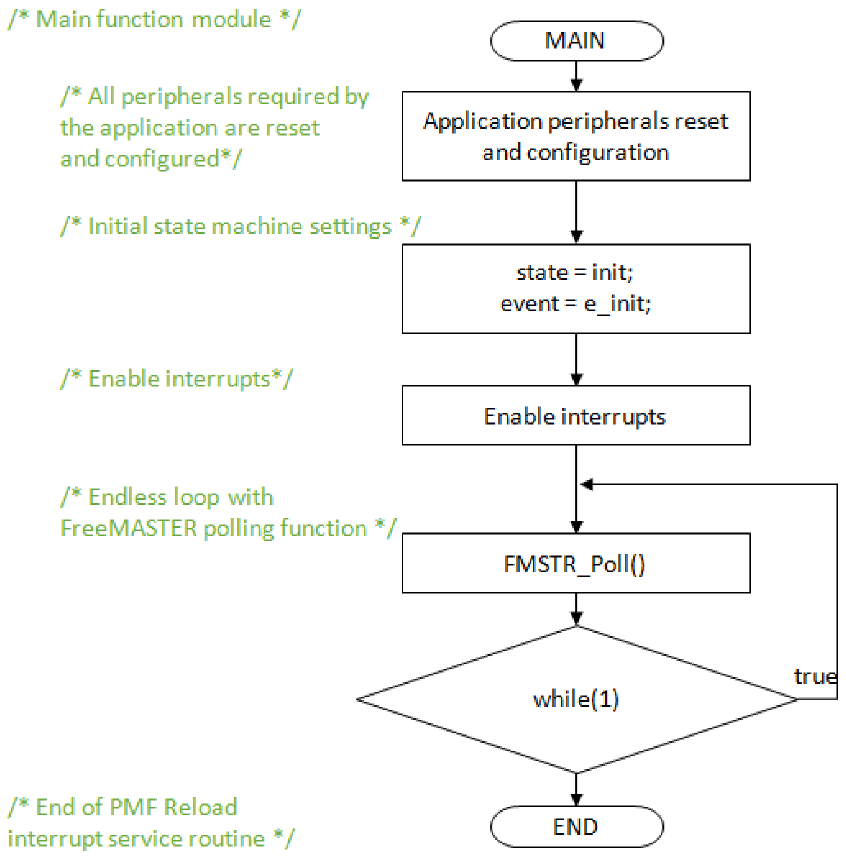

/* Main function module */
MAIN
/* All peripherals required by the application are reset and configured*/
Application peripherals reset and configuration
/* Initial state machine settings */
state = init;
event = e_init;
/* Enable interrupts*/
Enable interrupts
/* Endless loop with FreeMASTER polling function */
FMSTR_Poll()
while(1)
true
/* End of PMF Reload interrupt service routine */
END

Figure 17. Flow chart diagram of main function with background loop.

To achieve precise and deterministic sampling of analog quantities and to execute all necessary motor control calculations, the state machine functions are called within a periodic interrupt service routine. Hence in order to actually call state machine functions the periphery causing this periodic interrupt must be properly configured and the interrupt enabled. As is described in section *MCS12ZVM Device initialization*, all peripherals are initially configured and all interrupts are enabled after a RESET of the device. As soon as interrupts are enabled and all peripheries are correctly configured, the state machine functions are called from the ADC end of sequence interrupt service routine. The background loop handles non-critical timing tasks, such as the FreeMASTER communication polling.

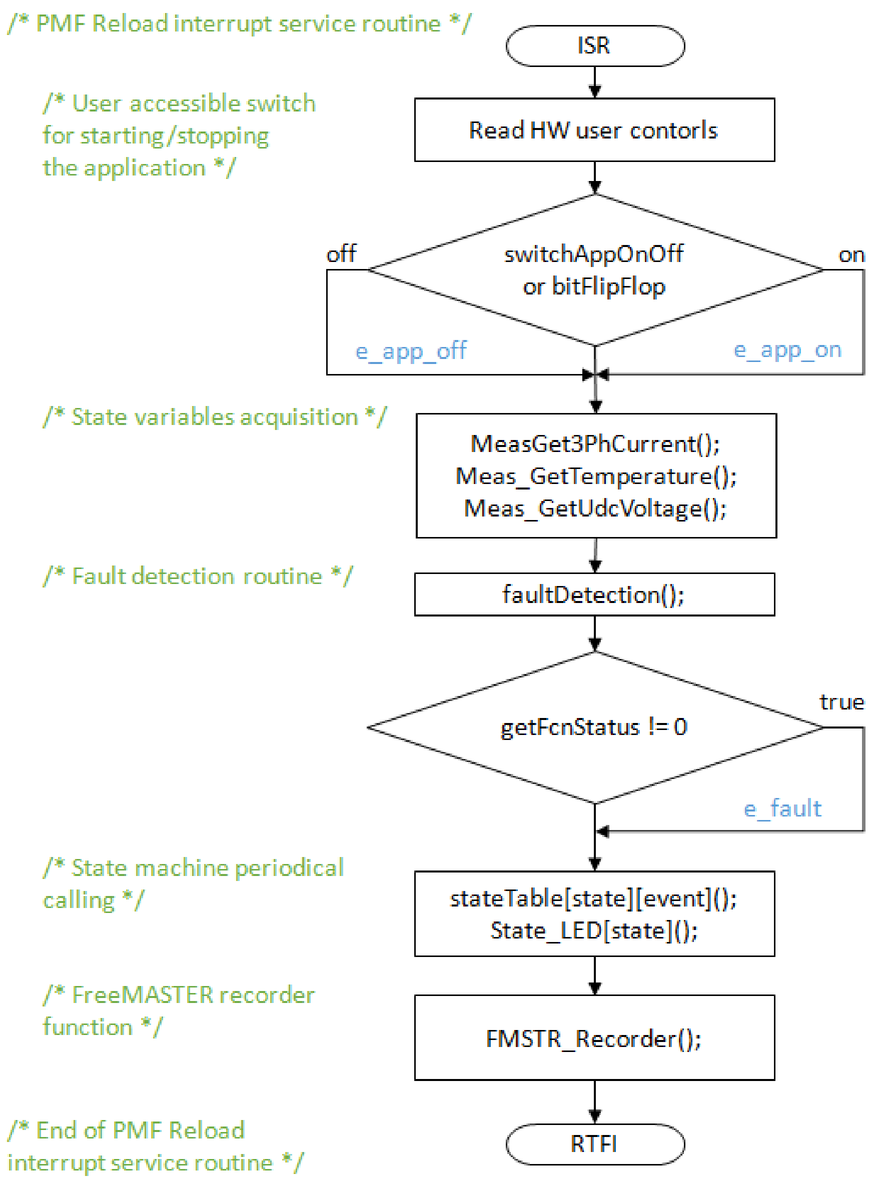

/* PMF Reload interrupt service routine */
ISR
/* User accessible switch for starting/stopping the application */
Read HW user contorls
switchAppOnOff or bitFlipFlop
off
on
e_app_off
e_app_on
/* State variables acquisition */
MeasGet3PhCurrent();
Meas_GetTemperature();
Meas_GetUdcVoltage();
/* Fault detection routine */
faultDetection();
getFcnStatus != 0
true
e_fault
/* State machine periodical calling */
stateTable[state][event]();
State_LED[state]();
/* FreeMASTER recorder function */
FMSTR_Recorder();
/* End of PMF Reload interrupt service routine */
RTFI

Figure 18. Flow chart diagram of periodic interrupt service routine

4.3.3 State machine

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

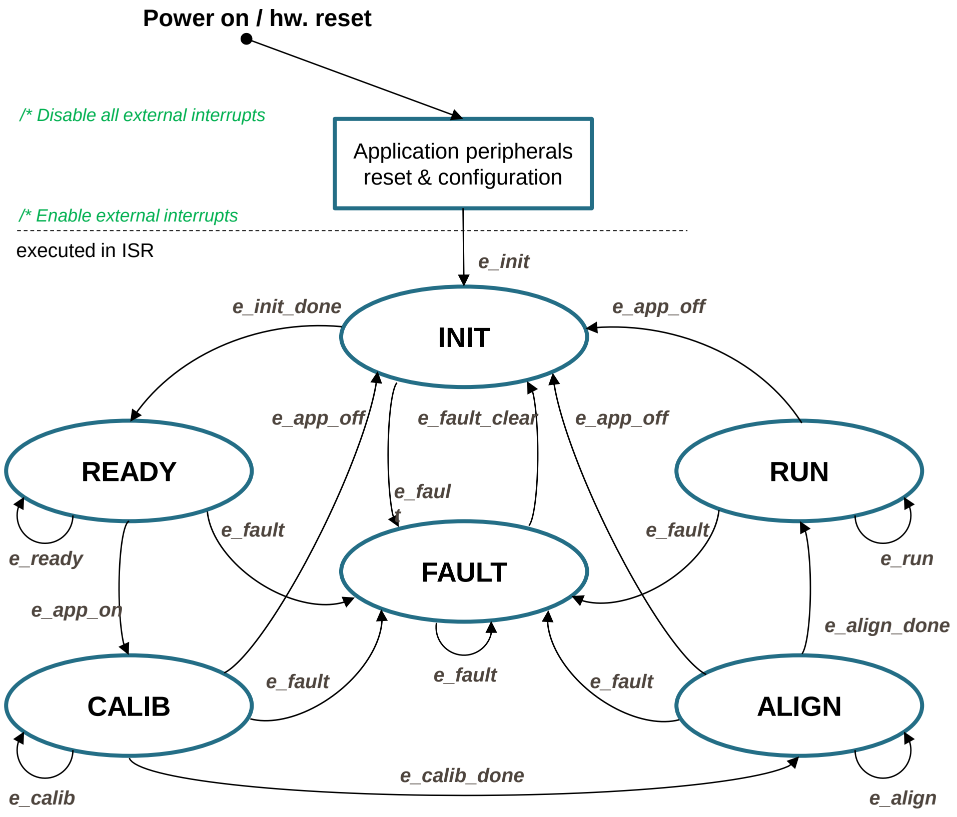


Figure 19. Application state machine

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

AppStates:

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

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

AppEvents:

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

4.3.3.1 State – FAULT

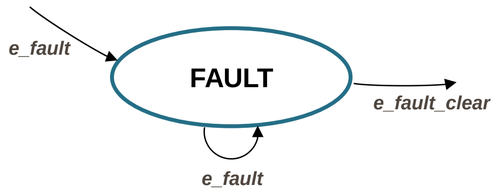


Figure 20. FAULT state with transitions

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 allows transition back to itself if a fault is present in the system and the user does not request clearing of fault flags. There are two different variables to signal fault occurrence in the application. The warning register tempFaults represents the current state of the fault pin/variable to warn the user that the system is getting close to its critical operation. And the fault register permFaults represents a fault flag, which is set and put the application immediately to fault state. Even if the actual fault is reset (fault source disappears), the fault remains set until manually cleared by the user. Such mechanisms allow for stopping the application and analyzing the cause of failure, even if the fault was caused by a short glitch on monitored pins/variables. State FAULT can only be left when application variable switchFaultClear is manually set to true (using FreeMASTER) or by simultaneously pressing the user buttons (SW1 and SW2) on the S12ZVM evaluation board. That is, the user has acknowledged that the fault source has been removed and the application can be restarted. When the user sets switchFaultClear = true; the following sequence is automatically executed:

```
        permFaults.mcu.R           = 0x0;      // Clear mcu faults
        permFaults.motor.R         = 0x0;      // Clear motor faults
        permFaults.stateMachine.R = 0x0;       // 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 event to cntrState.event = e_fault_clear while in FAULT state represents a new request to proceed to INIT state. This request is purely user action and does not depend on actual fault status. In other words, it is up to the user to decide when to set switchFaultClear true. However, according to the interrupt data flow diagram shown in Figure18, function faultDetection() is called before state machine function state_table[event][state](). Therefore, all faults will be checked again and if there is any fault condition remaining in the system, the respective bits in permFaults and tempFaults variables will be set. As a consequence of permFaults not equal to zero, function faultDetection() will modify the application event from e_fault_clear back to e_fault, which means jump to fall state when state machine function state_table[event][state]() is called. Hence, INIT state will not be entered even though the user tried to clear the fault flags using switchFaultClear. When the next state (INIT) is entered, all fault bits are cleared, which means no fault is detected (permFaults = 0x0) and application variable switchFaultClear is manually set to true.

The application is scanning for following system warnings and errors:

- Over voltage
- Under voltage
- Over phase current in each phase
- Over heating

The thresholds against the measured system variables are compared can be modified in INIT state. In addition, fault state is entered if following errors are detected:

- ADC Errors (Load OK, Restart Request, Trigger, End of List, Command Value, Illegal Access)
- GDU Errors (Desaturation, High V_{HD} Supply, Low V_{LS} Supply, Overcurrent)
- PMF Errors (Fault detected)
- PTU Errors (Memory Access, Reload, Timing)

4.3.3.2 State – INIT

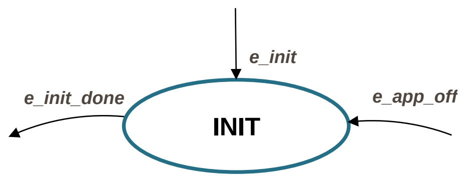


Figure 21. INIT state with transitions

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

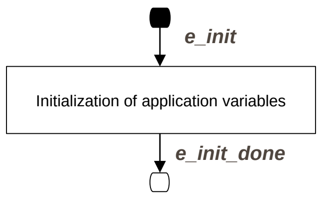


Figure 22. Flow chart of state INIT

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

4.3.3.3 State – READY

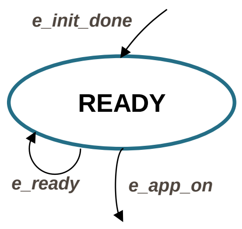


Figure 23. READY state with transitions

In READY state application is waiting for user command to start the motor. The application is released from waiting mode by releasing the on board user switch or by FreeMASTER interface setting the variable switchAppOnOff = true (see flow chart in *Figure 24*).

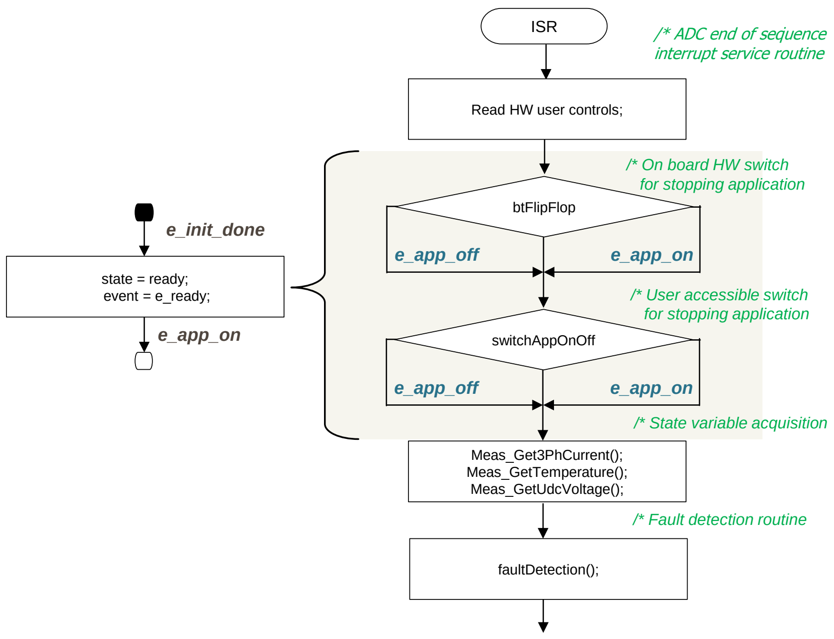


Figure 24. Flow chart of state READY

4.3.3.4 State – CALIB

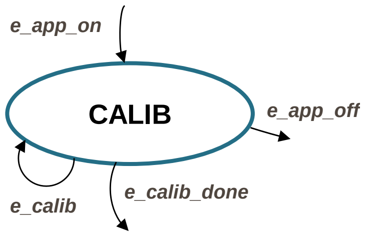


Figure 25. CALIB state with transitions

In this state, ADC DC offset calibration is performed. Once the state machine enters CALIB state, all PWM output are enabled. Calibration of the DC offset is achieved by disabling the PWM outputs, and taking several measurements of ADC channel connected to the current sensor. These measurements are then averaged, and the average value for the channel is stored. This value will be subtracted from the measured value when in normal operation. This way the half range DC offset, caused by voltage shift of 2.5V in conditional circuitry (see *Figure 5*), is removed in the measured phase. State CALIB is a state that allows transition back to itself, provided no faults are present, the user does not request stop of the application (by switchAppOnOff=true), and the calibration process has not finished. The number of samples for averaging is set by default to 2^10=1024, and can be modified in the state INIT. After all 1024 samples have been taken and the averaged values successfully saved, the application event is automatically set to cntrState.event=e_calib_done and state machine can proceed to state ALIGN (see flow chart in *Figure 26*).

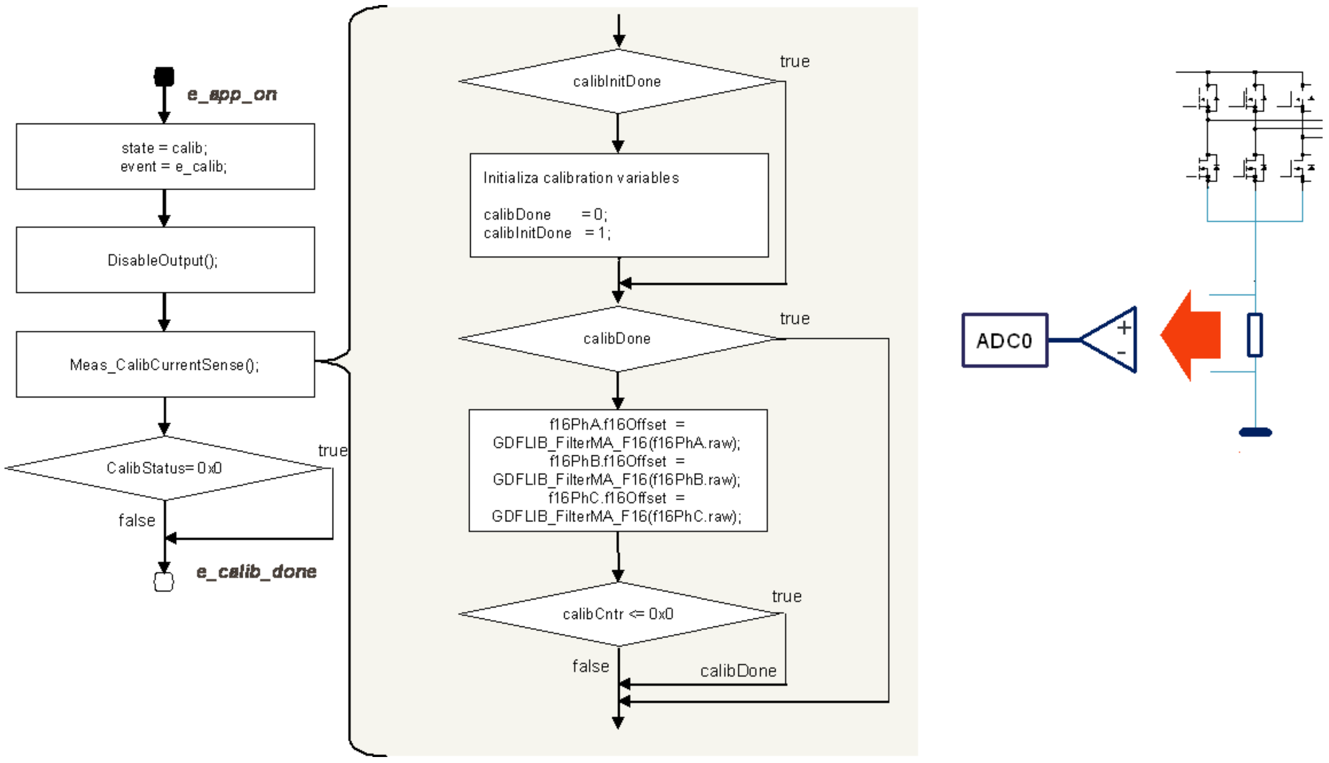

e_app_on
state = calib;
event = e_calib;
DisableOutput();
Meas_CalibCurrentSense();
CalibStatus=0x0
true
false
e_calib_done
calibInitDone
true
Initializa calibration variables
calibDone =0;
calibInitDone =1;
calibDone
true
f16PhA.f16Offset = GDFLIB_FilterMA_F16(f16PhA.raw);
f16PhB.f16Offset = GDFLIB_FilterMA_F16(f16PhB.raw);
f16PhC.f16Offset = GDFLIB_FilterMA_F16(f16PhC.raw);
calibCntr <= 0x0
true
false
calibDone
ADC0
+
-

Figure 26. Flow chart of state CALIB

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

4.3.3.5 State – ALIGN

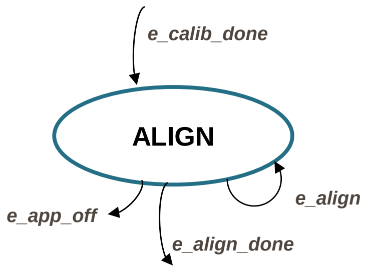


Figure 27. ALIGN state with transitions

This state shows alignment of the rotor and stator flux vectors to mark zero position. When using a model based approach for position estimation, the zero position is not known. The zero position is obtained at ALIGN state, where a DC voltage is applied in phase A for a certain period. This will cause the rotor to rotate to "align" position, where stator and rotor fluxes are aligned. The rotor position in which the rotor stabilizes after applying this DC voltage is set as zero position. In order to wait for rotor to stabilize in an aligned position, a certain time period is selected during which the DC voltage is constantly applied. The period of time and the amplitude of DC voltage can be modified in INIT state. Timing is implemented using a software counter that counts from a pre-defined value down to zero. During this time, the event remains set to cntrState.event=e_align. When the counter reaches zero, the counter is reset back to the pre-defined value, and event is automatically set to cntrState.event=e_align_done. This enables a transition to RUN state see flow chart in *Figure 28*.

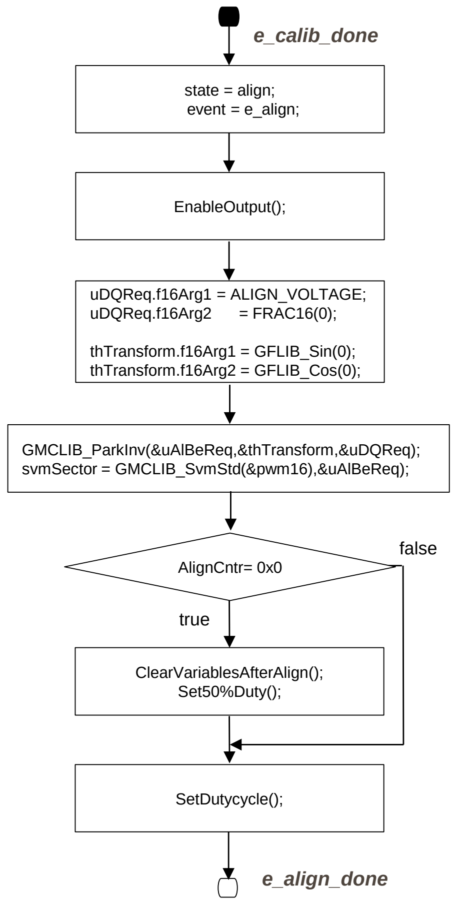

e_calib_done
state = align;
event = e_align;
EnableOutput();
uDQReq.f16Arg1 = ALIGN_VOLTAGE;
uDQReq.f16Arg2 = FRAC16(0);
thTransform.f16Arg1 = GFLIB_Sin(0);
thTransform.f16Arg2 = GFLIB_Cos(0);
GMCLIB_ParkInv(&uAlBeReq,&thTransform,&uDQReq);
svmSector = GMCLIB_SvmStd(&pwm16),&uAlBeReq);
AlignCntr= 0x0
false
true
ClearVariablesAfterAlign();
Set50%Duty();
SetDutycycle();
e_align_done

Figure 28. Flow chart of state ALIGN

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

4.3.3.6 State – RUN

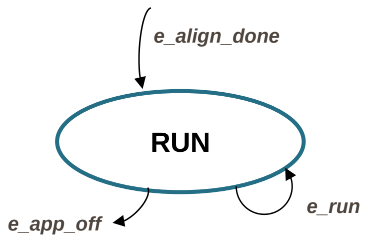


Figure 29. RUN state with transitions

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

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

1. **Force mode**: The FOC control is based on the generated position (so called open loop position), also this position is supplied to eBEMF observer in order to initialize its state.
2. **Tracking mode:** The FOC control is still using the open loop position, however, the eBEMF observer is left on its own, meaning that the observer is using its own estimated position and speed one calculation step delayed.
3. **Sensorless mode:** Both FOC control and eBEMF observer using estimated position.

The mode might be selected manually or automatically depending on the state of variable pos_mode. The pos_mode can be modified from FreeMASTER interface. For automatic mode the transition is based on the threshold value set in INIT state.

Calculation of fast current loop is executed every ADC end of sequence interrupt when in RUN state while calculation of slow speed loop is executed every Nth ADC end of sequence interrupt. Arbitration is done using a counter that counts from value N down to zero. When zero is reached, the counter is reset back to N and slow speed loop calculation is performed. This way, only one interrupt is needed for both loops and timing of both loops is synchronized. Slow loop calculations are finished before entering fast loop calculations (see flow chart in *Figure 30*).

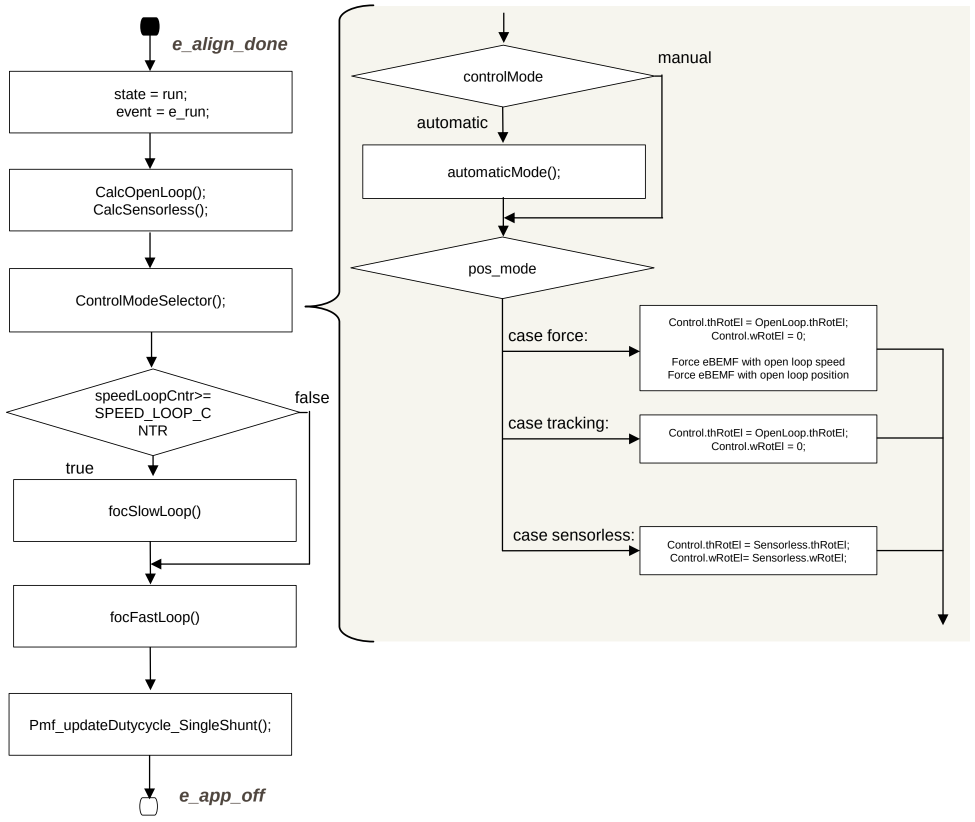

e_align_done
state = run;
event = e_run;
CalcOpenLoop();
CalcSensorless();
ControlModeSelector();
speedLoopCntr>= SPEED_LOOP_CNTR
false
true
focSlowLoop()
focFastLoop()
Pmf_updateDutycycle_SingleShunt();
e_app_off
controlMode
manual
automatic
automaticMode();
pos_mode
case force:
Control.thRotEl = OpenLoop.thRotEl;
Control.wRotEl = 0;
Force eBEMF with open loop speed
Force eBEMF with open loop position
case tracking:
Control.thRotEl = OpenLoop.thRotEl;
Control.wRotEl = 0;
case sensorless:
Control.thRotEl = Sensorless.thRotEl;
Control.wRotEl= Sensorless.wRotEl;

Figure 30. Flow chart of state RUN

Figure 31 shows implementation of FOC algorithm and the functions and variables used. As can be seen from the diagram, position/speed estimation is prepared for eBEMF observer.

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

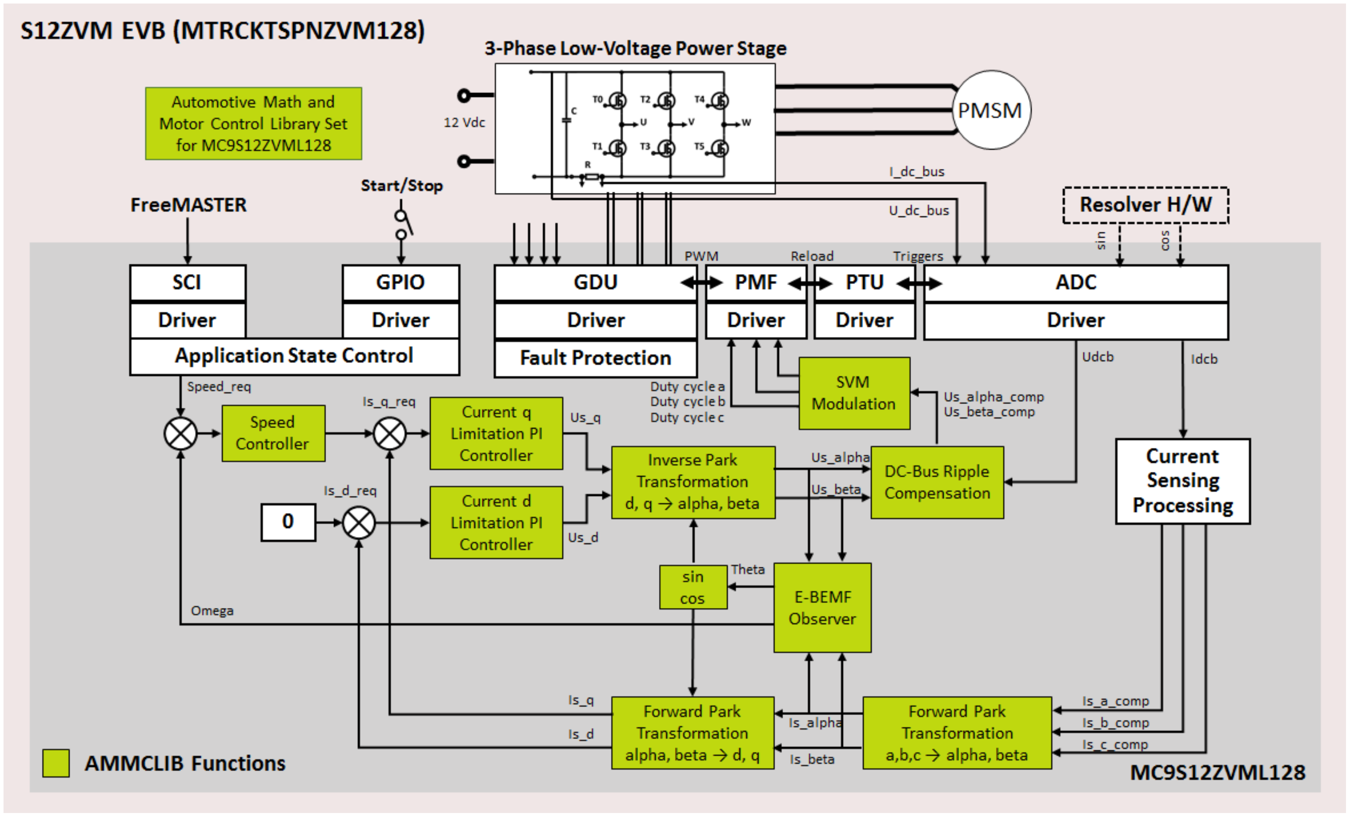


Figure 31. Sensorless FOC implementation

4.3.4 Scaling of Quantities

This application uses a fractional representation for most of the quantities. The fractional arithmetic is supported by several NXP devices as well as software.

The N-bit signed fractional format is represented using the 1.[N-1] format (1 sign bit, N-1 fractional bits). Signed fractional numbers (SF) lie in the following range:

$$-1.0 \leq SF \leq +1.0 - 2^{(-[N-1])}$$

Equation 14

For word and long-word signed fractions, the most negative number that can be represented is –1.0, whose internal representation is 0x8000 and 0x80000000, respectively. The most positive word is 0x7FFF or $1.0 - 2^{-15}$, and the most positive long-word is 0x7FFFFFFF or $1.0 - 2^{-31}$.

The following equation shows the relationship between a real and a fractional representation:

$$Fractional\ Value = \frac{Real\ Value}{Real\ Quantity\ Range}$$

Equation 15

where:

- Fractional Value = Fractional representation of quantities [–]
- Real Value = Real quantity in physical units [..]
- Real Quantity Range = Maximum defined quantity value used for scaling in physical units [..]

4.3.4.1 Voltage scale

Voltage scaling results from the sensing circuits of the hardware used; in case of the S12ZVM device the DC bus voltage is sensed directly on device HD pin. The voltage on HD pin is internally divided by 5 and routed to both ADC converters. The ADC converters can measure voltage level from zero to 5V. This is giving already the maximum measurable voltage of 25V on HD pin. For more details see the device reference manual available at nxp.com.

Voltage quantities are scaled to the maximum measurable voltage, in this case 25V. The relationship between real and fractional representations of voltage quantities is:

$$u_{Frac} = \frac{u_{Real}}{U\_DCB\_MAX}$$

Equation 16

where:

- u_{Frac} = Fractional representation of voltage [-]
- u_{Real} = Real voltage quantities in physical units [V]
- U_DCB_MAX = Defined voltage range maximum used for scaling in physical units [V]

In the application, the U_DCB_MAX value is the maximum measurable DCBus voltage, that is, U_DCB_MAX = 25 V. All application voltage variables are scaled in the same way.

4.3.4.2 Current Scaling

Current scaling also results from the sensing circuits of the hardware used. For more details see the device reference manual and board user manual available at nxp.com.

The relationship between real and fractional representation of current quantities is:

$$i_{Frac} = \frac{i_{Real}}{I\_MAX}$$

Equation 17

where:

- i_{Frac} = Fractional representation of current quantities [-]
- i_{Real} = Real current quantities in physical units [A]
- I_MAX = Defined current range maximum used for scaling in physical units[A]

In the application, the I_MAX value is the maximum measurable current:

I_MAX = 20 A. All application current variables are scaled in the same way.

4.3.4.3 Speed Scaling

Speed quantities are scaled to the defined speed range maximum. The speed range maximum is set higher than the maximum mechanical speed of the drive. The relationship between real and fractional representation of speed quantities is:

$$n_{Frac} = \frac{n_{Real}}{N\_MAX}$$

Equation 18

where:

- n_{Frac} = Fractional representation of speed quantities [-]
- n_{Real} = Real speed quantities in physical units [pm]
- N_MAX = Defined speed range maximum used for scaling in physical units [rpm]

In the application, the N_MAX value is defined as: N_MAX = 10000 mechanical rpm.

Electrical angular speed is scaled respectively:

$$\omega_{Frac} = \frac{\omega_{Real}}{WEL\_MAX}$$

Equation 19

where:

- ω_{Frac} = Fractional representation of speed quantities [-]
- ω_{Real} = Real speed quantities in physical units [radps]
- WEL_MAX = Defined speed range maximum used for scaling in physical units [radps]

In the application, the WEL_MAX value is defined as: WEL_MAX = 4188.79 electrical radps.

4.3.4.4 Position Scaling

The angles, such as rotor position, are represented as 16-bit signed fractional values in the range <–1, 1), which corresponds to the angle in the range <–π, π). In a 16-bit signed integer value, the angle is represented as follows:

$$-\pi = 0\text{x}8000$$
$$\pi = 0\text{x}7\text{FFF}$$

4.3.5 AMMC Library

The application source code uses the Freescale Automotive Math and Motor Control Library for the MC9S12ZVML128 microcontrollers (MC9S12ZVMMCLUG , Math and Motor Control Library for MC9S12ZVML128 User Manual, available at nxp.com). The library contains four independent library blocks: MLIB, GFLIB, GDFLIB and GMCLIB. MLIB includes basic mathematical operations such as summation, multiplication, and so on. GFLIB includes basic mathematical functions (such as sine, cosine, ramp, and so on). Advanced filter functions are part of the General Digital Filters Library, and standard motor control algorithms are part of the General Motor Control Library.

4.3.6 MCAT Integration

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

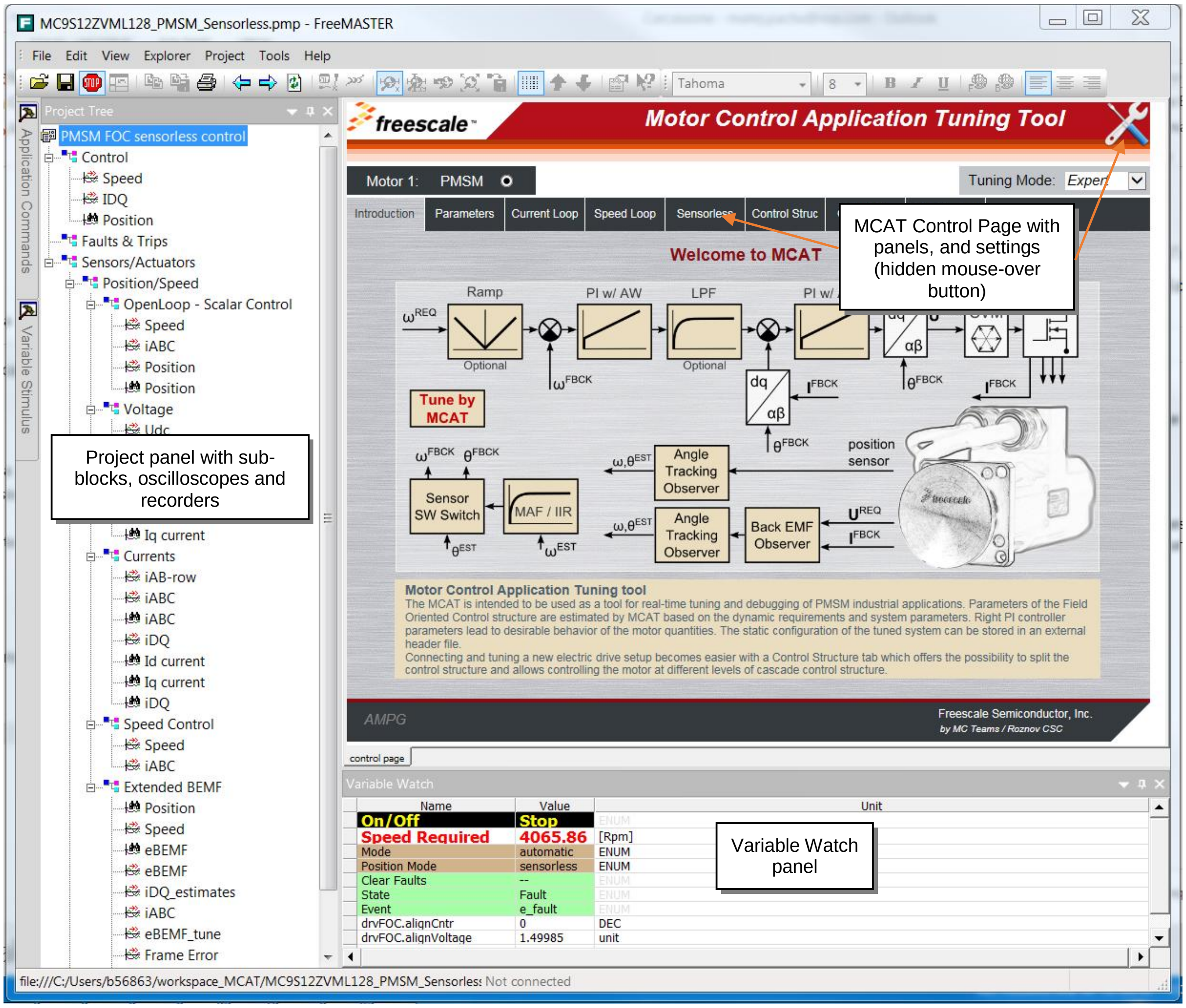
The MCAT tool runs under FreeMASTER online monitor, which allows the real-time tuning of the motor control application. Respecting the parameters of the controlled drive, the correct values of control structure parameters are calculated, which can be directly updated to the application or stored in an application static configuration file. The electrical subsystems are modeled using physical laws and the parameters estimation algorithms are based on 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}\FreeMASTER_control\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[5] debugging tool is used to control the application and monitor variables during run time. Communication with the host PC passes via USB. However, because FreeMASTER supports RS232 communication, there must be a driver for the physical USB interface, CP2102, installed on the host PC that creates a virtual COM port from the USB. The driver can be installed from www.silabs.com. The application configures the SCI module of the MC9S12ZVML128 for a communication speed of 19200bps. Therefore, the FreeMASTER user interface also needs to be configured respectively.



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

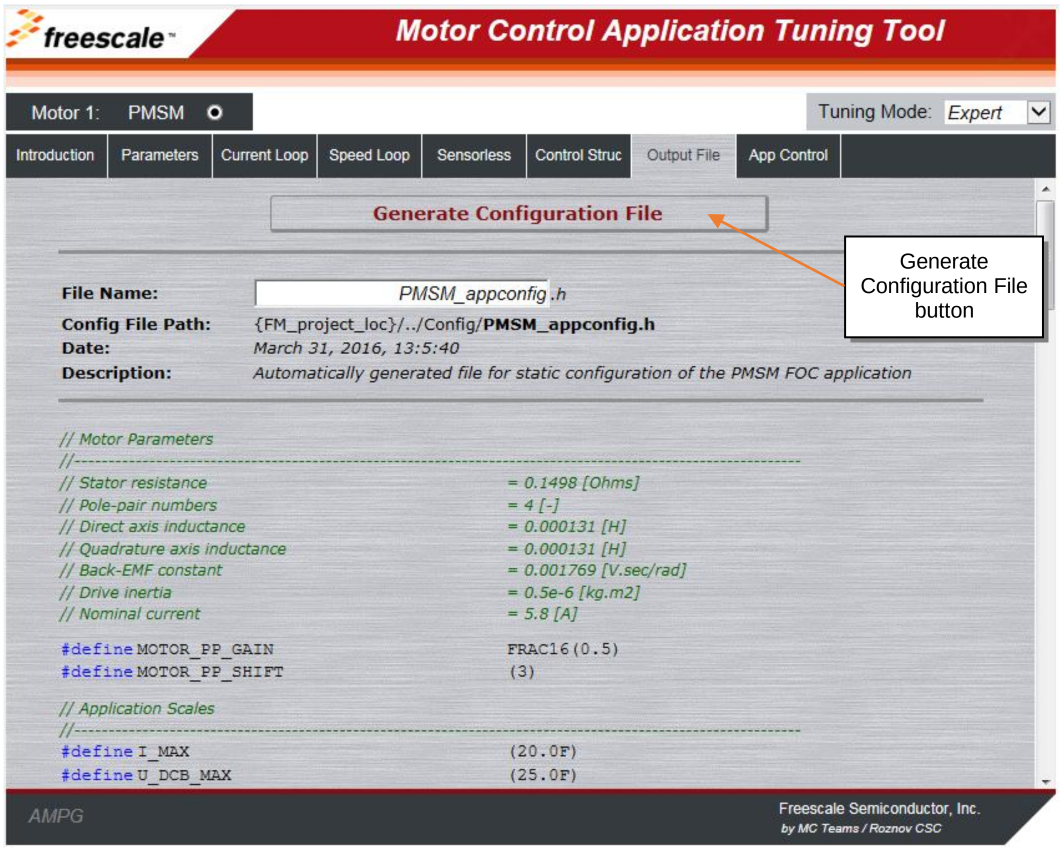
Name	Value	Unit
On/Off	Stop	ENUM
Speed Required	4065.86	[Rpm]
Mode	automatic	ENUM
Position Mode	sensorless	ENUM
Clear Faults	--	ENUM
State	Fault	ENUM
Event	e_fault	ENUM
drvFOC.alignCntr	0	DEC
drvFOC.alignVoltage	1.49985	unit

Figure 32. FreeMASTER and Motor Control Application Tunning Tool

5.1 MCAT Settings and Tuning

5.1.1 Application configuration and tuning

FreeMASTER and MCAT interface (*Figure 32*) enables online application tuning and control. The MCAT tuning shall be used before the very first run of the drive to generate the configuration header file (PMSM_appconfig.h). Most of the variables are accessible via MCAT online tuning (thus can be updated anytime), but some of them (especially the fault limit thresholds) must be set using the configuration header file generation, which can be done on the "Output File" panel by clicking the "Generate Configuration File" (see *Figure 33*).



File Name: PMSM_appconfig.h
Config File Path: {FM_project_loc}/../Config/**PMSM_appconfig.h**
Date: *March 31, 2016, 13:5:40*
Description: *Automatically generated file for static configuration of the PMSM FOC application*

```
// Motor Parameters
//----------------------------------------------------------------------
// Stator resistance                        = 0.1498 [Ohms]
// Pole-pair numbers                        = 4 [-]
// Direct axis inductance                   = 0.000131 [H]
// Quadrature axis inductance               = 0.000131 [H]
// Back-EMF constant                        = 0.001769 [V.sec/rad]
// Drive inertia                            = 0.5e-6 [kg.m2]
// Nominal current                          = 5.8 [A]

#define MOTOR_PP_GAIN                       FRAC16(0.5)
#define MOTOR_PP_SHIFT                      (3)

// Application Scales
//----------------------------------------------------------------------
#define I_MAX                               (20.0F)
#define U_DCB_MAX                           (25.0F)
```

Figure 33. Output File panel and “Generate Configuration File” button

Parameters runtime update is done using the “Update Target” button (see *Figure 34*). Changes can be also saved using “Store Data” button, or reloaded to previously saved configuration using “Reload Data” button.

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

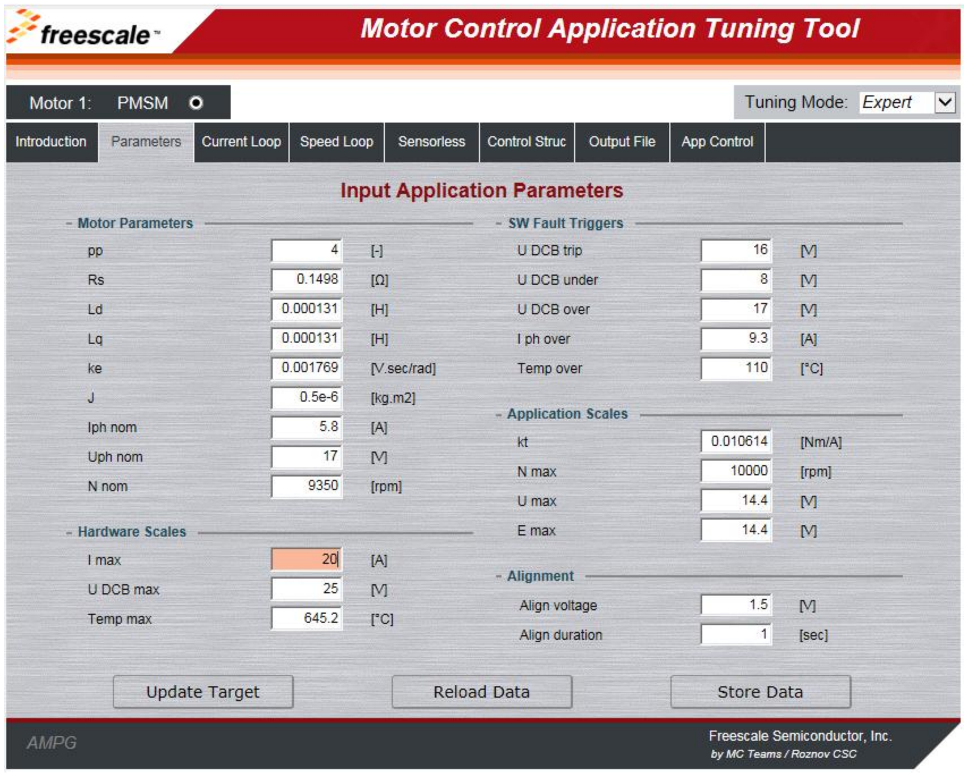
MCAT tool can be configured using hidden mouse-over “Settings” button (see *Figure 32*), where a set of advanced settings, for example PI controller types, speed sensors and other blocks of the control structure can be changed. However, it is not recommended to change these settings since it will force the MCAT to look for a different variables names and to generate different set of constants than the application is designed for. See MCAT tool documentation available at nxp.com.

The application tuning is provided by a set of MCAT pages dedicated to every part of the control structure. An example of the Application Parameters Tuning page is on *Figure 34*. Following list of settings pages is based on the PMSM sensorless application

- Parameters
  - Motor Parameters
  - Hardware Scales
  - SW Fault Triggers
  - Application Scales
  - Alignment
- Current Loop
  - Loop Parameters
  - D axis PI Controller
  - Q axis PI Controller
  - Current Controller Limits
- Speed Loop
  - Loop Parameters
  - Speed PI Controller
  - Speed Ramp
  - Speed Ramp Constants
  - Actual Speed Filter
  - Speed PI Controller Limits
- Sensorless
  - BEMF Observer Parameters
  - BEMF DQ Observer Constants
  - BEMF DQ Observer PI Constants
  - Tracking Observer Parameters
  - Tracking Observer Constants
  - Open Loop Start-up Parameters

Changes can be tested using MCAT “Control Struc” page (*Figure 35*), where the following control structures can be enabled:

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



Tuning Mode: Expert

Input Application Parameters

Motor Parameters

pp	4	[-]
Rs	0.1498	[Ω]
Ld	0.000131	[H]
Lq	0.000131	[H]
ke	0.001769	[V.sec/rad]
J	0.5e-6	[kg.m2]
Iph nom	5.8	[A]
Uph nom	17	[V]
N nom	9350	[rpm]

Hardware Scales

I max	20	[A]
U DCB max	25	[V]
Temp max	645.2	[°C]

SW Fault Triggers

U DCB trip	16	[V]
U DCB under	8	[V]
U DCB over	17	[V]
I ph over	9.3	[A]
Temp over	110	[°C]

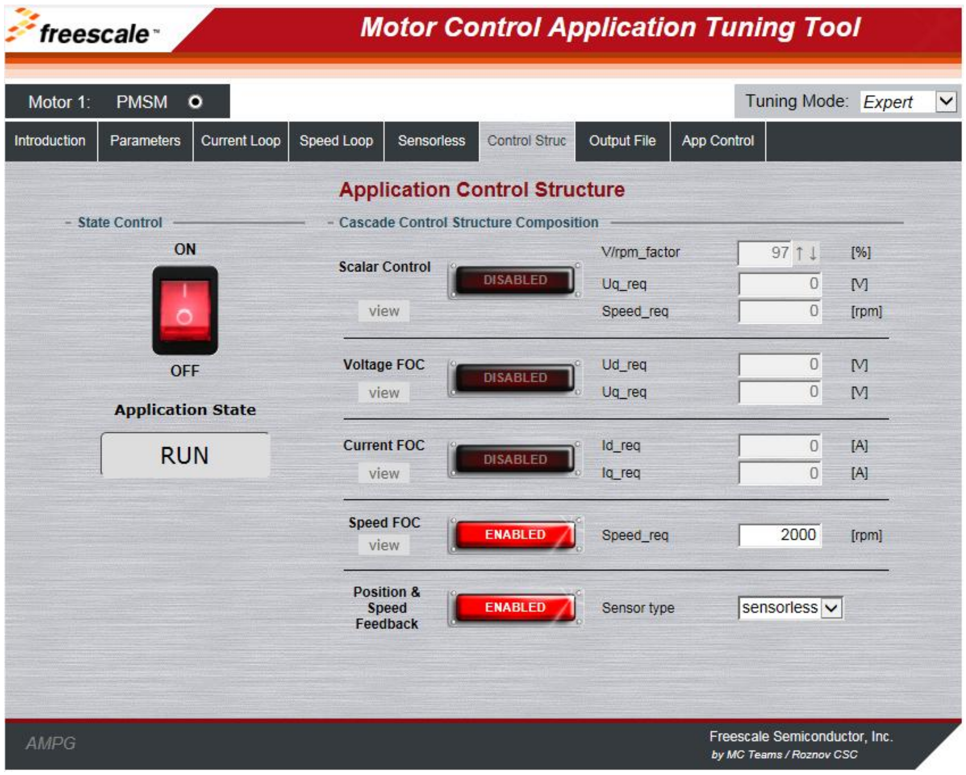
Application Scales

kt	0.010614	[Nm/A]
N max	10000	[rpm]
U max	14.4	[V]
E max	14.4	[V]

Alignment

Align voltage	1.5	[V]
Align duration	1	[sec]

Figure 34. MCAT Input Application Parameters Page



Tuning Mode: Expert

Application Control Structure

State Control

Application State: RUN

Cascade Control Structure Composition

Scalar Control	DISABLED	V/rpm_factor	97	[%]
		Uq_req	0	[V]
		Speed_req	0	[rpm]
Voltage FOC	DISABLED	Ud_req	0	[V]
		Uq_req	0	[V]
Current FOC	DISABLED	Id_req	0	[A]
		Iq_req	0	[A]
Speed FOC	ENABLED	Speed_req	2000	[rpm]
Position & Speed Feedback	ENABLED	Sensor type	sensorless	

Figure 35. MCAT Application Control Structure page

5.2 MCAT Application Control

All application state machine variables can be seen on the FreeMASTER MCAT control page as shown in *Figure 36*. Warnings and faults are signaled by a highlighted red color bar with name of the fault source. The warnings are signaled by a round LED-like indicator, which is placed next to the bar with the name of the fault source. The status of any fault is signaled by highlighting respective indicators. In the previous figure, for example, there is pending fault flag and one warning indicated ("Udcb HI" - DC bus voltage is close to its over voltage conditions). That means that the measured voltage on the DC bus exceeds the limit set in the MCAT_Init function. The warning indicator is still on if the voltage is higher than the warning limit set in INIT state. In this case, the application state FAULT is selected, which is shown by a frame indicator hovering above FAULT state. After all actual fault sources have been removed, no warning indicators are highlighted, but the fault indicators will remain highlighted. The pending faults can now be cleared by pressing the "FAULT" button. This will clear all pending faults and will enable transition of the state machine into INIT and then READY state. After the application faults have been cleared and the application is in READY state, all variables should be set to their default values. The application can be started by selecting APP_ON on application On/Off switch. Successful selection is indicated by highlighting the ON/OFF button in green.

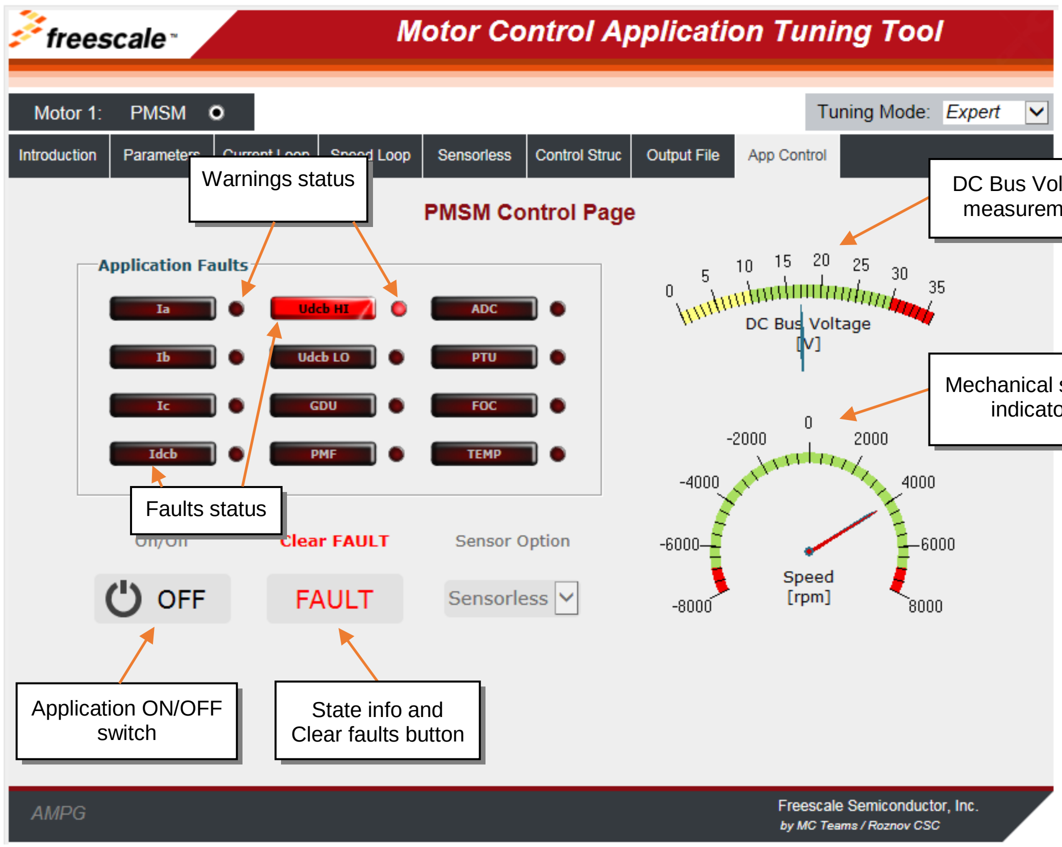


Figure 36. FreeMASTER MCAT Control Page for controlling the application

6 Conclusion

The design described shows the simplicity and efficiency in using the MC9S12ZVML128 microcontroller for Sensorless PMSM motor control using single shunt 3-phase current reconstruction, and introduces it as an appropriate candidate for various low-cost applications in the automotive area. MCAT tool provides interactive online tool which makes the PMSM drive application tuning friendly and intuitive.

7 References

- *MTRCKTSPNZVM128*, 3-phase Sensorless PMSM Motor Control Kit with the MagniV MC9S12ZVM available at www.nxp.com/AutoMCDevKits
- *FreeMASTER Run-Time Debugging Tool* available at www.nxp.com/FREEMASTER
- *MC9S12ZVMMCLUG* , Math and Motor Control Library for MC9S12ZVML128 User Manula available at www.nxp.com/AutoMCLib
- *MC9S12ZVMRMV1*, MC9S12ZVM-Family Reference Manual available at www.nxp.com
- MC9S12ZVML128 Evaluation Board User Manual available at www.nxp.com/AutoMCDevKits
- *MTRCKTSPNZVM128QSG* , Quick Start Guide for 3-Phase Sensorless PMSM Kit with MagniV MC9S12ZVML128 MCU, available at www.nxp.com/AutoMCDevKits
- Rashid, M. H. Power Electronics Handbook, 2nd Edition. Academic Press
- Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM available at www.nxp.com/mcat

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: AN5327
Rev. 0
08/2016