Freescale Semiconductor
Application Note

Document Number: AN4376
Rev. 0, 10/2011

BLDC Motor Control with Hall Effect Sensors Using MQX on Kinetis

by: Ivan Lovas
Freescale Czech System Laboratories
Rožnov pod Radhoštem, Czech Republic

1 Introduction

This application demonstrates low power three-phase BLDC motor drive software. It is focused on a simple and easy-to-understand control approach for a BLDC using MQX in time-critical applications.

The control concept of the application is based on a speed closed-loop BLDC drive using Hall effect positional sensors. It serves as an example of a BLDC motor control system for low-voltage motor applications. The application runs on a Freescale Kinetis K60 microcontroller. The hardware is built on the Freescale Tower rapid prototyping system and contains the following modules:

- TWR-Elevator
- TWR-K60N512
- TWR-MC-LV3PH
- TWR-SER

Contents

There are two versions of the application software. One is with the MQX RTOS and the other is bare-metal. Both use the same source code for motor control. The MQX version contains a web server to demonstrate the benefits of an MQX-based solution.

Both applications can also be controlled by FreeMASTER.

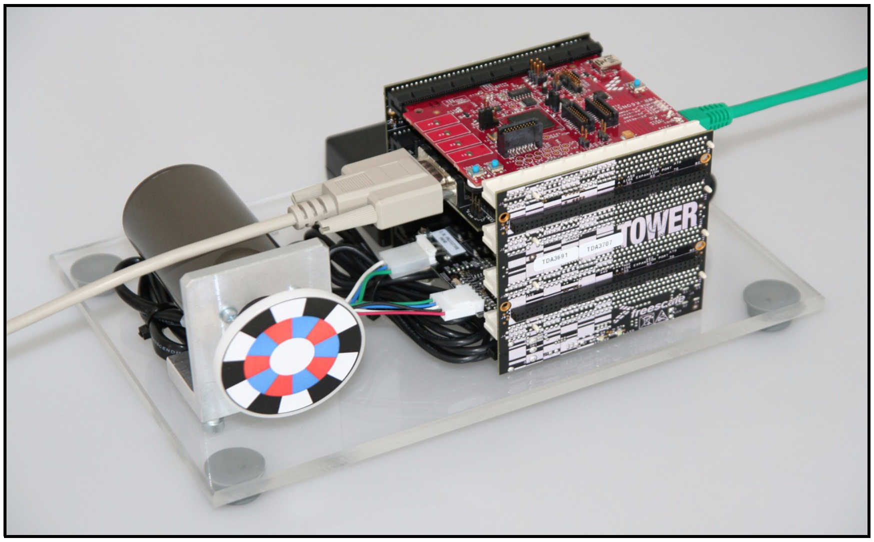

Figure 1. Demo of BLDC motor control using Kinetis

1.1 Application features

- MK60 Kinetis ARM Cortex-M4 microcontroller
- Hardware built on the Tower rapid prototyping system
- BLDC motor control using Hall effect sensors to determine position
- Speed closed-loop with speed measurement and adjustable parameters
- Adjustable speed ramp
- Versatility, thanks to fractional arithmetic
- MQX allows easy use with other applications and fast prototyping
- FreeMASTER software used for application control and monitoring
- Web page used for application control via ethernet
- Motor mode in both directions of rotation
- Minimum speed of 500 rpm
- Maximum speed of 4000 rpm
- Overvoltage, undervoltage, and overcurrent fault protection

1.2 Core features

The 32-bit Kinetis MCUs represent the most scalable portfolio of ARM® Cortex™-M4 MCUs in the industry. The first phase of the portfolio consists of five MCU families with over 200 pin-, peripheral-, and software-compatible devices with outstanding performance, memory, and feature scalability. Enabled by innovative 90nm Thin Film Storage flash technology with unique FlexMemory (configurable embedded

EEPROM), Kinetis features the latest low-power innovations and high-performance, high-precision mixed-signal capability. Kinetis MCUs are supported by a market-leading enablement bundle from Freescale and ARM third party ecosystem partners.

2 Motor control via MQX RTOS

2.1 What is MQX?

Freescale MQX software solutions offer a straightforward API with a modular architecture, making it simple to fine-tune custom applications and allowing scalability to fit most requirements. The combination of our market-proven Freescale MQX software solutions and silicon portfolio provides a streamlined and powerful platform by creating a comprehensive source for hardware, software, tools, and service needs.

2.2 When to use motor control via MQX RTOS

MQX is not a typical operating system for motor control applications. The MQX OS is suitable for large applications with advanced features, such as web control, USB, displays, and SDHC card reading, running on a dedicated device (usually one core). The primary strength of MQX is that it includes libraries for RTCS, Ethernet, USB communication, MFS file system, and many other applications.

2.3 How to implement motor control via MQX

The MQX RTOS is a complex system with dynamic allocations and POSIX scheduling. It has a system default tick duration of 5 ms. This is ideal for the majority of applications. However, this also means that the MQX task time resolution is more than 1000 times longer than needed when compared to the motor control requirements. Therefore, it is evident that the motor control process needs to be serviced with interrupts of a high priority.

These interrupts can be provided with standard MQX interrupt routines. The time duration from the interrupt request to the service routine's execution is usually units of a microsecond (depending on the CPU version and clock speed). And, if necessary, the motor control algorithms can be implemented using the kernel interrupts. The kernel interrupts are natural CPU interrupts with no MQX overhead and minimal execution duration. The disadvantage of the kernel interrupt is that no MQX functionalities such as events or semaphores are supported.

Details are described in the MQX documentation — see Section 8, "Reference," item 3.

3 BLDC motor theory

3.1 BLDC basic information

A BLDC motor is a rotating electric machine where the stator is a classic three-phase stator like that of an induction motor, and the rotor has surface-mounted permanent magnets (see Figure 2).

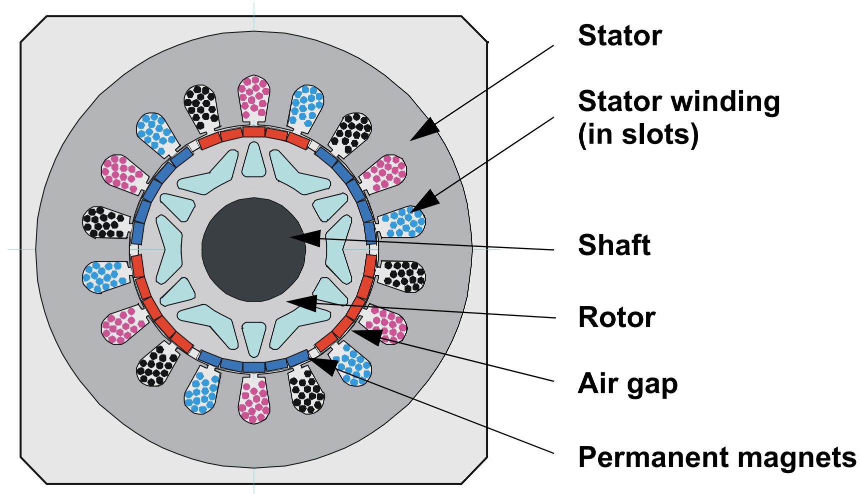

Figure 2. BLDC motor — cross-section

In this respect, the BLDC motor is equivalent to a reversed DC commutator motor, in which the magnet rotates while the conductors remain stationary. In the DC commutator motor, the current polarity is altered by the commutator and brushes. On the other hand, in the brushless DC motor, the polarity reversal is performed by power transistors switching in synchronization with the rotor position. Therefore, BLDC motors often incorporate either internal or external position sensors to sense the actual rotor position, or the position can be detected without sensors.

3.2 Digital control of a BLDC motor

The BLDC motor is driven by rectangular voltage strokes coupled with the given rotor position (see Figure 3). The generated stator flux interacts with the rotor flux, which is generated by a rotor magnet, defining the torque and thus the speed of the motor. The voltage strokes must be properly applied to the two phases of the three-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 for proper operation.

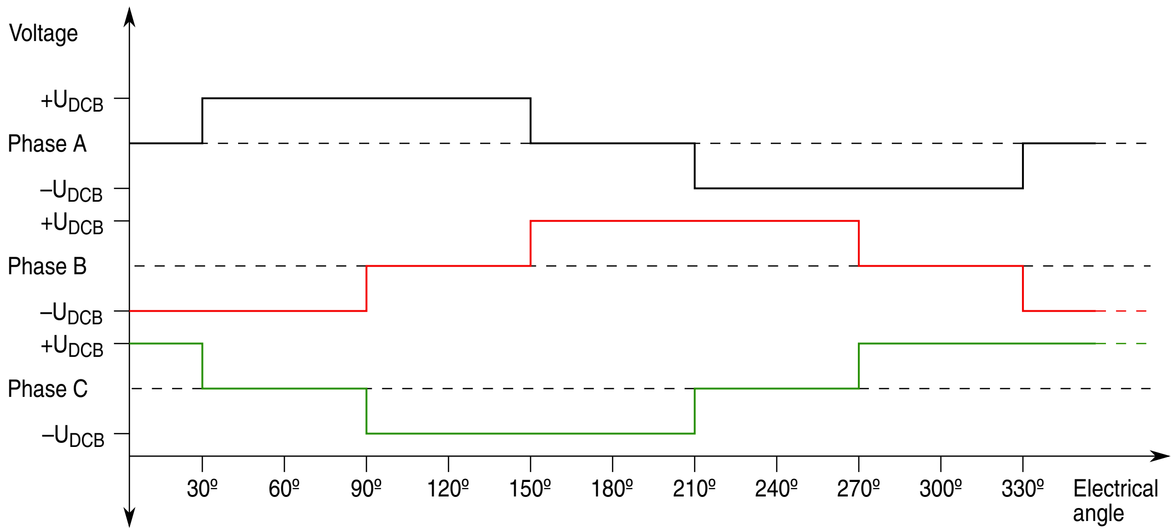

Figure 3. Voltage strokes applied to three-phase BLDC motor

For the common three-phase BLDC motor a standard three-phase power stage is used, as shown in Figure 4. The power stage utilizes six power transistors.

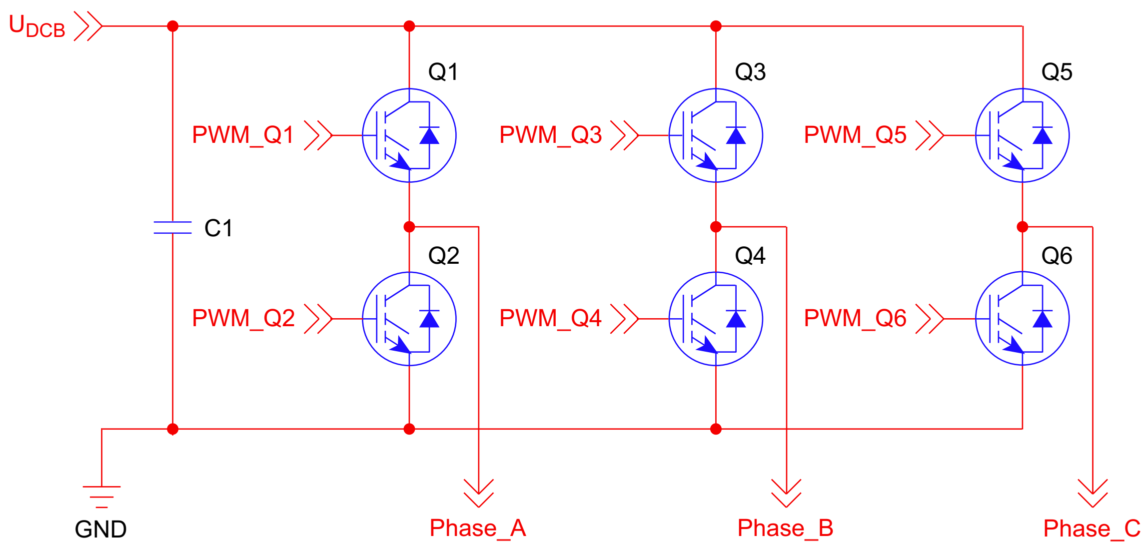

Figure 4. Three-phase BLDC power stage

In both modes, the three-phase power stage energizes two motor phases concurrently. The third phase is unpowered (see Figure 3). Thus, we get six possible voltage vectors that are applied to the BLDC motor using a PWM technique. There are two basic types of power transistor switching: complementary switching and independent switching.

3.3 Complementary versus independent switching

With complementary switching, two transistors are switched on when the phase of the BLDC motor is connected to the power supply. But there is a difference during freewheeling. With independent switching, all the transistors are switched off and the current continues to flow in the same direction through freewheeling diodes until it falls to zero. Contrary to this, with complementary switching, the complementary transistors are switched on during freewheeling. Thus the current may be able to flow in the opposite direction. Figure 5 depicts the complementary switching.

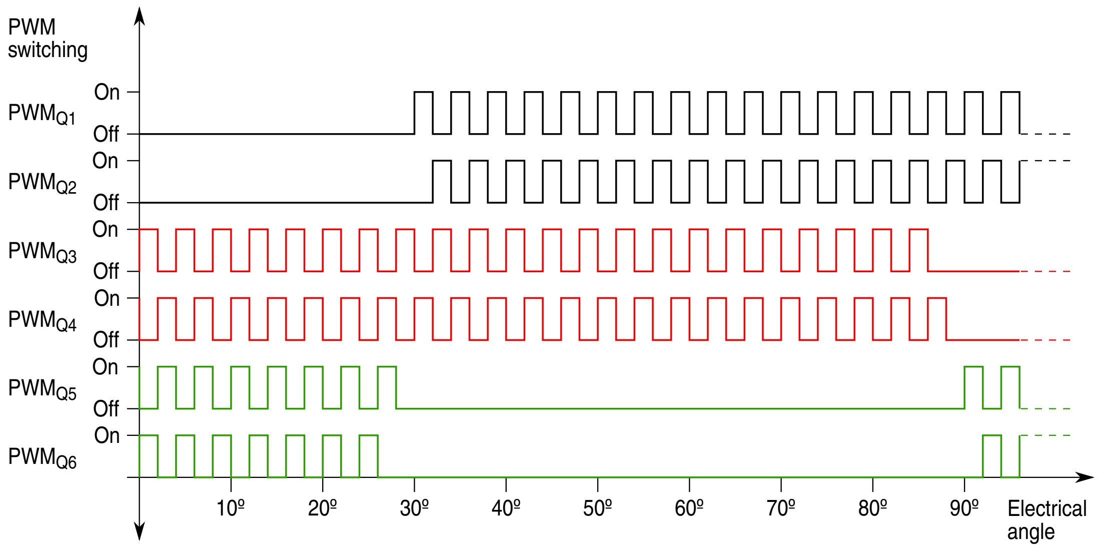

Figure 5. Complementary switching of power transistors

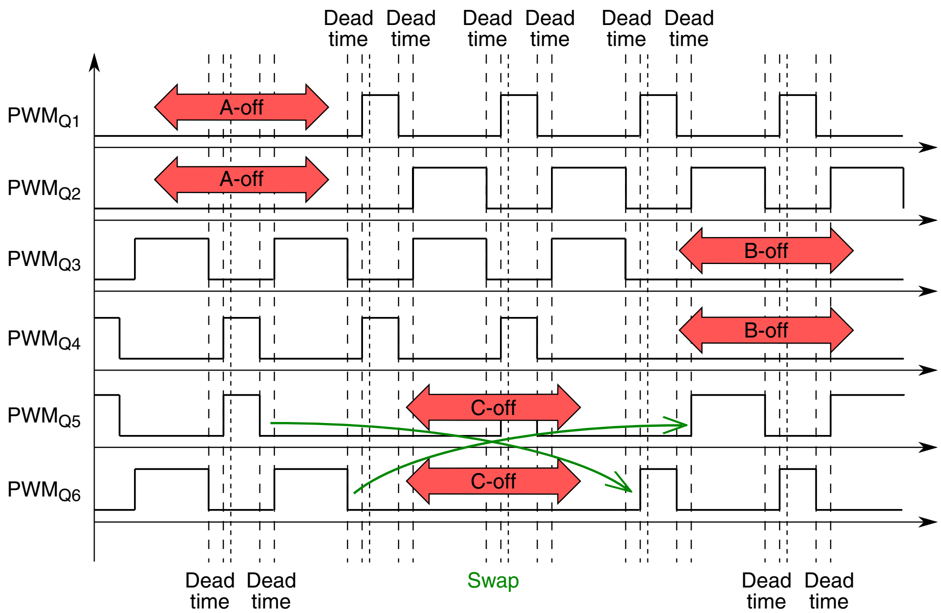

Figure 6. Bipolar PWM switching — detail

Details of the technique are shown in Figure 6. In this case we can see the characteristic of the bipolar four-quadrant (complementary) switching. The bipolar switching requires that the top and bottom switch PWM signals need to be swapped. Another important detail is the introduction of dead time insertion in the complementary top and bottom signals. This dead time insertion is typical for all four-quadrant power stage operations. Four-quadrant operation is enabled by the complementary operation of the top and bottom switches (the bottom switch of one phase is almost the negative of the top switch).

This requires the insertion of dead time, since the switching transient will cause a DC-bus short circuit with fatal power stage damage.

Bipolar PWM switching is not as popular as unipolar switching due to worse electromagnetic emission from the motor. This is because the PWM ripple is twice that of the DC-bus voltage. On the other hand, this switching is better for sensorless rotor position sensing.

Details are described in the documentation — see Section 8, “Reference,” items 1 and 2.

3.4 Commutation

Commutation provides the creation of a rotation field. As was explained, for the proper operation of a BLDC motor, it is necessary to keep the angle between the stator and rotor flux close to 90°. With six-step control, we get a total of six possible stator flux vectors. The stator flux vector must be changed at a certain rotor position. The rotor position is usually sensed by Hall effect sensors. The Hall sensors generate three signals that also comprise six states. Each of the Hall sensors’ states corresponds to a certain stator flux vector. All of the Hall sensor states with corresponding stator flux vectors are illustrated in Figure 7**.** The same figure is illustrated in Table 1.

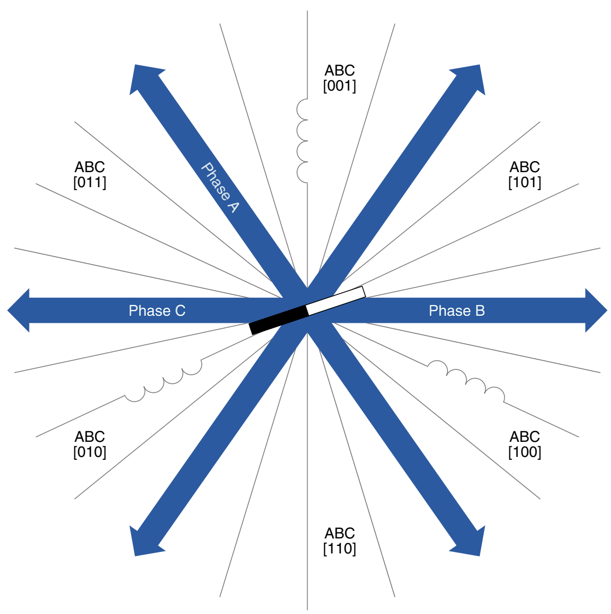

Figure 7. Stator flux vectors with six-step control

As can be seen, using a six-step control technique, there is no possibility of keeping the angle between the rotor flux and the stator flux precisely at 90°. The real angle varies from 60° to 120°.

The commutation is repeated per each sixty electrical degrees. The commutation event is critical for its angular (time) accuracy. Any deviation causes torque ripples and therefore speed variation.

Table 1. Commutation sequence for clockwise rotation

Hall sensor A	Hall sensor B	Hall sensor C	Phase A	Phase B	Phase C
0	1	1	$-V_{DCB}$	$+V_{DCB}$	NC
0	1	0	NC	$+V_{DCB}$	$-V_{DCB}$
1	1	0	$+V_{DCB}$	NC	$-V_{DCB}$
1	0	0	$+V_{DCB}$	$-V_{DCB}$	NC
1	0	1	NC	$-V_{DCB}$	$+V_{DCB}$
0	0	1	$-V_{DCB}$	NC	$+V_{DCB}$

3.5 Speed control

Commutation ensures proper rotor rotation of the BLDC motor, while the motor speed depends only on the amplitude of the applied voltage. The amplitude of the applied voltage is adjusted using the PWM technique. The required speed is controlled by a speed controller, which is implemented as a conventional proportional-integral (PI) controller. The difference between the actual and required speeds is input to the PI controller which then, based on this difference, controls the duty cycle of the PWM pulses that correspond to the voltage amplitude required to maintain the correct speed.

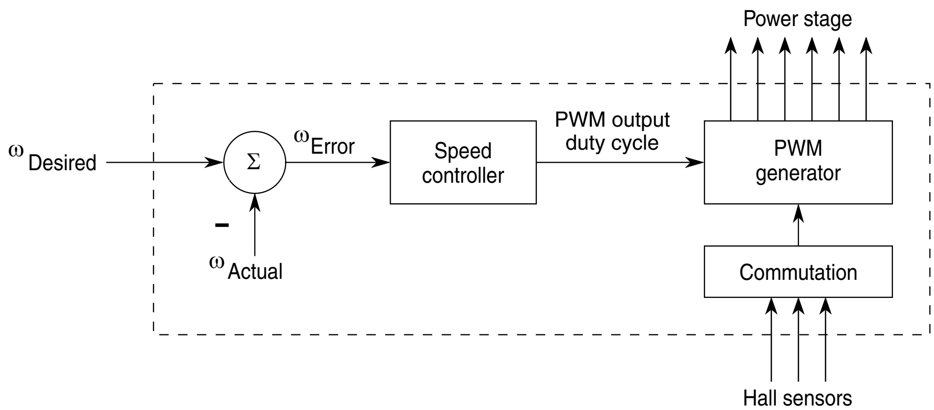

Figure 8. Speed controller

The speed controller calculates the PI algorithm given in the equation below:

$$u(t) = K_c\left[e(t) + \frac{1}{T_I}\int_0^t e(\tau)d\tau\right] \qquad \textit{Eqn. 1}$$

After transforming the equation into the discrete time domain using an integral approximation with the Backward Euler method, we get the following equations for the numerical PI controller calculation:

$$u(k) = u_P(k) + u_I(k) \qquad \textit{Eqn. 2}$$

$$u_P(k) = K_c \cdot e(k) \qquad \textit{Eqn. 3}$$

$$u_I(k) = u_I(k-1) + K_c\frac{T}{T_I} \cdot e(k)$$ Eqn. 4

where:

e(k)	=	Input error in step k
w(k)	=	Desired value in step k
m(k)	=	Measured value in step k
u(k)	=	Controller output in step k
$u_p(k)$	=	Proportional output portion in step k
$u_I(k)$	=	Integral output portion in step k
$u_I(k-1)$	=	Integral output portion in step k–1
T_I	=	Integral time constant
T	=	Sampling time
K_c	=	Controller gain

4 System description

4.1 Application outline

The system is designed to drive a three-phase BLDC motor. The application meets the following performance specifications:

- Voltage control of a BLDC motor using Hall effect sensors
- Tower System solution with the TWR-K60N512 board
- Power supply voltage +24 V_{DC}
- Control technique incorporates:
  - — Position sensing using Hall effect sensor signals
  - — Voltage BLDC motor control with a speed closed-loop
  - — Speed measurement based on one Hall effect sensor
  - — Two directions of rotation
  - — Ability to start from any rotor position
  - — Pre-charging of MOSFET pre-driver bootstraps before each motor start
  - — Minimal speed of 500 RPM (depending on the motor used)
  - — Maximal speed of 4000 RPM (depending on the motor used)
- FreeMASTER interface (input speed, measured speed, speed error, ramp parameters, overcurrent LED indication)
- Ethernet terminal (input speed, ethernet status)
- Fault protection:

— DC-bus overcurrent fault protection (hardware)
— Power supply reverse polarity protection circuitry (hardware)

4.2 Application description

The MCU PK60N512VMD100 runs the main control algorithm. According to the user interface and feedback signals, it generates three-phase PWM output signals for a three-phase inverter.

The whole application runs in interrupts for ease of use under MQX. When the required speed is other than zero, the application enables an interrupt from the Hall sensor and a first time call of the Hall interrupt routine is forced. Each new edge of the Hall sensor calls the interrupt routine automatically. In this routine, the signal from the Hall sensor is scanned, and corresponding PWM channels are swapped and masked. This process is called commutation. The Hall sensor scan is independent of speed control. The speed control loop is called periodically by a PIT 0. In this periodic loop, there is a speed ramp as well as the application state machine. In the main routine there is only an endless loop with a FreeMASTER poll function to control the application.

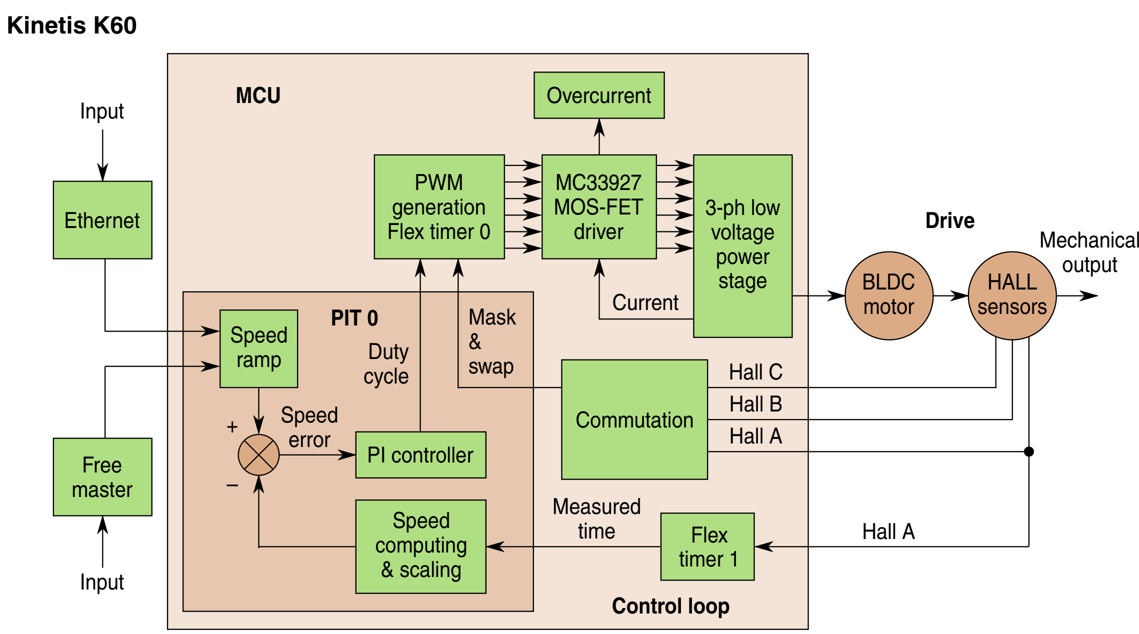

Figure 9. Application block diagram

4.3 Peripherals

NOTE

For the proper function of this application, the following peripherals must be used. It is not allowed to use these peripherals for any other purpose.

1. FTM0
   — Used to generate a PWM signal
   — Run in combine mode

- — Switching frequency is 19.2 kHz on a 48 MHz core clock
- — Dead time 1 µs

2. FTM1
   - — Used for speed measurement
   - — Run in input capture mode
   - — Prescaler is 128
   - — Modulo is 0xFFFF
   - — Period of overflow is 175 ms
3. PIT 0
   - — Used for periodic call of the speed control loop and application state machine
   - — Period of interrupt is 10 ms on a 48 MHz core clock
4. PORT A
   - — Used for Hall effect sensor interrupt
   - — If any other signal is applied on this port, the interrupt will be called on each edge of the signal and the program will not work properly
5. PORT D
   - — Used for Hall effect sensor interrupt
   - — If any other signal is applied on this port, the interrupt will be called on each edge of the signal and the program will not work properly
6. PTE26
   - — Used for the emergency stop button
7. PTA27
   - — Used to read the overcurrent pin on the MC33937 MOS-FET predriver
8. PTA10
   - — Used to indicate the first level of overcurrent on the MC33937 MOS-FET predriver
9. SPI 2
   - — Used for communication with the MC33937 MOS-FET predriver

4.4 Data flow diagram

The requirements of the drive dictate that software takes some values from the user interface and sensors, processes them and generates three-phase PWM signals for motor control. The control algorithm of the closed-loop BLDC drive is described in Figure 10.

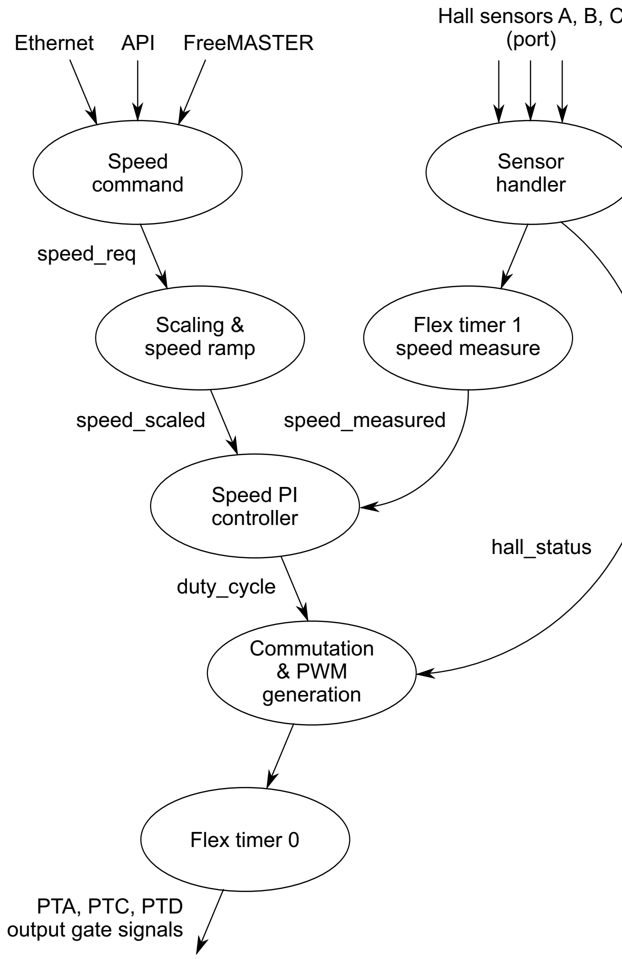

Figure 10. Data flow

The data flow diagram consists of the processes described in the following sub-sections:

- Speed command

  Every speed change command causes a change of value in the speed_req variable. There are three possible ways to enter the speed command:

  — Via Ethernet interface (MQX version only)

  — Via FreeMASTER communication software

  — Via API

- Scaling and speed ramp

  Provides the scale change to frac32 and the speed ramp; see Section 5.3, "Speed scaling," and Section 4.6, "Speed ramp." It runs in the PIT interrupt handler.

- Speed PI controller

  This is used to the count the difference between the real and required speeds and compensate the duty cycle of the PWM module accordingly; see Section 4.7, "PI controller."

- Commutation and PWM generation

  This is used to create a rotating field based on signals from the Hall sensors.

- Sensor handler
  This is used to generate the commutation vector.
- Speed measurement
  See Section 4.5, "Speed and position measurement."

4.5 Speed and position measurement

The actual motor speed is calculated based on the revolution period (*time_measured*) and compared with the *speed_req*, provided by the user. The speed command is then processed by means of the speed-ramp algorithm. The comparison between the actual speed command obtained from the ramp algorithm output and the *speed_measured* generates a *speed_error.*

The revolution period is scanned via Hall sensor A and flex timer 1, which is configured in capture mode.

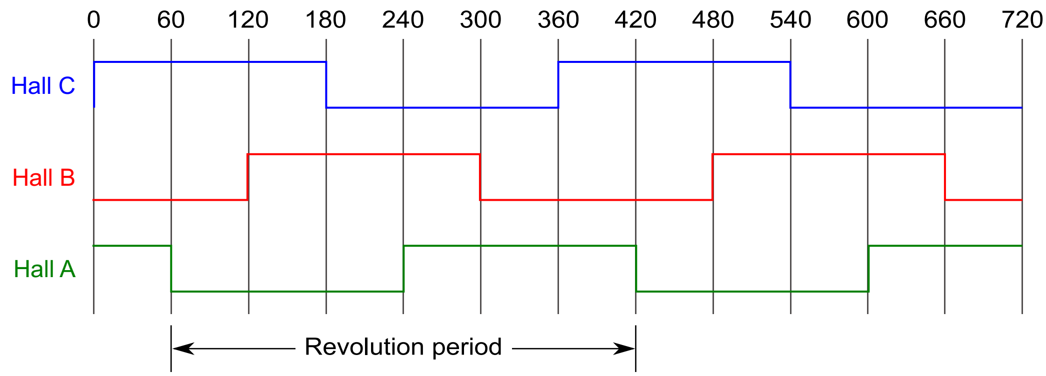

Figure 11. Hall sensor signals for speed measurement

4.6 Speed ramp

Since the overall application is a system with large inertia, it is necessary to profile the speed command when applying — otherwise it could overload the system possibilities. One approach is ramp generation. This ramp implements steps between the actual speed and the speed command.

In this application, a ramp is used from library GFLIB. This ramp needs three parameters:

- s32RampUp
- s32RampDown
- s32State

The first two parameters are the up and down angles of the ramp. The last parameter is the required speed. All three parameters are in 32-bit fractional data format.

It is possible to enter the first two parameter in *variables_init.h* as integers s in [r/s^2], but they are recalculated back to frac32 during application initialization. The parameters are named *speed_ramp_down* and *speed_ramp_up*.

The speed ramp is called in the PIT interrupt handler. The ramp execution is highly dependent on the PIT period. The default value is 10 ms. If the period of PIT is changed, it is necessary to change the PIT period in the MAX_SCALED_SPEED_INV macro. This macro is used to recompute *speed_ramp_down* and *speed_ramp_up* from int to frac32.

The second possibility for entering the speed ramp parameters is use to FreeMASTER. In FreeMASTER data can be entered as an integer in [r/s^2] at program runtime.

Details are described in the documentation, see Section 8, "Reference," item 4.

4.7 PI controller

The speed PI control algorithm processes the *speed_error* between *speed_req* and *speed_measured*. The PI controller output is passed to the PWM generator as a newly corrected value of the applied motor voltage.

The PI controller routine is calculated in the interrupt routine of the PIT device, in PIT0_isr, which is called every 10 ms. This interrupt is disabled when the motor is stopped, so PI is disabled too. The integral part of the PI controller is disabled at low speeds (under 499 RPM), because in that case measurement of speed is not accurate and the PI controller can be unstable. To determine when it should be disabled, in the program there are two macros: MIN_CW_SPEED_32 and MIN_CWW_SPEED_32.

The input of the PI controller is the output from the ramp algorithm *speed_scaled*, and the other input is the actual *speed_measured*. The other two inputs are pointers to the structure of the PI controller parameters *trMyPI*. All these parameters are used by the PI controller function GFLIB_ControllerPIp.

The output of this function is *s32Output*. It is scaled to the PWM scale as *delta_duty* and is added to *half_duty*. The result of this process is *duty_cycle* which is loaded into the Flex Timer registers.

The PI speed controller parameters must be configured each time the speed scale is changed or when the motor is changed.

Details are described in the documentation, see Section 8, "Reference," item 4.

5 Software implementation

5.1 Implementation of functions in software

The complete motor control algorithm is driven by interrupts. The main function is used only for MCU and application initialization; see Figure 12. When initialization terminates, the program goes into an endless loop, or another application handler (web server, USB, FreeMASTER, etc.).

NOTE

This arrangement is very important for proper functioning of another application. When this arrangement is used, it is easy to implement via MQX or bare metal.

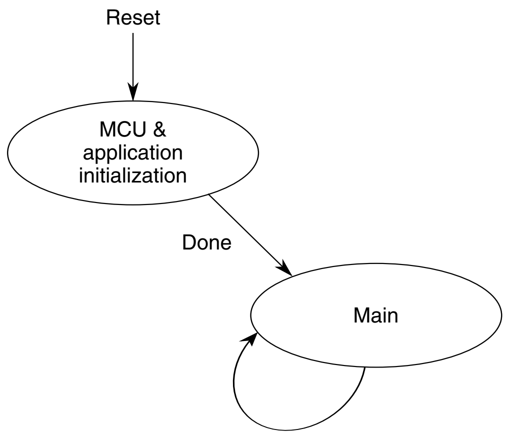

Figure 12. Implementing main function in software

For proper functioning of this motor application, four interrupts are needed.

The interrupt handler provides the following services:

- Overflow Interrupt Handler

  Used for motor stop detection and for speed measurement. The overflow handler is used to reset the speed ramp.
- Input Capture Interrupt Handler

  Reads the time between the two Hall sensor edges (a basic part of the Process Speed Sensor).
- Periodic interrupt

  Used for periodic calls of the speed controller, speed ramp, and application state machine. The PIT interrupt is disabled when the motor is stopped.
- Hall sensor interrupt

  Used to scan the status of the Hall sensors and for the commutation process. The commutation process generates the proper commutation pattern on six gate signal outputs, while the PWM generation process generates the appropriate PWM signal for the selected gate outputs. The commutation pattern is selected from the commutation table using the three-bit commutation vector “hall_status.” The commutation pattern is then loaded into the registers of the microcontroller.

In Figure 13 it can be seen how the interrupts are implemented in software.

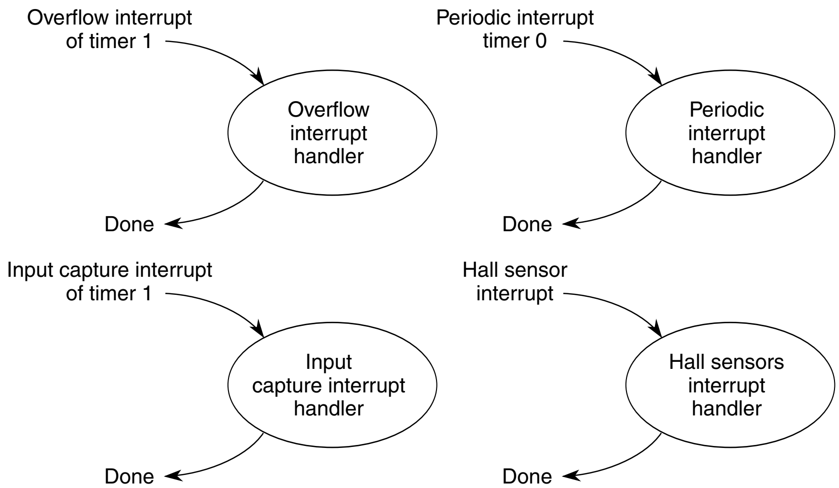

Figure 13. Interrupts in motor control application

5.2 Interrupt installation

The difference between the bare-metal version and the MQX version is only in the method used to install interrupts. The method used depends on whether MQX or Bare_Metal is predefined in the IAR project options (Options / C/C++Compiler / Preprocessor / Defined symbols).

In the bare-metal version, we can install an interrupt directly. You can simply allow an interrupt by setting the right bit in the NVICISER register. Priority of the interrupt can be configured using the register NVIC_IP. For a better understanding, see the following example.

Installation of interrupts:

```
NVICICPR2  = 0x4800010;   // clear a possible pending interrupt first
NVICISER2  = 0x4800010;   // enable interrupts
NVICICPR1 |= 0x80000000;  // clear a possible pending interrupt first
NVICISER1 |= 0x80000000;  // enable interrupts
```

Set priority of interrupts:

```
NVIC_IP(63) = 0x40;    // set priority for FTM1
NVIC_IP(68) = 0x50;    // set priority for PIT0
NVIC_IP(87) = 0x40;    // set priority for the Hall sensors
NVIC_IP(90) = 0x40;    // set priority for the Hall sensors
```

If the installation of interrupts in MQX were the same as in the bare-metal version, MQX would not work correctly. The MQX's own interrupts are quite slow for motor control applications. The best solution for high-speed applications is kernel interrupts.The kernel interrupts are natural CPU interrupts with no MQX overhead and minimal execution duration. The disadvantage of the kernel interrupt is that no MQX functionalities, such as events or semaphores, are supported. For a better understanding of how to use kernel interrupts, see the following example.

Installation of interrupts:

```
_int_install_kernel_isr(INT_PIT0,PIT0_isr);
_int_install_kernel_isr(INT_FTM1,FTM1_isr);
_int_install_kernel_isr(INT_PORTA,Hall_Status_isr);
_int_install_kernel_isr(INT_PORTD,Hall_Status_isr);
```

Set priority of interrupts:

```
_bsp_int_init((IRQInterruptIndex)INT_PIT0, 1, 0, 1);
_bsp_int_init((IRQInterruptIndex)INT_FTM1, 1, 0, 1);
_bsp_int_init((IRQInterruptIndex)INT_PORTA, 1, 0, 1);
_bsp_int_init((IRQInterruptIndex)INT_PORTD, 1, 0, 1);
```

5.3 Speed scaling

This application uses a fractional representation for all real quantities, except time. The N-bit signed fractional format is represented using the1.[N-1] format (1 sign bit, N-1 fractional bits). Signed fractional numbers (SF) lie in the following range:

$$-1.0 \le \mathrm{SF} \le +1.0 - 2^{-[N-1]}$$

For words and long-word signed fractions, the most negative number that can be represented is –1.0, whose internal representation is $8000 and $80000000, respectively. The most positive word is $7FFF or $1.0 - 2^{-15}$, and the most positive long-word is $7FFFFFFF or $1.0 - 2^{-31}$. The following equation shows the relationship between a real and a fractional representation:

Eqn. 5

$$\text{Fractional Value} = \frac{\text{Real Value}}{\text{Real Quatity Range}}$$

In the case of the speed scale:

Fractional Value = Fractional representation of speed quantities [–]

Real Value = Real speed quantities in physical units [rpm]

Real Quantity Range = Max. defined speed used for scaling in physical units [rpm]

In this application the scale constant is calculated by this equation:

Eqn. 6

$$\text{Scale Constant} = \frac{1}{\text{MAX\_SCALED\_SPEED}} x \frac{30}{\text{PP}\frac{\text{TPM\_P}}{\text{TPM\_C}}} = \frac{1}{5000} x \frac{30}{2\frac{128}{48000000}} = 1125$$

In this application the SCALE_CONST macro is used for computing the scale:

SCALE_CONST (tFrac32)((30 / (PP*(TPM_P/TPM_C))) / MAX_SCALED_SPEED)

PP — Number of pole pairs of motor used. Default value is 2.

TPM_C — Input clock of the timer in Hz. Default value is 48e6.

TPM_P — Prescaler of the timer input clock. Default value is 128.

MAX_SCALED_SPEED — Maximal speed with reserve. Default value is 5000.

As you can see from Equation 6, this scale constant is calculated for only half a period. This is done for better use of the timer. *Time_measured* is shifted to the right (divided by two) to contain only positive numbers. The timer modulo is set to 0xFFFF and the prescaler is set to 128. In this case, the overflow period of the timer is 174.8 ms, so we can detect a minimal speed of 171.6 rpm.

Finally, the *speed_measured* is calculated by this equation:

Eqn. 7

$$\text{measured\_speed} = \frac{1}{\frac{\text{revolution\_period}}{2}} \cdot \text{scaling\_constant}$$

speed_measured = F32DivF32F16((SCALE_CONST), (time_measured >> 1));

Details are described in the documentation; see Section 8, "Reference," item 6.

5.4 Changing the speed scale

The scaling constant is calculated in the macro, so it is calculated only when you compile the program. You can change parameters of the scaling constant in the file *variables_init.h* according to the motor used, but you must recompile the program.

These parameters can be set in the file *variables_init.h*:

PP — Number of pole pairs of the motor used. This number describes the ratio between the electrical and mechanical revolutions of the motor. Default value is two. This parameter can be found in the data sheet of the motor.

TPM_C — Input clock of the timer in Hz. Default value is 48e6.

TPM_P — Prescaler of the timer input clock. Default value is 128.

MAX_SCALED_SPEED — Maximal speed with reserve. Reserve is 20%. This parameter has to be greater than the nominal speed of the motor.

NOTE

PI speed controller parameters must be configured each time the speed scale is changed or when the motor is changed.

5.5 Integration of driver into other applications

Both versions of the motor control driver can be part of other applications. If the application is implemented in another application, it is necessary to define MQX or Bare_Metal in the project settings. The API is defined by the following three functions:

1. `void Set_speed(signed short ,int)`

   The first parameter is the input speed in signed short data format. Input value is in RPM.

The second parameter is the number of the motor which receives the command. This demo has only one motor, so enter 1.

Return: void

```
/***********************************************************
 * Function Name: Set_speed
 * Parameters: required speed, number of motor
 * Return: Measured speed
 * Description: Function to enter the req speed for each motor
 ***********************************************************/
void Set_speed(signed short speed_input, int motor_number)
{ speed_req = INT16TOF32((speed_input*SPEED_TO_RPM_SCALE)); }
```

2. `unsigned char Get_status(void)`

   Return: Status of the application

   0 — IDLE

   1 — STOP

   2 — RUNNING

```
/******************************************************
 * Function Name: Get_speed
 * Parameters: number of the motor
 * Return: Measured speed
 * Description: Function to get the speed of each motor
 ******************************************************/
signed short Get_speed(int motor_number)
{
  signed long speed_temp;
  speed_temp = (speed_measured * MAX_SCALED_SPEED);
  return  (speed_temp >> 15);
}
```

3. `signed short Get_speed(int)`

   The first parameter is the number of the motor which receives the command. This demo has only one motor, so enter 1.

   Return: Measured speed in signed short data format. The value is in RPM.

```
/**************************************************
 * Function Name: Get_status
 * Parameters: none
 * Return: Application status
 **************************************************/
unsigned char Get_status(void)
{ return App_state; }
```

6 Application configuration

Details about the configuration of the application are described in the documentation; see Section 8, "Reference," item 5.

7 Definitions and acronyms

AN	Application note
API	Application interface
BLDC	Brushless DC motor
DC	Direct current
MQX™	Freescale MQX real-time operating system
Tick	Operating system time unit (also reflects the minimal time resolution)
POSIX	Portable operating system interface, produced by IEEE and standardized by ANSI and ISO. MQX conforms to POSIX.4 (real-time extensions), and POSIX.4a (threads extensions).
RTOS	Real-time operating system
RTCS	Embedded Internet stack provides IP networking for the MQX platform. RTCS is provided with a rich assortment of TCP/IP networking application protocols and uses the MQX RTOS drivers for Ethernet or serial connectivity.

8 Reference

1. Freescale document DRM022, 3-*Phase BLDC Drive Control with Hall Sensors*, Jiri Ryba and Petr Stekl, 2003
2. Freescale document DRM117, *3-Phase Sensorless BLDC Motor Control Using MC9S08MP16,* Libor Prokop, 2009
3. Freescale document AN4254, *Motor Control Under the Freescale MQX Operating System,* Libor Prokop, 2011
4. Freescale document MCLIBCORETXM4UG, *Set of General Math and Motor Control Functions for Cortex M4 Core,* Jaroslav Musil and Pavel Rech, 2011
5. Freescale document BLDCK60UG, *3-Phase BLDC Motor Control on Kinetis,* Ivan Lovas, 2011
6. Freescale document AN1930, *3-Phase AC Induction Motor Vector Control Using a 56F80x, 56F8100 or 56F8300 Device,* Jaroslav Lepka and Petr Stekl, Freescale Semiconductors, 2005

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4376
Rev. 0
10/2011