Freescale Semiconductor, Inc.
Design Reference Manual

Document Number: DRM144
Rev. 1, 03/2016

3-Phase BLDC Sensorless Motor Control Application

1 Introduction

The Brushless DC (BLDC) motor is the right choice for applications that require high reliability, high efficiency, and high power-to-volume ratio. The BLDC motor is considered to be a high performance motor, capable of providing large amounts of torque over a vast speed range. BLDC motors are a derivative of the most commonly used DC motor, the brushed DC motor, and share the same torque and speed performance curve characteristics. The major difference between the two is the use of brushes. BLDC motors do not have brushes (hence the name "brushless DC") and must be electronically commutated.

The primary advantages of the BLDC motor:

- High-speed operation – A BLDC motor can operate at speeds above 10,000 rpm under loaded and unloaded conditions.
- Responsiveness and quick acceleration – Inner rotor Brushless DC motors have low rotor inertia, allowing them to accelerate, decelerate, and reverse direction quickly.
- High-power density – BLDC motors have the highest running torque per cubic inch of any DC motor.

Contents

1 Introduction 1
2 Control theory basics 2
2.1 Brushless DC motor (BLDC motor) 2
2.2 Six step BLDC control 3
2.3 Digital control of BLDC motor 6
2.4 Sensorless BLDC motor control 7
3 Software Design 11
3.1 Data types 11
3.2 Scaling of analog quantities 12
3.3 Application overview 13
3.4 BLDC motor control 14
3.5 Motor control algorithms synchronization 16
3.6 Algorithms call 17
3.7 Application state machine function 19
3.8 Motor state machine 23
3.9 Sensorless BLDC control 28
3.10 Peripherals control 37
Acronyms and abbreviations 37
References 38
Revision history 38

- High reliability – BLDC motors do not have brushes, meaning they are more reliable and have life expectancies of over 10,000 hours. This results in fewer instances of replacement or repair, and less overall down time for your project.

The application software provided uses the concept of an isolated motor control algorithm software and hardware. This software approach enables easy porting of an application across other devices or hardware platforms.

This design reference manual describes the hardware-independent BLDC motor control algorithm part of code only. It does not cover individual implementation to the respective device. This design reference manual is supplemented by application notes describing implementation of the hardware-dependent part of code to the concrete devices.

The concept of the BLDC motor control algorithm is a speed closed-loop motor control algorithm using a back-EMF voltage integration method for a sensorless motor control. It serves as an example of a BLDC motor control design, and is focused on a simple and easy to understand control approach to BLDC.

This reference manual includes a basic BLDC motor theory, system concept, and software design description.

2 Control theory basics

2.1 Brushless DC motor (BLDC motor)

The brushless DC (BLDC) motor is a rotating electric machine where the stator is a classical 3-phase stator, similar to an induction motor, and the rotor has surface-mounted permanent magnets. The motor can have more than one pole pair per phase. See the figure below, which shows two pole pair motor - 2 pole pairs per phase. The number of pole pairs per phase defines the ratio between the electrical revolution and the mechanical revolution.

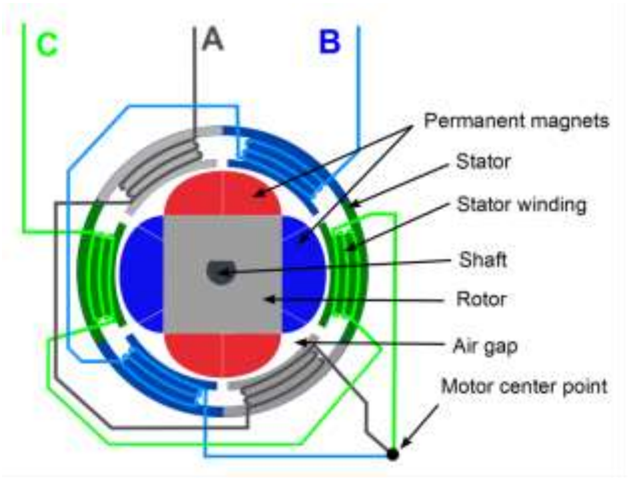


Figure 1. BLDC motor – cross section

2.2 Six step BLDC control

The BLDC motor is also referred to as an electronically commutated motor. There are no brushes on the rotor, and commutation is performed electronically at certain positions. The stator magnetic circuit is usually made of magnetic steel sheets. Magnetization of the permanent magnets and their displacement on the rotor are chosen in such a way that the back EMF (the voltage induced into the stator winding due to rotor movement) shape is trapezoidal. This allows a rectangular shaped 3-phase voltage system (see figure 4) to be used to create a rotational field (see figure 2) with low torque ripples (see figure 3).

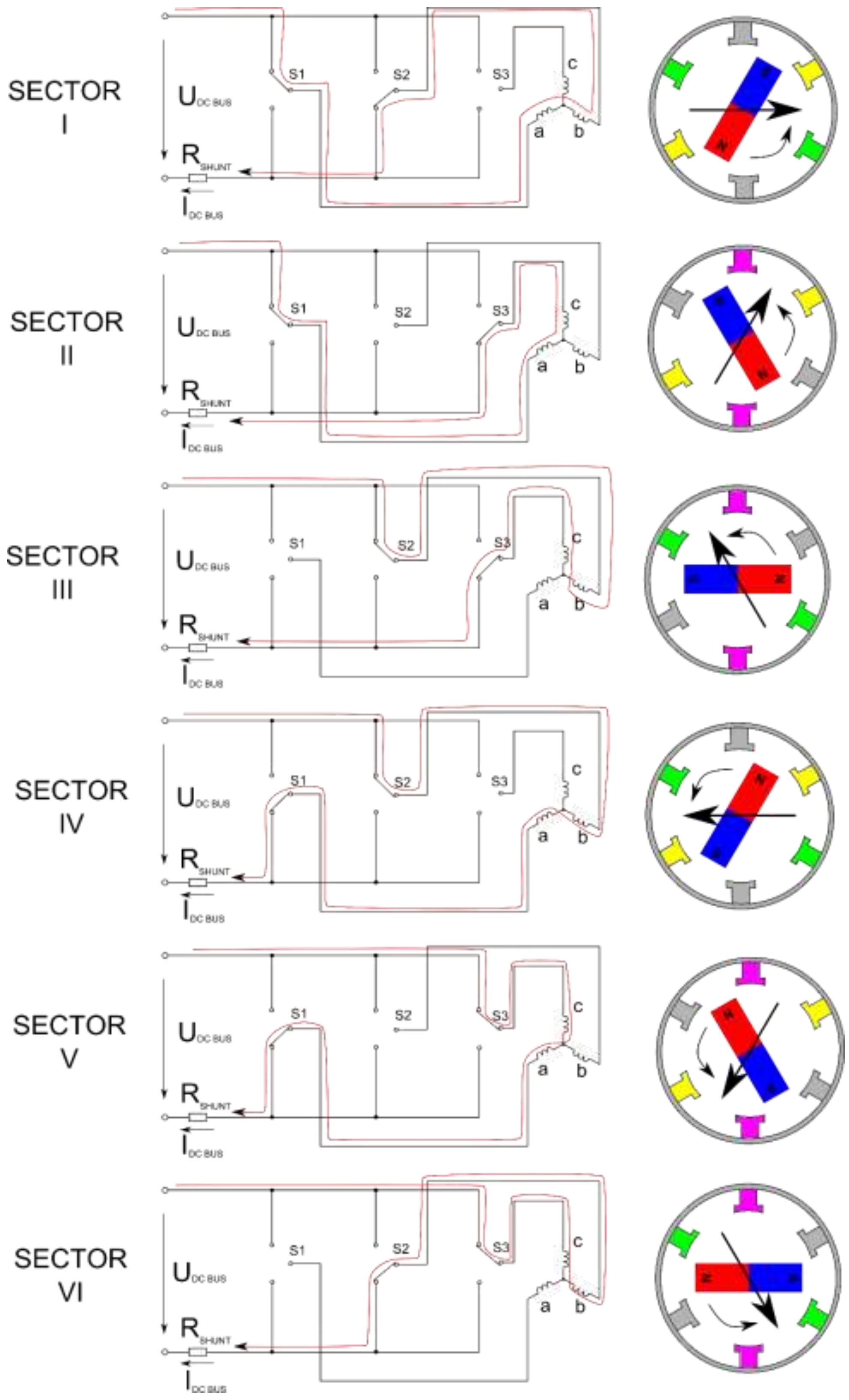


Figure 2. BLDC motor - six-step commutation technique

The BLDC motor rotation is controlled by a six-step commutation technique (sometimes called 60, 120 degree control). The six-step technique creates the voltage system with six vectors over one electronic rotation, as shown in the above figure.

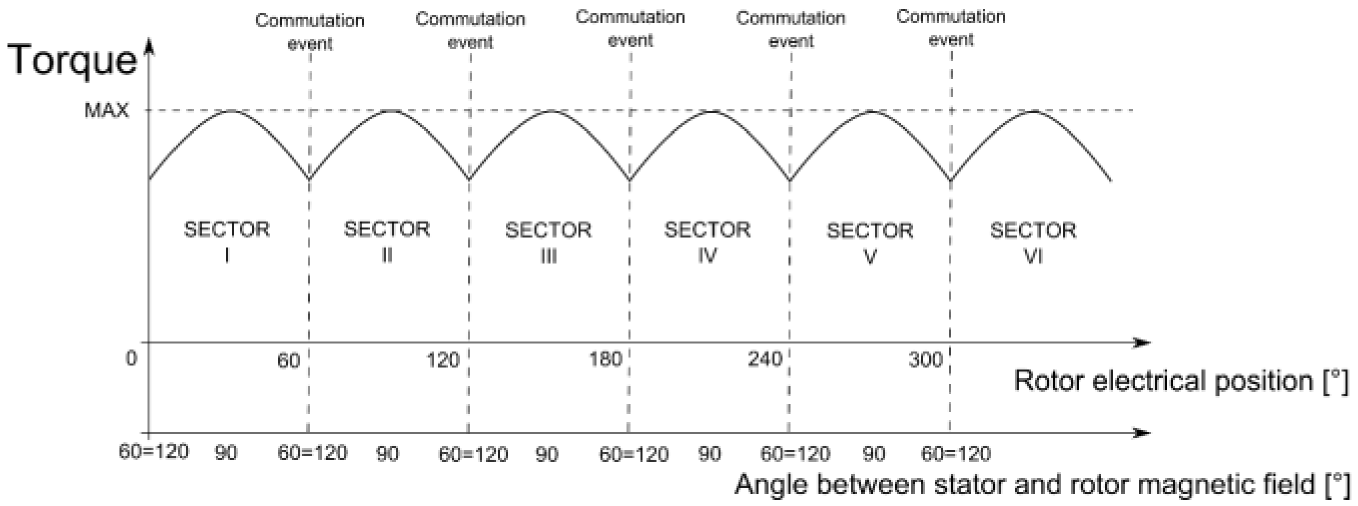


Figure 3. BLDC motor six-step commutation technique – torque ripples

The BLDC motor controller must control the applied voltage amplitude and synchronize the six-step commutation with the rotor position. The rotor position must be known at certain angles in order to synchronize the applied voltage with the back EMF. Under this condition, the BLDC motor is controlled with minimal torque ripple and maximum power efficiency.

2.3 Digital control of the BLDC motor

The BLDC motor is driven by rectangular voltage strokes coupled with the given rotor position (see the figure below). The generated stator flux interacts with the rotor flux, which is generated by a rotor magnet, defines the torque and thus the speed of the motor. The voltage strokes must be properly applied to the two phases of the 3-phase winding system, so that the angle between the stator flux and the rotor flux is kept close to 90°, to get the maximum generated torque. Due to this fact, the motor requires electronic control based on actual rotor angle position for proper operation.

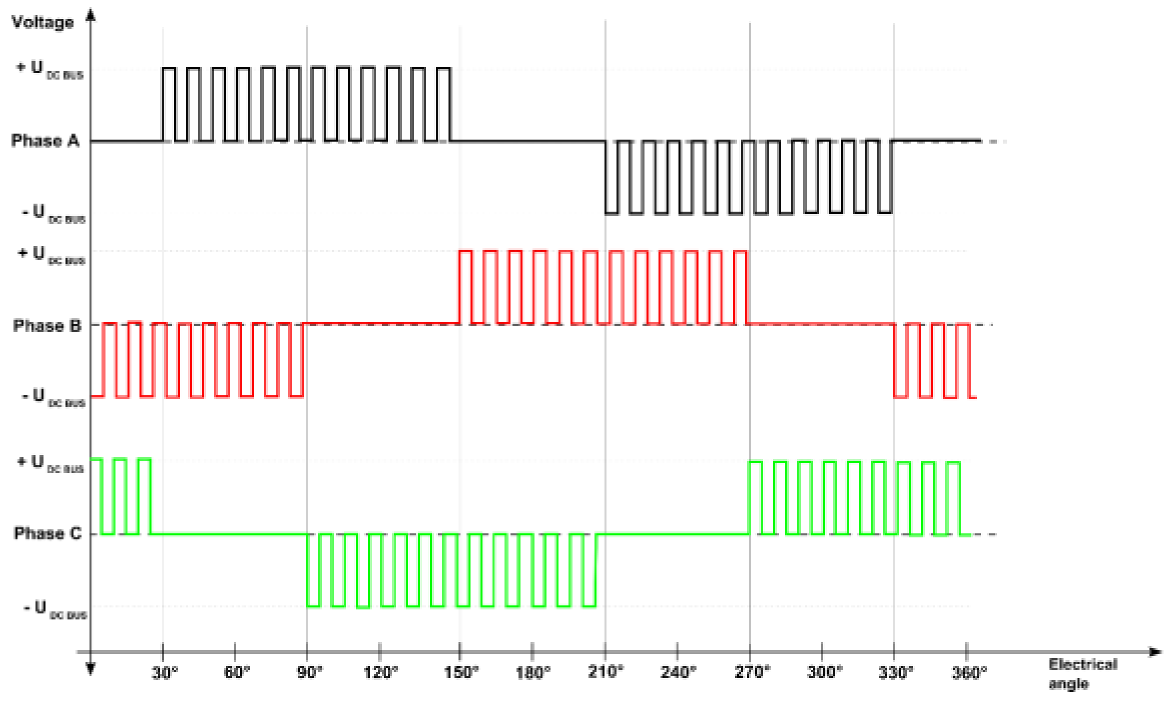


Figure 4. Voltage strokes applied to the 3-ph BLDC motor

For the common 3-phase BLDC motor, a standard 3-phase power stage is used, as illustrated in the figure below. The power stage utilizes six power transistors.

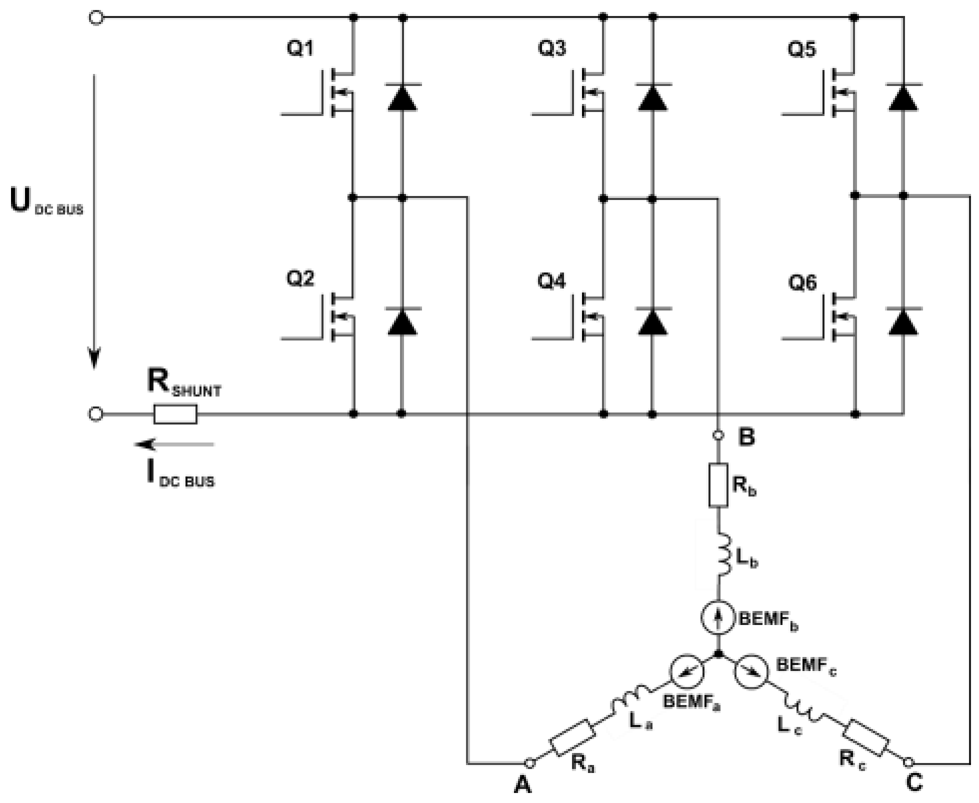


Figure 5. 3-phase BLDC power stage

The 3-phase power stage energizes two motor phases concurrently, while the third phase is unpowered (see Figure 4). There are six possible voltage vectors that are applied to the BLDC motor using PWM switching signals. The unpowered phase can be used for rotor position evaluation in sensorless motor control. The most common BLDC control topology uses the power stage with a constant power source DC voltage. Therefore, the 3-phase average voltage amplitude is controlled by a PWM duty cycle.

2.4 Sensorless BLDC motor control

2.4.1 BLDC motor back-EMF voltage shape

The back-EMF sensing technique is based on the fact that only two phases of the Brushless DC motor are energized at a time. The third phase is a non-fed phase that can be used to sense the back-EMF voltage.

The figure below shows a branch and motor phase winding voltages during a 0 - 360° electrical interval. The yellow interval (4) means a conduction interval of a phase. During this time period, current flows through the winding, and the back-EMF voltage is impossible to measure. Cyan lines (3) determine the time period when the back-EMF voltage can be sensed, as the phase is unpowered. Gray lines (5) show the transient pulse measured on the phase voltages right after the commutation event, which is produced by the current recirculation when the fly-back diodes are conducting. Green lines (6) determine the time period before the zero-cross event, the red line (7) shows the time after zero-cross event, in this period the back-EMF voltage is integrated, and at the end of the red interval, the integrated value of the back-EMF voltage reaches the back-EMF threshold value, and this is the signal to perform a commutation.

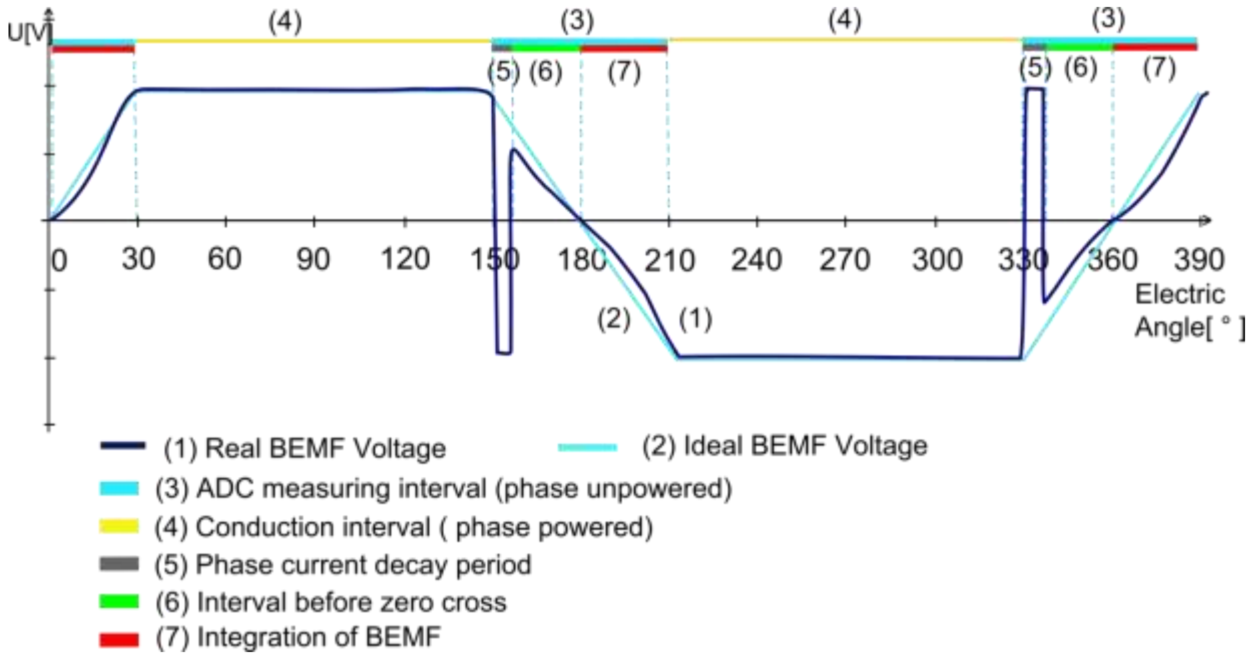


Figure 6. Single phase voltage waveform

For more detailed information about BEMF sensing see the following chapters.

2.4.2 Back-EMF voltage integration method

One of the sensorless BLDC motor control methods is the back-EMF voltage integration. It is based on a simple principle, the integrated value (triangle area) of the non-fed phase's back-EMF voltage after the zero cross is approximately the same at all speeds (S1 $\approx$ S2 $\approx$ S3), as shown in the figure below.

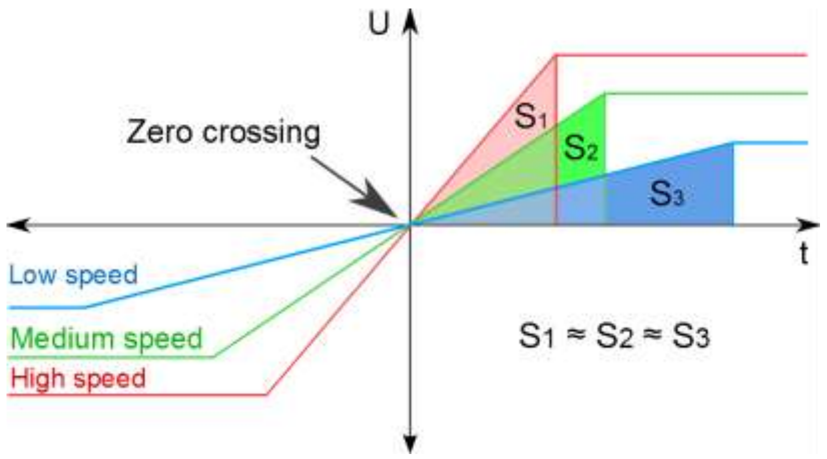


Figure 7. BEMF integration method

Integration starts when the non-fed phase's back-EMF voltage crosses the zero value. Finding of the exact time of the zero-cross point is not very important, as the back-EMF voltage around the zero cross is relatively low, integration of these low voltage values around the zero-cross has small effect to the final integrated value. Commutation is performed when the integrated value reaches a predefined threshold value. If a flux weakening operation is required, current advance can be achieved by decreasing the threshold value.

In comparison to the standard sensorless technique based on the BEMF zero-cross findings, the integration approach is less sensitive to switching noise and offset voltage problems (resistance precisions, noise, and so on). This brings more precise control in low speeds of the motor. On the other hand, the integration approach is less effective in very high speeds of motor (with only a few PWM periods per commutation period). In this case, the sensorless technique, based on the BEMF zero-cross findings, with simple ADC samples interpolation implemented, demonstrates a more precise motor control.

2.4.3 Back-EMF voltage sensing limitations

Accuracy of the sensorless BLDC motor control algorithm, based on back-EMF voltage sensing, is mostly limited by the precision of the measured back-EMF voltage on a non-fed motor's phase. For example, ADC accuracy, precision of phase voltage sensing circuitry, signal noise and distortion caused by the power switching modules, and all their effects, need to be taken into consideration. Noise generated by power switching modules can be eliminated by correcting the setting of measurement event to be farther away from switching edges (PWM-to-ADC synchronization). Some limitations cannot be eliminated, which are called decay or freewheeling periods. As soon as the phase is disconnected from the power by commutation event, there is still current flowing through the freewheeling diode. The conduction freewheeling diode connects the released phase to either a positive or a negative DC bus voltage. The conduction time depends on the momentary load of the motor. In some circumstances, the conduction time is so long that it does not allow for the detection of back-EMF voltage, as represented in the figure below.

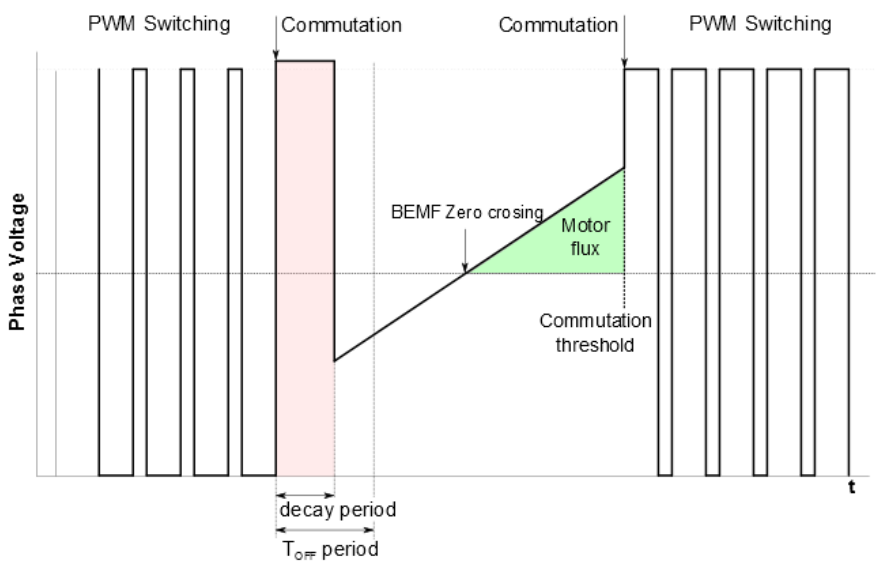


Figure 8. BEMF decay period

For sensorless BLDC motor control, it is important that the period after the back-EMF voltage crosses the zero. If the decay period is too long, it is impossible to control the motor with sensorless algorithm based on the back-EMF technique properly. It is also important to differentiate back-EMF voltage generated by the motor and phase voltage tied to positive or negative DC bus voltage during the decay period. For this purpose, after commutation, the application uses a blank period called the T_{OFF} period parameter. During this T_{OFF} period, the back-EMF voltage is not sensed, and is used for sensorless control. The T_{OFF} period can be calculated from the previous commutation period, and must be calculated with respect to the motor and load parameters to ensure the sensorless algorithm can work correctly during all the application conditions (high motor loads, dynamic speed change, and so on). In this application, the T_{OFF} period is calculated from the previous commutation period using *ui16PeriodToff* parameter, which defines the T_{OFF} period as a proportional value to the previous commutation period.

3 Software Design

This application has been written as a simple C project with the inclusion of Freescale Embedded Software Libraries (FSLESL). These libraries can be downloaded from www.nxp.com/fslesl and contain the necessary algorithms, math functions, filters, and so on, used in the application.

This chapter describes the software design of the sensorless BLDC application. First, the numerical scaling in fixed-point fractional arithmetic of the controller is discussed. Specific issues such as startup and BEMF voltage sensing are explained. Finally, the control software implementation is described.

3.1 Data types

This application uses several data types: (un)signed integer, fractional, accumulator, and floating point. The integer data types are useful for general-purpose computation; they are familiar to the MPU and MCU programmers. The fractional data types allow numeric and digital signal processing algorithms to be implemented. The accumulator data type is a combination of both, which means it has integer and fractional portions.

The floating-point data types are capable of storing real numbers in wide dynamic ranges. The type is represented by binary digits and an exponent. The exponent allows scaling the numbers from extremely small to extremely big numbers. Because the exponent takes part of the type, the overall resolution of the number is reduced when compared to the fixed-point type of the same size.

The following list shows the integer types defined in the libraries:

- Unsigned 16-bit integer—<0 ; 65535> with the minimum resolution of 1
- Signed 16-bit integer—<-32768 ; 32767> with the minimum resolution of 1
- Unsigned 32-bit integer—<0 ; 4294967295> with the minimum resolution of 1
- Signed 32-bit integer—<-2147483648 ; 2147483647> with the minimum resolution of 1

The following list shows the fractional types defined in the libraries:

- Fixed-point 16-bit fractional—<-1 ; 1 - 2^{-15}> with the minimum resolution of 2^{-15}
- Fixed-point 32-bit fractional—<-1 ; 1 - 2^{-31}> with the minimum resolution of 2^{-31}

The following list shows the accumulator types defined in the libraries:

- Fixed-point 16-bit accumulator—<-256.0 ; 256.0 - 2^{-7}> with the minimum resolution of 2^{-7}
- Fixed-point 32-bit accumulator—<-65536.0 ; 65536.0 - 2^{-15}> with the minimum resolution of 2^{-15}

The following list shows the floating-point types defined in the libraries:

- Floating point 32-bit single precision—<-3.40282 * 1038 ; 3.40282 * 1038> with the minimum resolution of 2^{-23}

3.2 Scaling of analog quantities

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

Eqn. 1. $$\boldsymbol{fractional\ value} = \frac{\boldsymbol{Real\ value}}{\boldsymbol{Real\ quantity\ range}}$$

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 [..]

A few examples of how the quantities are scaled are provided in the following subsections.

3.2.1 Voltage scale

The voltage is generally measured on the voltage resistor divider by the ADC. The maximum voltage scale is proportional to the maximum ADC input voltage range. In this example, the maximum voltage is 36.3 V. The example below shows how the fractional voltage variable is used:

Voltage scale: V_{max} = 36.3 V

Measured voltage: $V_{measured}$ = 24 V

Eqn. 2. $$(\boldsymbol{Frac16})\boldsymbol{voltage}_{\boldsymbol{variable}} = \frac{\boldsymbol{V}_{\boldsymbol{measured}}}{\boldsymbol{V}_{\boldsymbol{max}}} = \frac{\mathbf{24\ V}}{\mathbf{36.3\ V}} = \mathbf{0.6612}$$

This 16-bit fractional variable is internally stored as a 16-bit integer variable:

Eqn. 3. $$(\boldsymbol{Int16})\boldsymbol{voltage}_{\boldsymbol{variable}} = (\boldsymbol{Frac16})\boldsymbol{voltage}_{\boldsymbol{variable}}.\mathbf{2^{15}} = \mathbf{0.6612.2^{15}} = \mathbf{21666}$$

The following figure illustrates previous equations of voltage scaling into fractional number. The voltage value is read by ADC as 12-bit signed number with left justification for 16-bit number.

If the floating point number representation of this value is needed, the fractional number is converted to the float number as the following:

Eqn. 4. $$(\boldsymbol{float})\boldsymbol{voltage}_{\boldsymbol{variable}} = \frac{\boldsymbol{V}_{\boldsymbol{max}}}{\mathbf{2^{15}}}(\boldsymbol{Frac16})\boldsymbol{voltage}_{\boldsymbol{variable}}$$

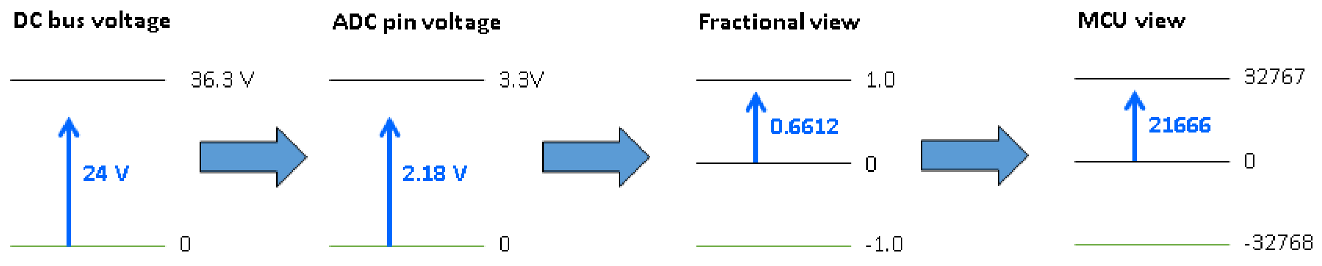


Figure 9: Voltage measurement

3.2.2 Current scale

The current is generally measured as voltage drop on the shunt resistors which is amplified by an operational amplifier. If both the positive and negative currents are needed, the amplified signal has an offset. The offset is generally half of the ADC range. The maximum current scale is proportional to the half of the maximum ADC input voltage range, and the manipulation with the current variable is similar to the voltage variable manipulation. See the figure below.

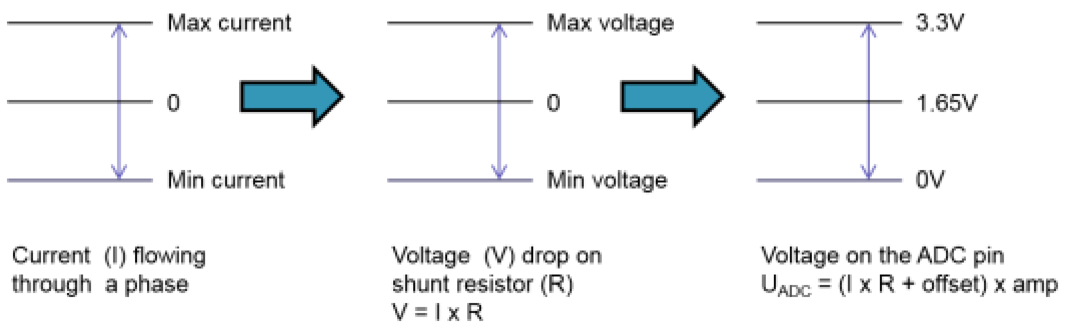

Max current
0
Min current
Current (I) flowing through a phase
Max voltage
0
Min voltage
Voltage (V) drop on shunt resistor (R)
V = I x R
3.3V
1.65V
0V
Voltage on the ADC pin
U_{ADC} = (I x R + offset) x amp

Figure 10: Current measurement

3.2.3 Angle scale

The angles, such as rotor position, are represented as fixed-point 32-bit accumulator, where the lower 16-bit is 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:

Eqn. 5. $$-\pi = 0x8000$$

Eqn. 6. $$\pi = 0x7FFF$$

3.3 Application overview

This application is developed with a focus on easy porting of software across devices or platforms. For this reason, the application software is divided into two separate parts of code:

1. Hardware-dependent code, which are dependent on the hardware boards used and the MCU device's peripheral modules. Includes CPU and peripheral modules initialization, I/O control drivers, and interrupt service routines handling.
2. Hardware-independent code (pure BLDC motor control application).

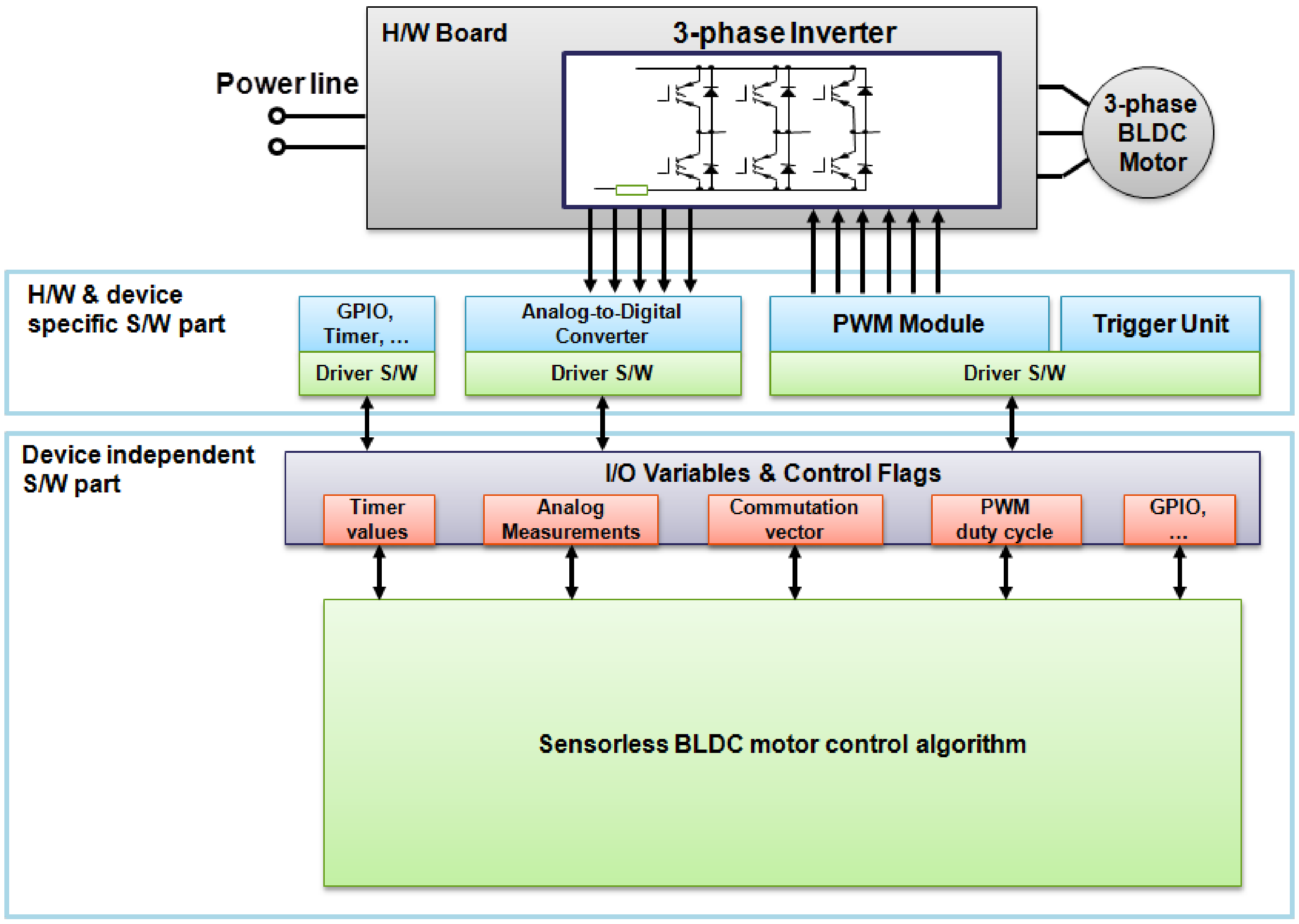


Figure 11. System concept overview

An overview of the system concept of application is shown in the above figure. Input and output variables and output control flags are used as a bridge between the hardware-specific part of code and motor control algorithm. Most of the application input and output variables are stored in a fractional number representation (in range from -1 to 1), and scaling of these variable to and from the real values is calculated in drivers (hardware-related part of code).

3.4 BLDC motor control

The sensorless BLDC motor control method based on back-EMF voltage needs a measurable value of back-EMF voltage. This voltage is proportional to the actual rotor speed, thus it is not measurable in zero and very low speeds. In order to start and run the BLDC motor, the control algorithm has to go through the following states:

- Alignment: initial rotor position setting.
- Startup: forced commutations in open-loop mode.

- Run: close-loop control with BEMF acquisition.

The figure below shows the timing of individual BLDC motor control states.

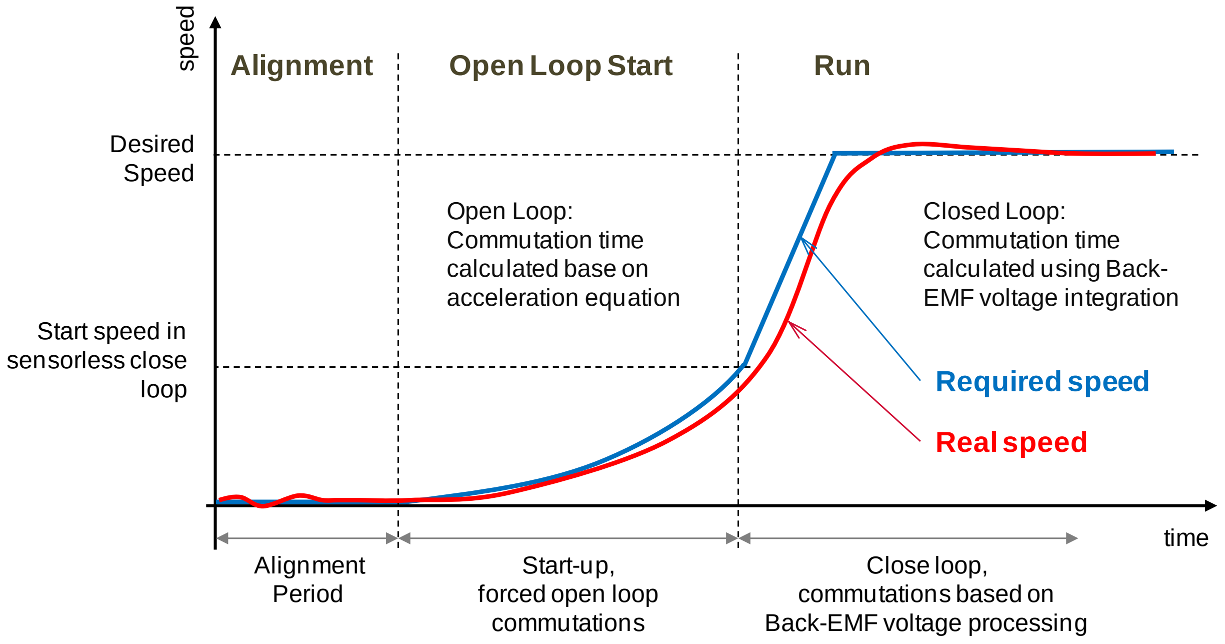


Figure 12. BLDC motor control states

3.4.1 Alignment

As discussed in the earlier sections, the main task for sensorless control of a BLDC motor is position estimation. Before starting the motor, however, the rotor position is not known. The aim of the alignment state is to align the rotor to a known position. This known position enables starting the rotation of the shaft in the desired direction and generating the maximal torque during startup. During the alignment state, all three phases are powered in order to get the required rotor position – alignment vector. Phase A and Phase B are powered with the positive voltage, and Phase C is powered with negative voltage. The alignment time depends on the mechanical constant of the motor, including the load, and also on the applied motor current. In this state, the motor current (torque) is controlled by the PI controller.

3.4.2 Startup

In the startup state, the motor commutations are controlled in an open-loop mode without any rotor position feedback. The open-loop start is required only until the shaft speed is high enough (approximately 3-10% of nominal motor speed) to produce an identifiable back-EMF voltage. The open-loop commutation periods are calculated based on required startup acceleration parameter and number of startup commutations is predefined constant.

3.4.3 Run

The block diagram of the run state is represented in the following figure. It includes the BEMF acquisition with back-EMF integration to control the commutations. The motor speed is based on commutation time periods. The difference between demanded and estimated speeds is fed to the speed PI controller. The output of the speed PI controller is proportional to the voltage to be applied to the BLDC motor. The motor current is measured and filtered in fast loop ISR and is used as feedback into the current controller. The output of the current PI controller limits the output of the speed PI controller. The limitation of the speed PI controller output protects the motor current from exceeding the maximum allowed motor current.

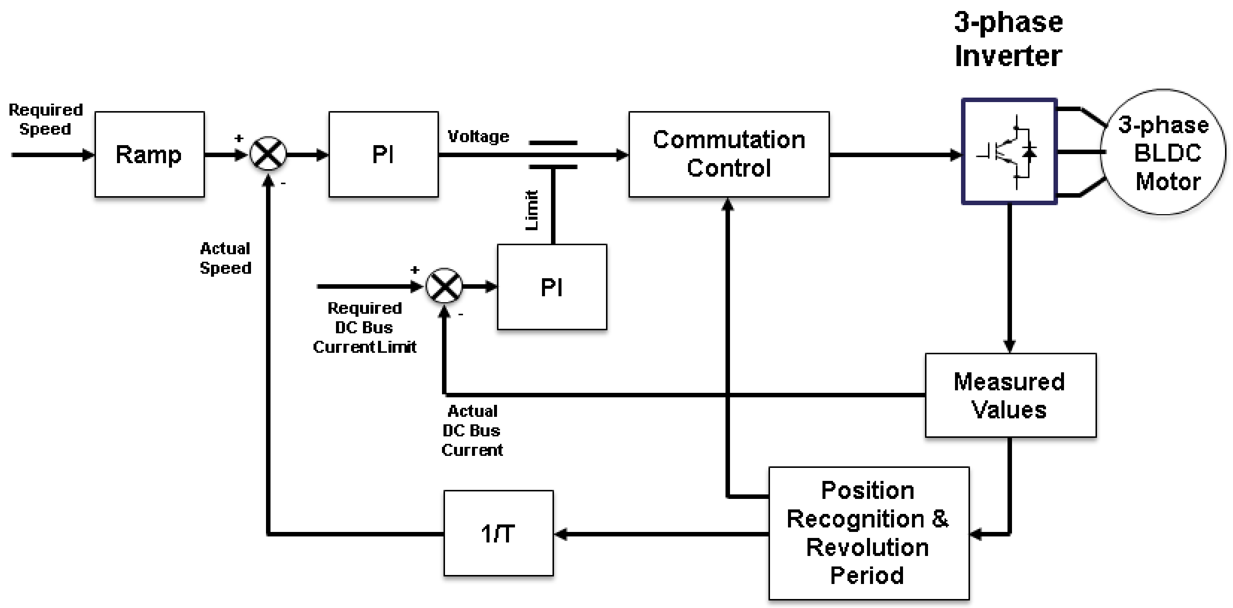


Figure 13. Speed control with current limitation

3.5 Motor control algorithms synchronization

The motor control algorithm is synchronized with the PWM module. The algorithms of the motor control are calculated every PWM reload.

The figure below shows the synchronization of the PWM signals. The user can observe the top and bottom MOSFET PWM signal of the motor, the points where the desired quantities are measured, and the points where the algorithm is calculated. Interrupt periods are dependent on hardware settings. For more information, see Sensorless BLDC Control on Kinetis KV (document AN5263).

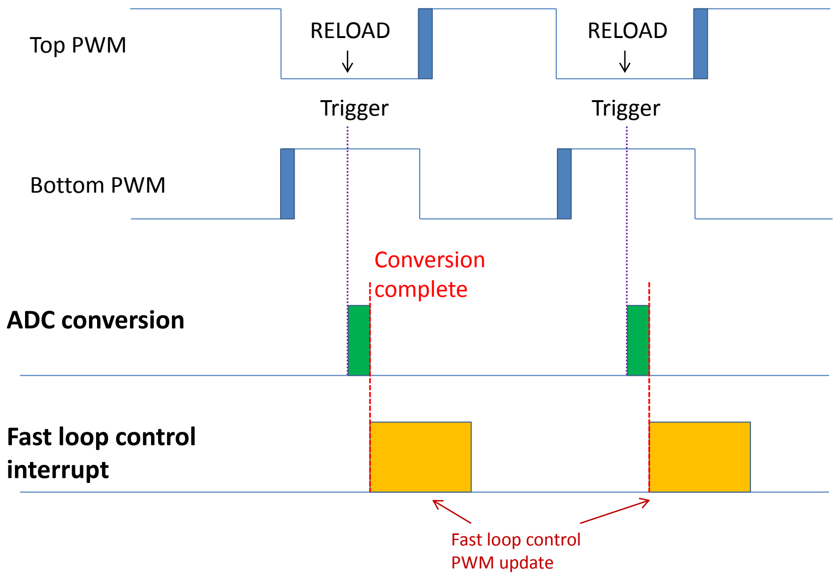


Figure 14. Application timing

3.6 Algorithms call

The whole BLDC motor control application is designed to run in the Interrupt Service Routines only. In the main routine, there is only initialization of the microcontroller, application initialization, and an endless loop with a FreeMASTER poll function to control and monitor the application. The application uses three interrupts with different timing needs:

- Fast Control loop - ADC ISR: executed at every PWM period.
- Slow Control loop - ISR: executed with 1 ms period.
- Time Event ISR: executed according to the current application needs.

The application data flow diagram with the main processes is shown in the following figure. The interrupt service routines are described in detail in the next chapter.

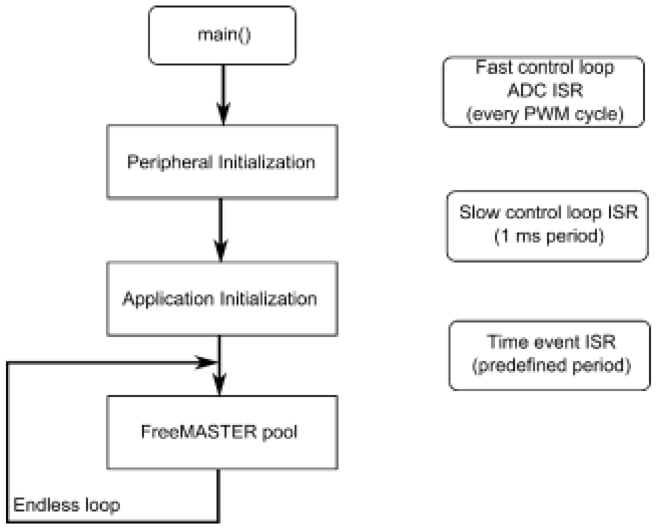

main()
Peripheral Initialization
Application Initialization
FreeMASTER pool
Endless loop
Fast control loop ADC ISR (every PWM cycle)
Slow control loop ISR (1 ms period)
Time event ISR (predefined period)

Figure 2. Application main process overview

Fast control loop ISR and time event ISR uses the same interrupt priority level. Slow control loop ISR uses the lower priority level then previous two ISRs. The application state machine function is executed in fast and time event interrupt service routines.

3.6.1 I/O values processing

This application uses the concept of a separate MCU-specific part of code, and hardware-independent BLDC motor control algorithm. The hardware-related code reads the input values from the peripheral modules (timer counter, ADC result, and others), scales them to the fractional number representation if needed, and stores as the input variables. These input variables are then processed in the hardware-independent BLDC motor control algorithm part of software. Similarly, the BLDC algorithm output variables are processed and updated into the respective peripheral modules in hardwar-related code.

3.6.2 Input application values

Input application variables are updated at the beginning of respective interrupt service routines:

- *ui16TimeCurrent*—actual time value. If the timer is used for the timing purpose, the actual timer counter register value is stored in this variable.
- *ui16TimeCurrentEvent*—time of the last *Time Event*. If the timer is used for this timing purpose, the timer value register is stored in this variable.
- *f16IDcBusFilt*—Filtered DC bus current value in fractional number representation is stored in this 16-bit variable. -1 means the maximal negative DC Bus current; +1 means maximum positive DC Bus current.

- *f16DcBusCurrentOffset*—DC bus current offset value in a fractional number representation is stored in this 16-bit variable. The value is measured during the calibration state after the application initialization.
- *f16UDcBusFilt—Filtered* DC bus voltage value in fractional number representation is stored in this 16-bit variable. 0 means 0 Volts, +1 means maximal measurable DC bus voltage.
- *f16UPhase*—phase voltage value in fractional number representation is stored in this 16-bit variable. 0 means 0 Volts, +1 means maximum measurable DC bus voltage.
- *f16UPhaseBemf*—calculated back-EMF voltage value in fractional number representation is stored in this 16-bit variable. The following equation is used to calculate the value (*f16UDcBusFilt / 2* is considered as a middle point of the motor windings):

$$\mathrm{f16UPhaseBemf} = \mathrm{f16UPhase} - \frac{\mathrm{f16UDcbBusFilt}}{2}$$

3.6.3 Output application functions

There can be pending request for hardware output update after the execution of BLDC motor control algorithm. For this purpose following functions are executed.

MCDRV_FtmSetPwmOutput– applies the right commutation vector according to rotor position. When aligning, it is alignment vector. If a fault occurs, PWM outputs are disabled.

MCDRV_FtmSetDutyCycle – updates duty cycle generated with PWM module. In variable *f16DutyCycle* there is stored new value of duty cycle in the fractional number representation. Fractional number value in range <-1; +1) is then processed and scaled in appropriate PWM driver function to fit the actual PWM module configuration.

MCDRV_FtmCmtSet – set a new time of *Time Event*. In variable *ui16TimeNextEvent* there is stored new time to be set. If timer is used for timing purpose, timer value register is updated with the value of *ui16TimeNextEvent*.

3.7 Application state machine function

Global application state and BLDC motor state are controlled using two state machine functions:

Main application state machine

Motor's state machine

Main application state machine function globally controls the whole application. It serves as a global application switch including the fault state processing. *Motor's state Machine* function controls the motor state. If more than one motor is driven by application, each motor uses its own application substate machine.

3.7.1 Main application state machine

The state machine structure has been unified for all components of the application and consists of four main states. See the figure below. The states are the following:

- Fault—system detects a fault condition and is waiting until it's cleared.

- Init—variables' initialization.
- Stop—system is initialized and waiting for the Run command.
- Run—system is running and can be stopped by the Stop command.

There are transition functions between these state functions:

- Init > Stop—initialization was done and the system is entering the Stop state.
- Stop > Run—the Run command has been applied and the system is entering the Run state if the Run command has been acknowledged.
- Run > Stop—the Stop command was applied, the system is entering the Stop state if the Stop command has been acknowledged.
- Fault > Init—fault flag was cleared and the system is entering the Init state.
- Init, Stop, Run > Fault—a fault condition occurred and the system is entering the Fault state.

The state machine structure uses the following flags to switch between the states:

- SM_CTRL_INIT_DONE—if this flag is set, the system goes from the Init to the Stop state.
- SM_CTRL_FAULT—if this flag is set the system goes from any state to the Fault state.
- SM_CTRL_FAULT_CLEAR—if this flag is set, the system goes from the Fault state to the Init state.
- SM_CTRL_START—this flag informs the system that there is a command to go from the Stop state to the Run state. The transition function is called; nevertheless, the action must be acknowledged. The reason is that sometimes it can take time before the system gets ready to be switched on.
- SM_CTRL_RUN_ACK—this flag acknowledges that the system can proceed from the Stop state to the Run state.
- SM_CTRL_STOP—this flag informs the system that there is a command to go from the Run state to the Stop state. The transition function is called; nevertheless, the action must be acknowledged. The reason is the system must be properly turned off, which can take time.
- SM_CTRL_STOP_ACK—this flag acknowledges that the system can proceed from the Run state to the Stop state.

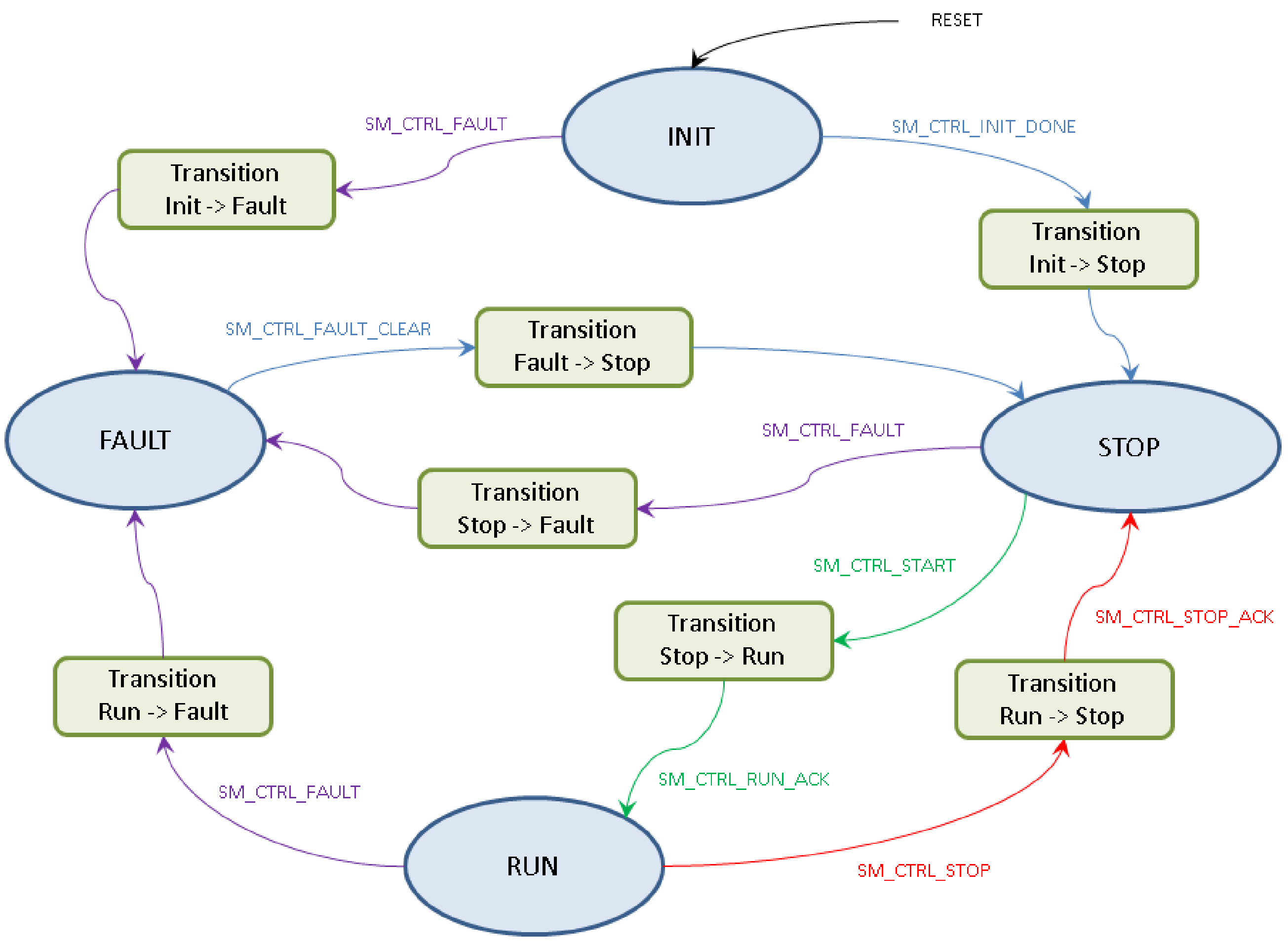


Figure 3. Application State Machine diagram

The implementation of this structure of state machine is done in the state_machine.c .h files. The following code lines define the state machine structure:

```
/* State machine control structure */
typedef struct
{
    SM_APP_STATE_FCN_T const*      psState;  /* State functions */
    SM_APP_TRANS_FCN_T const*      psTrans;  /* Transition functions */
    SM_APP_CTRL                    uiCtrl;   /* Control flags */
    SM_APP_STATE_T                 eState;   /* State */
} SM_APP_CTRL_T;
```

The four variables used in the code are defined as follows.

- *psState*—pointer to the user state machine functions. The particular state machine function from this table is called when the state machine is in a specific state.
- *psTrans*—pointer to the user transient functions. The particular transient function is called when the system goes from one state to another.

- *uiCtrl*—this variable is used to control the state machine behavior using the above mentioned flags.
- *eState*—this variable determines the actual state of the state machine

The user state machine functions are defined in the following structure:

```
/* User state machine functions structure */
typedef struct
{
    PFCN_VOID_VOID       Fault;
    PFCN_VOID_VOID       Init;
    PFCN_VOID_VOID       Stop;
    PFCN_VOID_VOID       Run;
} SM_APP_STATE_FCN_T;
```

The user transient state machine functions are defined in the following structure:

```
/* User state-transition functions structure*/
typedef struct
{
    PFCN_VOID_VOID       FaultInit;
    PFCN_VOID_VOID       InitFault;
    PFCN_VOID_VOID       InitStop;
    PFCN_VOID_VOID       StopFault;
    PFCN_VOID_VOID       StopRun;
    PFCN_VOID_VOID       RunFault;
    PFCN_VOID_VOID       RunStop;
} SM_APP_TRANS_FCN_T;
```

These are the definitions for the control flag variable:

```
typedef unsigned short SM_APP_CTRL;

/* State machine control command flags */
#define SM_CTRL_NONE            0x0
#define SM_CTRL_FAULT           0x1
#define SM_CTRL_FAULT_CLEAR     0x2
#define SM_CTRL_INIT_DONE       0x4
#define SM_CTRL_STOP            0x8
#define SM_CTRL_START           0x10
#define SM_CTRL_STOP_ACK        0x20
#define SM_CTRL_RUN_ACK         0x40
```

These are the definitions for the state identification variable:

```
/* Application state identification enum */
typedef enum
{
    FAULT   = 0,
    INIT    = 1,
    STOP    = 2,
    RUN     = 3,
} SM_APP_STATE_T;
```

The state machine must be periodically called from the code using the following inline function. This function input is the pointer to the state machine structure described above. This structure is declared and initialized in the code where the state machine is called:

```
/* State machine function */
static inline void SM_StateMachineFast(SM_APP_CTRL_T *sAppCtrl)
{
          gSM_STATE_TABLE_FAST[sAppCtrl -> eState](sAppCtrl);
}
```

The particular example that shows how to initialize and use the state machine structure is in the following section.

3.8 Motor state machine

The motor state machines are based on the main state machine structure. The Run state sub-states have been added on top of the main structure to control the motors properly. Following is the description of the main states' user functions.

- Fault—the system faces a fault condition and waits until the fault flags are cleared. The DC-bus voltage and DC-bus current is measured and filtered.
- Init—variables' initialization
- Stop—the system is initialized and waits for the Run command. The PWM output is disabled. The DC-bus voltage is measured and filtered.
- Run—the system is running and can be stopped by the Stop command. The Run sub-state functions are called from here.

There are transition functions between these state functions:

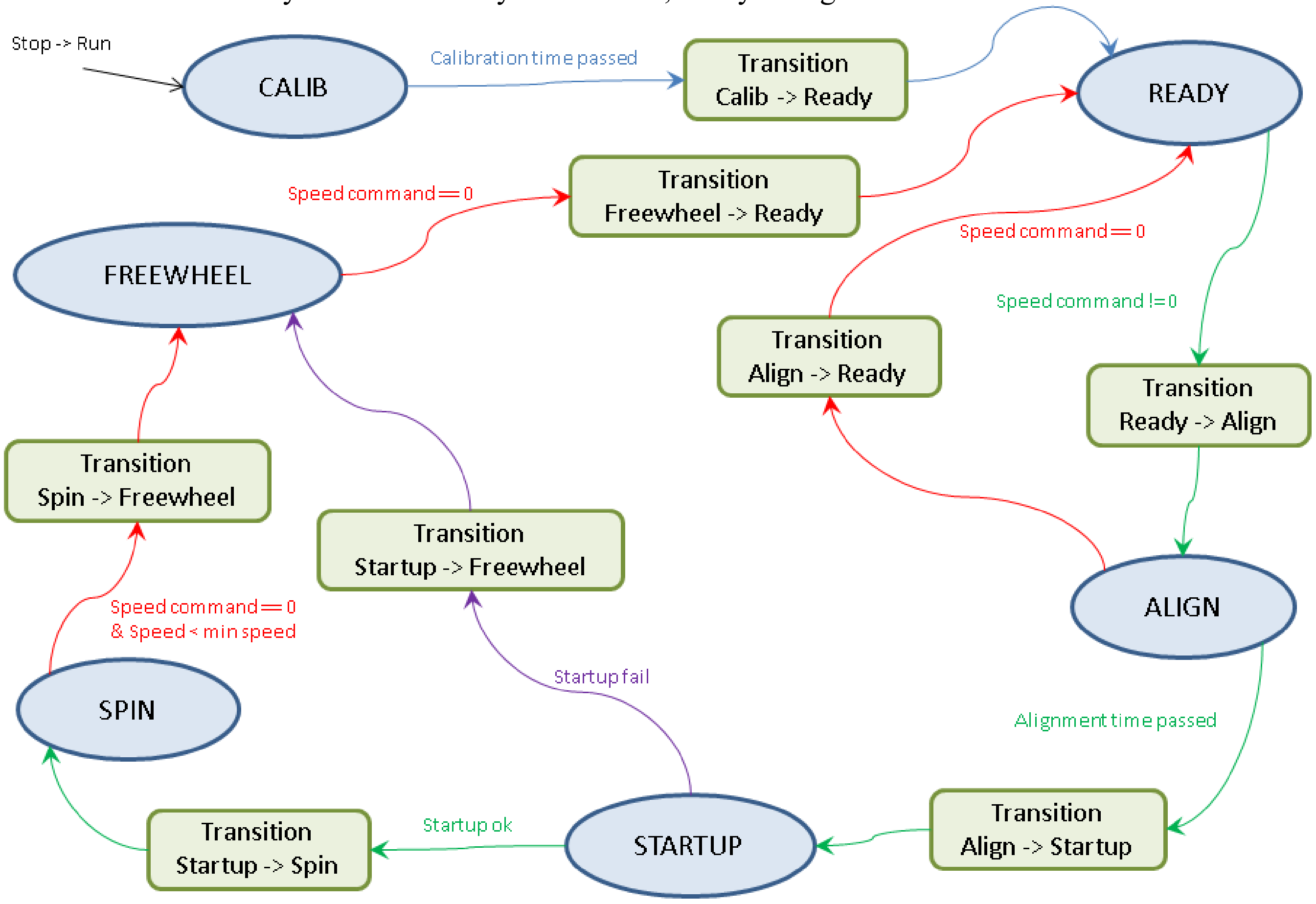
- Init > Stop—nothing is processed in this function.
- Stop > Run:
  - The duty cycle is initialized to 50%.
  - PWM output is enabled.
  - The current ADC channels are initialized.
  - The Calib sub-state is set as the initial Run sub-state.
- Run > Stop:

    - The Stop command has been applied and the system is entering the Stop state if the Stop command has been acknowledged. The system does not go directly to Stop when the system is in certain Run sub-states.
- Fault > Stop
    - All state variables and accumulators are initialized.
- Init, Stop > Fault
    - The PWM output is disabled.
- Run > Fault:
    - Certain current and voltage variables are zeroed.
    - The PWM output is disabled.

The Run sub-states are called when the state machine is in the Run state. The Run sub-state functions are the following:

- Calib:
    - The current channels ADC offset calibration.
    - After the calibration time expires, the system is switched to Ready.
    - The DC-bus voltage is measured and filtered.
    - The PWM is set to 50% and its output is enabled.
    - The DC-bus current channel offsets are measured and filtered.
- Ready:
    - The PWM is set to 50% and its output is enabled.
    - The DC-bus current is measured and filtered the ADC channels are set up.
    - DC-bus voltage is measured and filtered.
    - Phase-voltage is measured and BEMF voltage is calculated.
    - Certain variables are initialized.
- Align:
    - The DC-bus current is measured and filtered the ADC channels are set up.
    - DC-bus voltage is measured and filtered.
    - The rotor alignment algorithm is called and the PWM is updated.
    - After the alignment time expires, the system is switched to Startup.
- Startup:
    - The DC-bus current is measured and filtered the ADC channels are set up.
    - Phase-voltage is measured and BEMF voltage is calculated.
    - If the startup is successful, the system is switched into Spin, otherwise to Freewheel.
    - The DC-bus voltage is measured and filtered.
    - The open-loop startup algorithm is called.
    - The estimated speed is calculated from filtered (if the filter is necessary).
- Spin:
    - The DC-bus current is measured and filtered.
    - Phase-voltage is measured and BEMF voltage is calculated.
    - The BLDC sensorless commutation algorithm is called and the PWM is updated.
    - The motor spins.
    - The DC-bus voltage is measured and filtered.
    - The estimated speed is calculated from measured BEMF voltage.
    - The speed ramp and the speed PI controller algorithms are called.

  - The speed command and speed overload condition are evaluated.
- Freewheel:
  - The PWM output is disabled and the module is set to 50%.
  - The current is measured and the ADC channels are set up. The DC-bus voltage is measured and filtered.
  - The system waits in this sub-state for certain time which is generated due to rotor inertia, it means to wait until the rotor stops itself.
  - Then the system evaluates the conditions and proceeds into one of these sub-states: Align or Ready. If several faulty starts occur, the system goes into the Fault state.



Stop -> Run
CALIB
Calibration time passed
Transition Calib -> Ready
READY
Speed command == 0
Transition Freewheel -> Ready
FREEWHEEL
Speed command == 0
Speed command != 0
Transition Align -> Ready
Transition Ready -> Align
Transition Spin -> Freewheel
Transition Startup -> Freewheel
ALIGN
Speed command == 0 & Speed < min speed
SPIN
Startup fail
Alignment time passed
Transition Startup -> Spin
Startup ok
STARTUP
Transition Align -> Startup

Figure 4. Motor Run sub-state diagram

The Run sub-states also have the transition functions that are called in between the sub-states’ transition. The sub-state transition functions are the following:

- Calib > Ready—calibration done, entering the Ready state.
- Ready > Align—non-zero-speed command; entering the Align state.
  - Align current controller is initialized.
  - The startup commutation counter is set to 1.
  - The alignment time is set up.
  - Applied alignment vector.
- Align > Ready—zero-speed command; entering the Ready state.

- Align > Startup—alignment done; entering the Startup state.
  - Initialize startup commutation period counter and time.
  - Select next PWM sector based on required spin direction.
- Startup > Spin—startup successful; entering the Spin state.
  - Clear BEMF integrator.
  - Reset commutation error counter.
- Startup > Freewheel—startup failed; entering the Freewheel state.
  - The Freewheel time is set up.
  - PWM outputs are disabled.
- Spin > Freewheel—zero-speed command; entering the Freewheel state.
  - The Freewheel time is set up.
- Freewheel > Ready—zero-speed command; entering the Ready state.
  - The PWM output is disabled.

The implementation of this structure of motor state machine is done in the *m1_statemachine.c* and *m1_statemachine.h* files. Each state is doubled. States with suffix *Fast* are called in fast loop and states with suffix *Slow* are called in slow loop. However, the state transitions are not doubled and the state machine flow is controlled from the fast-loop state machine only. The following is a description of the main state-machine structure:

- The main states' user function prototypes:

```
static void M1_StateFault(void);
static void M1_StateInit(void);
static void M1_StateStop(void);
static void M1_StateRun(void);
```

- The main states' user transient function prototypes:

```
static void M1_TransFaultStop(void);
static void M1_TransInitFault(void);
static void M1_TransInitStop(void);
static void M1_TransStopFault(void);
static void M1_TransStopRun(void);
static void M1_TransRunFault(void);
static void M1_TransRunStop(void);
```

- The main states functions table initialization:

```
/* State machine functions field */
static const SM_APP_STATE_FCN_T msSTATE = {M1_StateFault, M1_StateInit, M1_StateStop,
M1_StateRun};
```

- The main state transient functions table initialization:

```
/* State-transition functions field */
static const SM_APP_TRANS_FCN_T msTRANS = {M1_TransFaultInit, M1_TransInitFault,
M1_TransInitStop, M1_TransStopFault, M1_TransStopRun, M1_TransRunFault,
M1_TransRunStop};
```

- Finally, the main state machine structure initialization:

```
/* State machine structure declaration and initialization */
SM_APP_CTRL_T gsM1_Ctrl =
{
        /* gsM1_Ctrl.psState, User state functions */
        &msSTATE,

        /* gsM1_Ctrl.psTrans, User state-transition functions */
        &msTRANS,

        /* gsM1_Ctrl.uiCtrl, Default no control command */
        SM_CTRL_NONE,

        /* gsM1_Ctrl.eState, Default state after reset */
        INIT
};
```

- Similarly, the Run sub-state machine is declared. Thus, the Run sub-state identification variable has the following definitions:

```
typedef enum {
    M1_CALIB                = 0,
    M1_READY                = 1,
    M1_ALIGN                = 2,
    M1_STARTUP              = 3,
    M1_SPIN                 = 4,
    M1_FREEWHEEL            = 5
} M1_RUN_SUBSTATE_T;          /* Run sub-states */
```

- For the Run sub-states, two sets of user functions are defined. The user function prototypes of the Run sub states are as follows.

```
static void M1_StateRunCalib(void);
static void M1_StateRunReady(void);
static void M1_StateRunAlign(void);
static void M1_StateRunStartup(void);
static void M1_StateRunSpin(void);
static void M1_StateRunFreewheel(void);
```

- The Run sub-states’ user transient function prototypes are given below.

```
static void M1_TransRunCalibReady(void);
static void M1_TransRunReadyAlign(void);
static void M1_TransRunAlignStartup(void);
static void M1_TransRunAlignReady(void);
static void M1_TransRunStartupSpin(void);
static void M1_TransRunStartupFreewheel(void);
static void M1_TransRunSpinFreewheel(void);
static void M1_TransRunFreewheelAlign(void);
static void M1_TransRunFreewheelReady(void);
```

- The Run sub-states functions table initialization:

```
/* Sub-state machine functions field */
static const PFCN_VOID_VOID mM1_STATE_RUN_TABLE[6] = {
        M1_StateRunCalib,
        M1_StateRunReady,
        M1_StateRunAlign,
        M1_StateRunStartup,
        M1_StateRunSpin,
        M1_StateRunFreewheel
};
```

- The state machine is called from the interrupt service routine as mentioned in Algorithms call. The code syntax used to call the state machine is given below.

```
/* StateMachine call */
SM_StateMachine(&gsM1_Ctrl);
```

- Inside the user Run state function, the sub-state functions are called as follows:

```
/* Run sub-state function */
mM1_STATE_RUN_TABLE[meM1_StateRun]();
```

where the variable identifies the Run sub-state.

3.9 Sensorless BLDC control

3.9.1 Fast control loop ISR

Fast control loop ISR is executed at every PWM cycle and is used for a rotor position recognition and commutation. Fast control loop ISR uses the same priority level as of time event ISR (it must not be interrupted by time event ISR), which is higher than slow control loop ISR. The overview of fast control loop ISR is represented in the following figure:

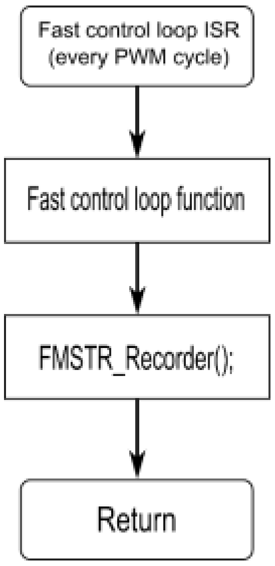


Figure 5. Fast control loop ISR

Inside the fast control loop ISR there is a hardware independent *state machine function* call.

3.9.2 Fast control loop function

The main function in fast control loop ISR is *state machine* function. This function is hardware independent; it uses only input and output variables and flags to interface hardware drivers and application code.

Detailed diagram of a fast control loop function is shown in Figure 6.

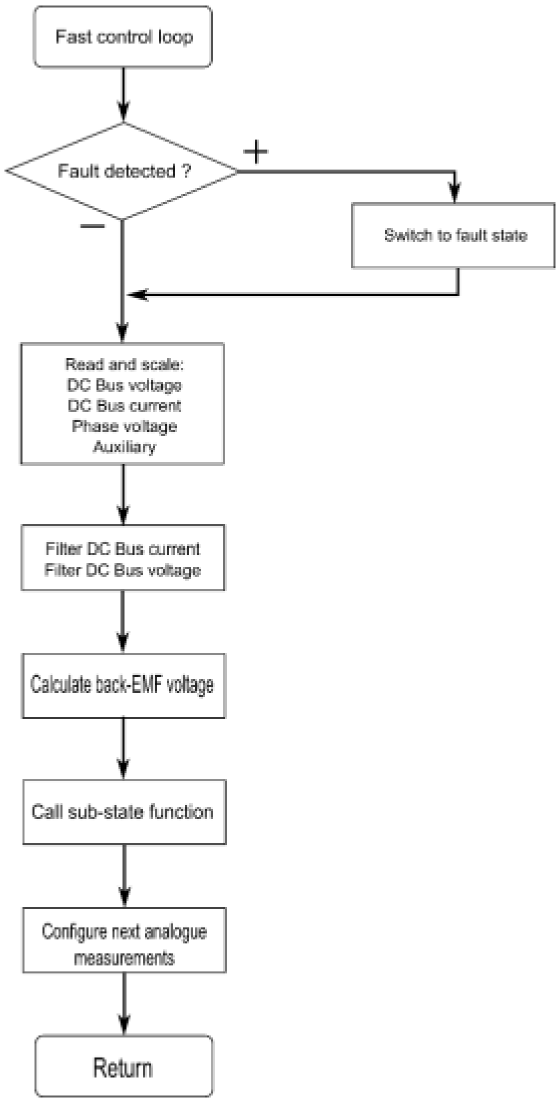


Figure 6. Fast control loop function

In the beginning of a fast control loop function the fault condition is tested. The results of analog measurements are scaled in 16-bit fractional number representation and stored to the following variables:

- f16IDcBus
- f16UPhase
- f16UDcBus

The DC Bus current and DC bus voltage is filtered using library GDFLIB_*FilterIIR1_F16* function (IIR filter) and stored in *f16IDcBusFilt and f16UDCBusFilt* variable respectively. Phase back-EMF voltage *f16UPhaseBemf* is calculated using phase voltage and DC Bus voltage values. Then *fast control loop substate* function is called according to current application substate (calib, freewheel, ready, align startup, spin). For example, if the actual sub-state is the spin state, the BEMF integration and commutation process is performed in spin sub-state function.

At the end of fast control loop the analog measurement configuration for next PWM cycle module is set according to the updated BLDC sector.

3.9.3 Slow control loop ISR

Slow control loop ISR is executed at every 1 ms, and is used for speed and torque evaluation and control. Slow control loop ISR uses a lower priority level in comparison with other ISRs used in the application. Therefore, it can run more time-consuming functions. The overview of slow control loop ISR is represented in the following figure:

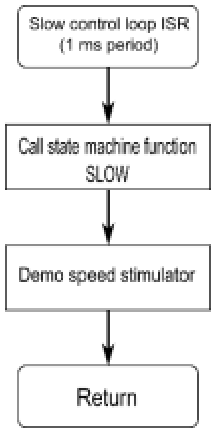


Figure 20. Slow control loop ISR

Inside the slow control loop ISR there is a hardware-independent *state machine function* call and *Demo speed stimulator function*. Demo speed stimulator function stimulates required motor speed in demo mode only.

3.9.4 Slow control loop function

The main function in a slow control loop ISR is the *state machine slow* function. This function is hardware-independent; it uses only the input and output variables and flags to interface hardware drivers and application code. According to the actual application main state and sub-state, dedicated sub-state function is called (calib, freewheel, ready, align, startup, spin).

The detailed diagram of slow control loop function for spin state is shown in the following figure:

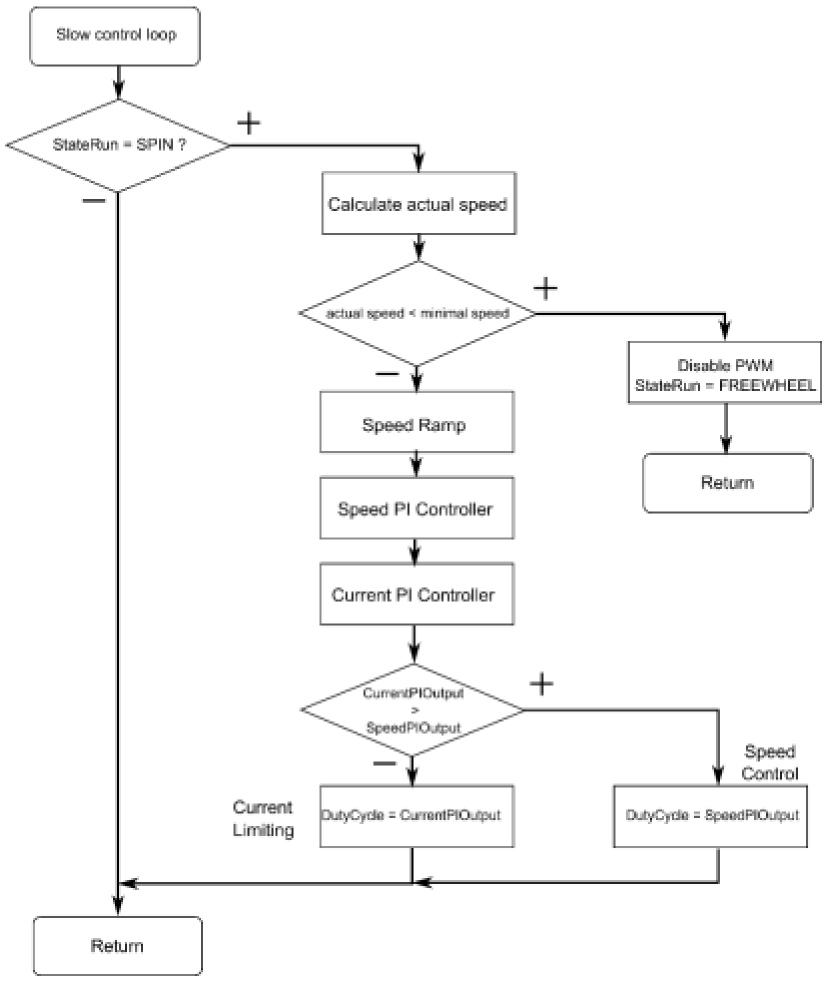


Figure 21. Slow control loop function

A detailed description of the slow loop function in their respective states is described in the following chapters.

3.9.4.1 Slow loop function - Spin state

In Spin state, the actual motor speed (*f16SpeedMeasured*) is calculated using the last six commutation periods (*ui16PeriodCmt[n]*). The library *GFLIB_DivSat* function is used for fractional division. Because of the saturation feature in this function, the result cannot overflow. In this case, the actual speed is lower than the minimal allowed speed (*f16SpeedMinimal*), and the application enters the Freewheel state. From the actual speed and required speed (*f16SpeedRequired*) the speed ramp output (*f16SpeedRampOutput*) is calculated using the library *GFLIB_Ramp* function.

According to the actual direction of rotation the sign of f16SpeedMeasured variable is set and input to the speed PI controller (f16SpeedPiErr) is calculated. The speed PI controller and current PI controller are calculated using the library GFLIB_ControllerPIpAW function. If speed PI controller output is higher than the output of current PI controller, then "current limiting" is applied to the output, which means the current PI controller output is used instead of the speed PI controller output. This reduces the speed of the motor, but the phase current does not exceed the limit value in f16IDcBusLim variable.

3.9.4.2 Slow loop function - Alignment state

In the Alignment state, the slow control loop function checks whether the required speed is higher than minimal. If this is not the case, it is switched to the Freewheel state. This function is also used for timing purposes, and decreases ui16TimeAlignment variable every 1 ms down to zero, as well as tests if the alignment time runs out.

3.9.4.3 Slow loop function - Calibration state

In the Calibration state, the DC Bus current offset is measured, and at the end of calibration process the offset is stored in the ui16OffsetDcCurr variable.

3.9.4.4 Slow loop function - Freewheel state

In the Freewheel state, the application is waiting until the Freewheel time passes out.

3.9.4.5 Slow loop function - Startup state

In the Startup state, the required speed is compared to the minimal speed. If the required speed is less than the minimal speed, the application is switched into the Freewheel state.

3.9.5 Time event ISR

The time event ISR is used for individual application states timing, open-loop commutations timing in the Startup state, and to protect the motor from over-currents in the Spin state using "safety" commutations, if a sensorless motor control algorithm cannot recognize a rotor's position for any reason. The overview of time event ISR is represented in the following figure:

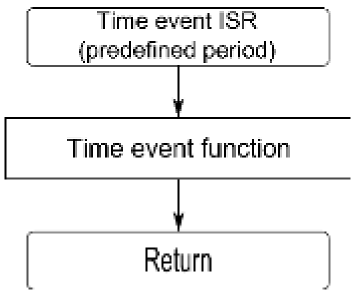


Figure 22. Time event ISR

3.9.6 Time event function

The time event function has a different functionality based on the actual motor's sub-state machine state. In the Startup state, the time event function is used for the open-loop commutations timing, where each following commutation period is calculated from the previous commutation period. In the Spin state, the time event function serves for the “safety” commutations if a sensorless motor control algorithm cannot recognize the rotor’s position for any reason. The detailed diagram of time event function is represented in the following figure.

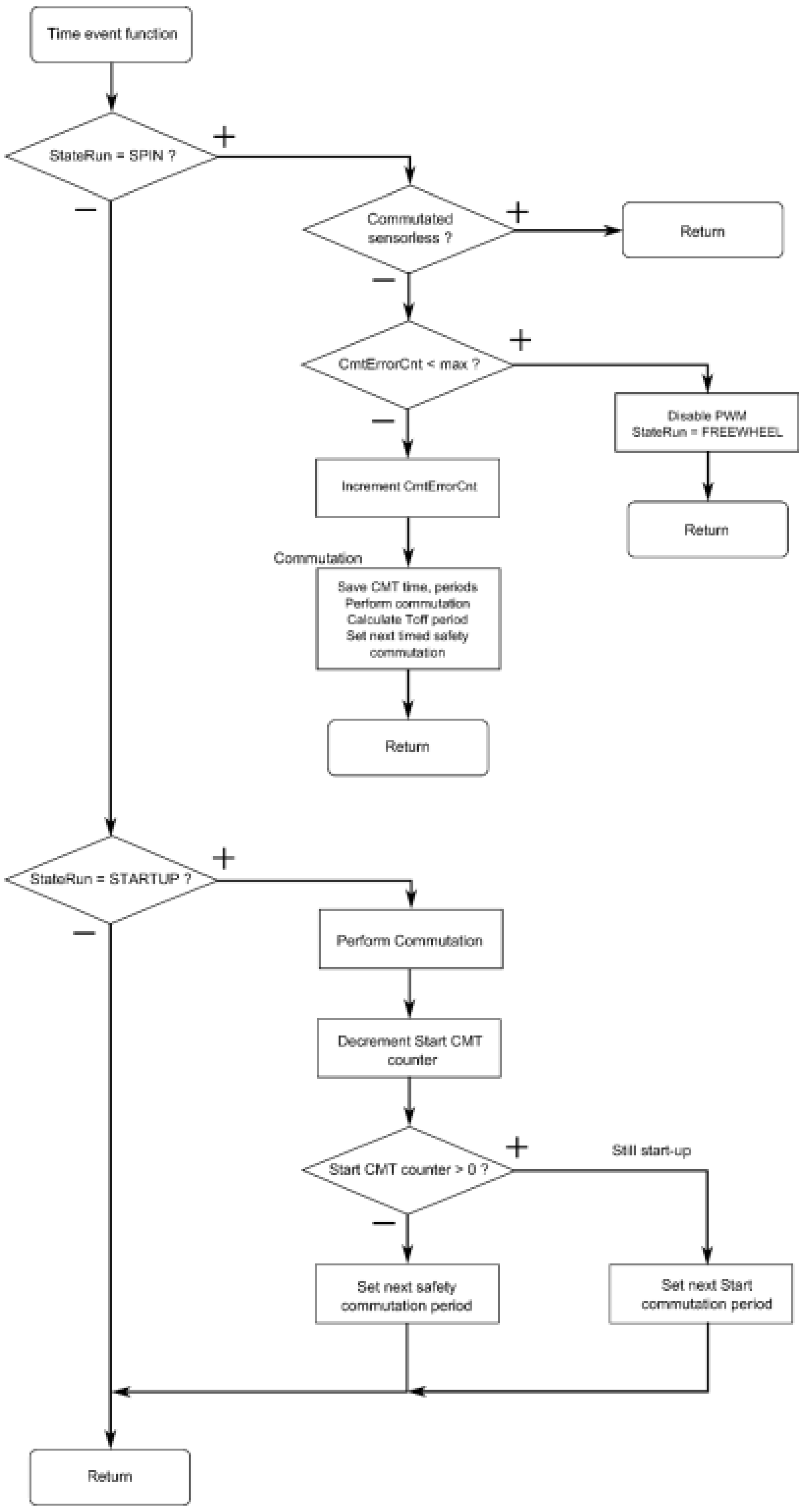

Time event function
StateRun = SPIN ?
+
–
Commutated sensorless ?
+
Return
–
CmtErrorCnt < max ?
+
Disable PWM StateRun = FREEWHEEL
Return
–
Increment CmtErrorCnt
Commutation
Save CMT time, periods Perform commutation Calculate Toff period Set next timed safety commutation
Return
StateRun = STARTUP ?
+
Perform Commutation
Decrement Start CMT counter
Start CMT counter > 0 ?
+
Still start-up
–
Set next safety commutation period
Set next Start commutation period
–
Return

Figure 23. Time event function

3.9.6.1 Time event function - Spin state

In the Spin state, if a sensorless BLDC motor control algorithm is working fine, the time event ISR is not executed because the commutations are performed in a fast control loop function, executed in fast control loop ISR. If the sensorless BLDC motor control algorithm cannot recognize the rotor's position correctly, the time event ISR is used for the "safety" commutation of the motor.

The safety commutation period is set doubled from the last commutation period, while the motor is being commutated in the fast control loop function. This is a window where the next commutation should be performed by the sensorless BLDC motor control algorithm during the next commutation period. If the sensorless algorithm cannot recognize the rotor's position correctly, the time event ISR is used for the "safety" commutation of the motor. Safety commutations limit the overcurrents in a motor if sensorless algorithm fails.

At the beginning of time event, the function b*CommutatedSnsless* variable is checked. If it is set, then the commutation was performed in fast control loop just before the time event ISR was originated. In that case, the function exits because the sensorless motor control algorithm still works, even it was commutated at the end of the window established for the sensorless commutations. Otherwise, the safety commutation is going to be performed.

Every time the safety commutation is performed *ui16CmtErrorCnt* variable is increased by 3. Conversely, when the motor is commutated in fast control loop using sensorless algorithm, *ui16CmtErrorCnt* is decreased. If *ui16CmtErrorCnt* is higher than the maximum allowed number of commutation fails, the application enters Freewheel state. If the number of errors is less than the limit, then the application continues same way as in fast control loop. The actual commutation time variable *ui16TimeOfCmt* is updated, the commutation period variable is ui16PeriodCmt[n] is calculated from the actual and previous commutation times, and the commutation is performed calling *BldcCommutation()* function.

The new value of T_{OFF} period *ui16PeriodToff* is calculated from the last commutation period. The integrated value of back-EMF voltage (*f32UBemfIntegSum*) is cleared and the next safety commutation (*ui16TimeNextEvent*) is set to as double of last commutation period.

3.9.6.2 Time event function - Startup state

In the Startup state, the time event function is used for open-loop commutations timing. At the beginning the commutation is performed. Then number of open-loop commutations is decreased and checked (*ui16CounterStartCmt*). If it is still higher than zero, the next commutation period is calculated from the previous commutation period using library function *MLIB_Mul_F16* - fractional multiplication of previous commutation period *ui16PeriodCmtNext* and acceleration coefficient *f16StartCmtAcceleration*. The acceleration coefficient is calculated from the first commutation period after an alignment and the target Startup speed. If all the Startup commutations were performed, the safety commutation is set as double of the last startup commutation period.

3.10 Peripherals control

Peripherals control such as the PWM and ADC setting, as well as the current reading and PWM duty cycle update are done via Motor Control Drivers (MCDRV). From the application perspective, the MCDRV have unified API. The MCDRV provide the hardware abstractive layer so the application code is hardware-independent. The MCDRV have a low-level hardware-dependent layer. This layer is MCU and platform dependent.

4 Acronyms and abbreviations

Table 1. Acronyms

Term	Meaning
ADC	Analog-to-digital converter.
BEMF	Back-electromotive force.
BLDC	Brushless DC motor.
CPU	Central processing unit.
DC	Direct current.
DRM	Design reference manual.
I/O	Input/output interfaces between a computer system and the external world — a CPU reads an input to sense the level of an external signal and writes to an output to change the level of an external signal.
ISR	Interrupt Service Routine.
MCU	Microcontroller.
PI controller	Proportional-integral controller.
PWM	Pulse-width modulation.
SPI	Serial peripheral interface module.

5 References

These references are available on www.nxp.com:

1. *Set of General Math and Motor Control Functions for ARM Cortex M0+ Core*, (document CM0AMCLIBUG, CM0GDFLIBUG, CM0GMCLIBUG, CM0GFLIBUG, and CM0MLIBUG).

2. *Set of General Math and Motor Control Functions for ARM Cortex M4 Core*, (document CM4AMCLIBUG, CM4GDFLIBUG, CM4GMCLIBUG, CM4GFLIBUG, and CM4MLIBUG).

3. *Set of General Math and Motor Control Functions for ARM Cortex M7 Core*, (document CM7AMCLIBUG, CM7GDFLIBUG, CM7GMCLIBUG, CM7GFLIBUG, CM7MLIBUG).

4. *3-Phase BLDC Motor Sensorless Control using MC9S08AW60*, (document DRM086).

5. *3-Phase BLDC Motor Sensorless Control using MC9S08MP16*, (document DRM117).

6. *Sensorless BLDC Control using Kinetis KV*, (document AN5263).

Revision history

Table 2. Revision history

Revision number	Date	Substantive changes
0	01/2014	Initial release
1	03/2016	Updates for revision 1

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
www.freescale.com/support

Document Number: DRM144
Rev. 1
03/2016