Freescale Semiconductor
Application Note

Document Number: AN4813
Rev 0, 10/2013

Encoder Position and Speed Sensing Utilizing the Quad Timer on the MC56F827xx DSCs

by: Libor Prokop

1 Introduction

Incremental encoders are used by various motor control applications for rotor position and speed sensing; so the decoding of incremental encoder signals is an essential task for the controller devices. The Freescale Digital Signal Controllers (DSC), dedicated to motor control are equipped with Quad Timer TMR module(s). This timer supports incremental encoder signal decoding.

This application note explains and shows how to set and use the Quad Timer TMR module for position and speed sensing to get a high resolution over a wide speed range.

The MC56F827xx family is a new Freescale DSC dedicated to motor control. This application note focuses on the encoder implementation on MC56F827xx Freescale DSC. However the solution, with more or less changes, can be used on any Freescale device with a Quad Timer TMR module.

Contents

2 Digital signal controllers

One suitable DSC for a motor control application is MC56F827xx. The implementation of the quadrature encoder can benefit from the following processor features:

- Core and peripheral clock 50 MHz (Core clock can be set to 100 MHz in the Fast mode)
- Quad Timer TMR with four 16-bit counter/timer groups
- Input signal multiplexing (SIM_GPS registers)
- Two crossbar units to interconnect signals between the peripherals

3 Quad timers and encoder signal detection system

The quadrature encoder Phase A and Phase B signals are depicted in Figure 1. The task of the motor control applications is to get the position and speed from these signals. The encoder signal detection described in this application note utilizes two sub-modules of the Quad Timer module.

The block diagram of position and speed detection of the quadrature encoder signals using the Quad Timer is shown in Figure 1.

Because of the flexibility of the Quad Timer module, any of the four timer sub-modules can be used for rotor position detection from the quadrature encoder signals. Likewise, any of the sub-modules can be used for any timer input edge time capturing. Each of the Quad Timer sub-modules has one primary input and one secondary input. The inputs can be multiplexed using the GPIO Peripheral Select register (GPS) of the SIM module (SIM_GPS) and the Crossbar module XBARA. However in the final example, the sub-modules TMR 0 and TMR 1 are chosen.

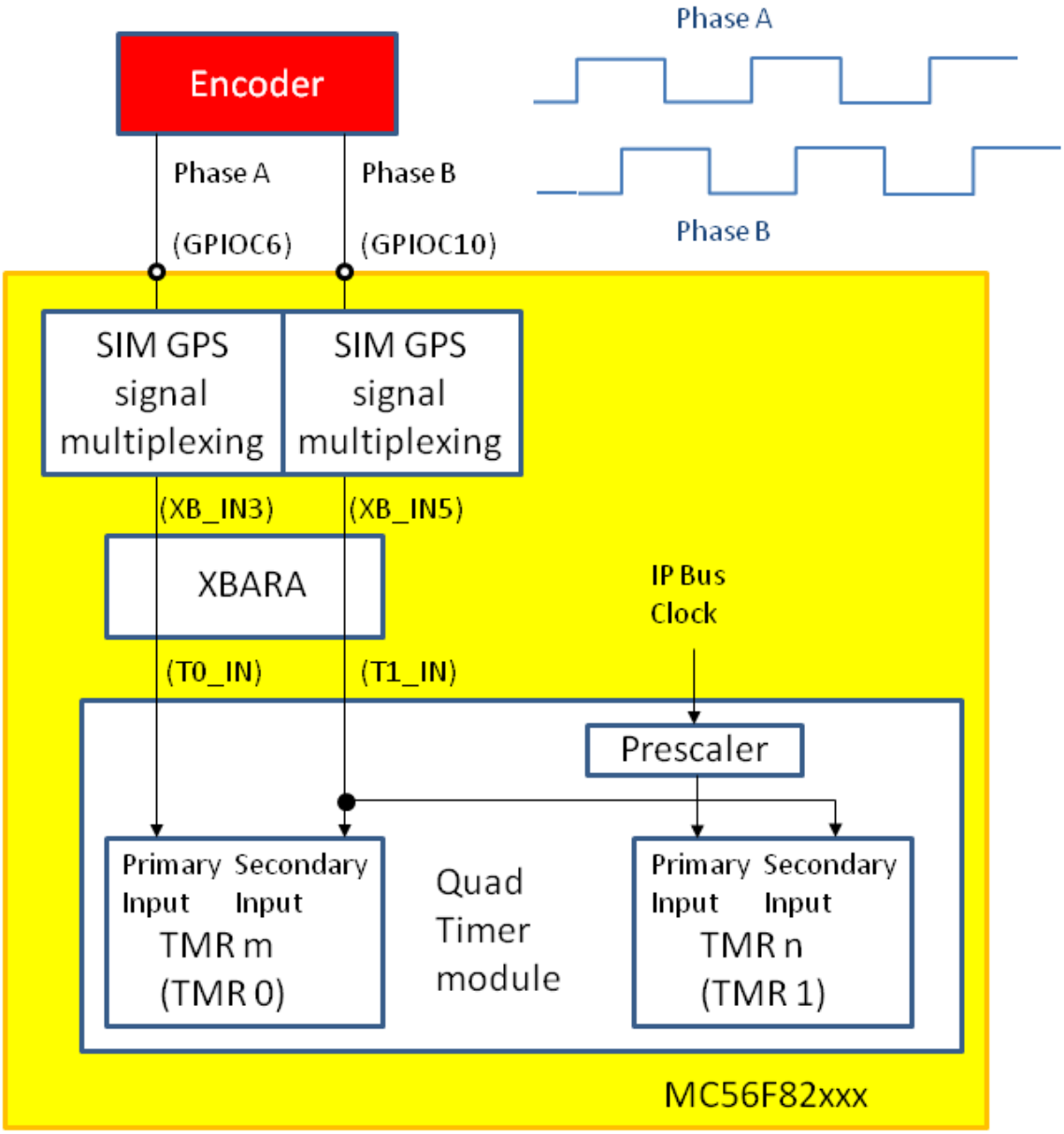

Figure 1. Quad Timer and encoder signal detection system

4 Position detection using encoder

The most important task when utilizing an encoder is to detect the relative rotor position from the quadrature signals.

Because of the flexibility of the Quad Timer module, any of the four timer sub-modules can be used for rotor position detection.

Now, mark the Timer m sub-module (where m can be 0, 1, 2 or 3) for position detection as shown in this figure.

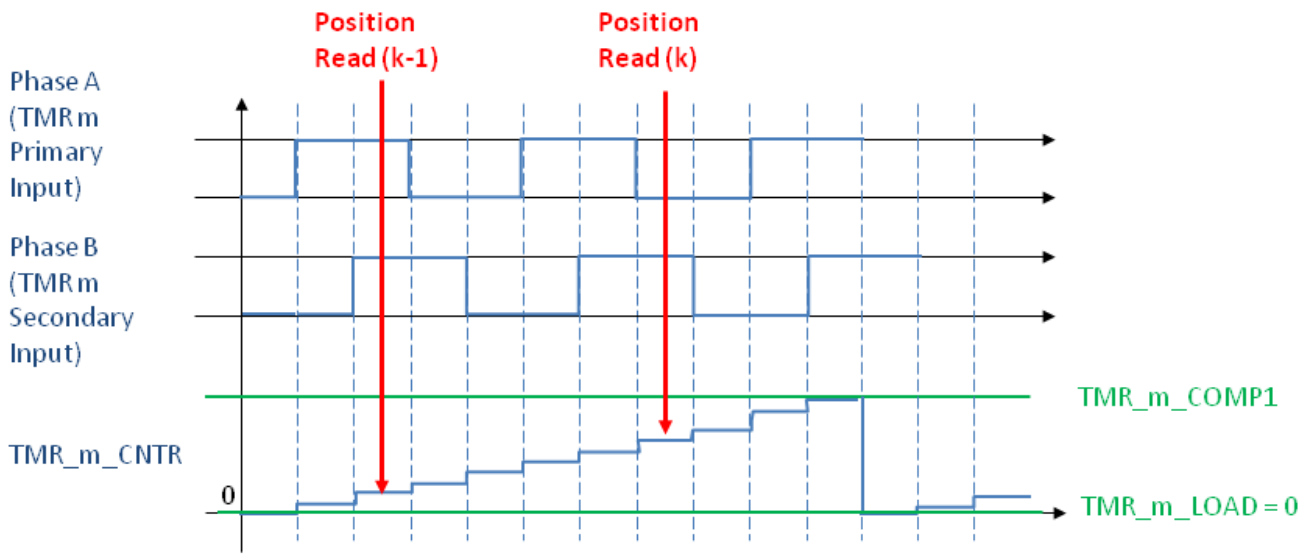

Figure 2. Position up counting using TMR m submodule

The encoder signals can be simply detected using the Quadrature count mode of the Quad Timer module. This mode uses primary and secondary sources connected to encoder Phase A and Phase B signals respectively. The Quadrature count mode is set in Timer Channel Control register (TMR_m_CTRL), with the following fields of this register:

Register fields	Value	Description
TMR_m_CTRL[CM]	100	Determines the Quadrature count mode. The Timer Channel Counter register (TMR_m_CNTR) is updated according to the primary and secondary input quadrature encoder signals.
TMR_m_CTRL[PCS]	0000	The primary timer source is timer input 0 (encoder Phase A).
TMR_m_CTRL[SCS]	01	The secondary timer source is timer input 1 (encoder Phase B).

TMR_m_CTRL[LENGTH]	1	Determines that the counter TMR_m_CNTR will be reloaded at Timer Channel Compare Register 1 (TMR_m_COMP1).

The syntax is as follows:

```
TMR_m_CTRL = TMR_m_CTRL_CM_2|TMR_m_CTRL_SCS_0|TMR_m_CTRL_LENGTH;
```

Set TMR_m_COMP1 = (Number of the encoder edges per one revolution - 1), as shown in this code line.

```
TMR_m_COMP1 = (ENC_NU_EDGES_REV-1);
```

In this way the counter counts from 0 up to one encoder revolution (TMR_m_COMP1) and then is reset to 0, defined by the initialization of Timer Channel Load register (TMR_m_LOAD) as per this code line:

```
TMR_m_LOAD = 0;
```

The functionality is displayed in Figure 2.

In the case of a counter rotation, the encoder pulses are processed in the Quadrature count mode as a down counting TMR_m_CNTR. Here, set TMR_m_COMP1 = (a negative number of encoder edges per one revolution + 1), using this code line.

```
TMR_m_COMP2 = -(ENC_NU_EDGES_REV-1);
```

This functionality can be seen in Figure 3.

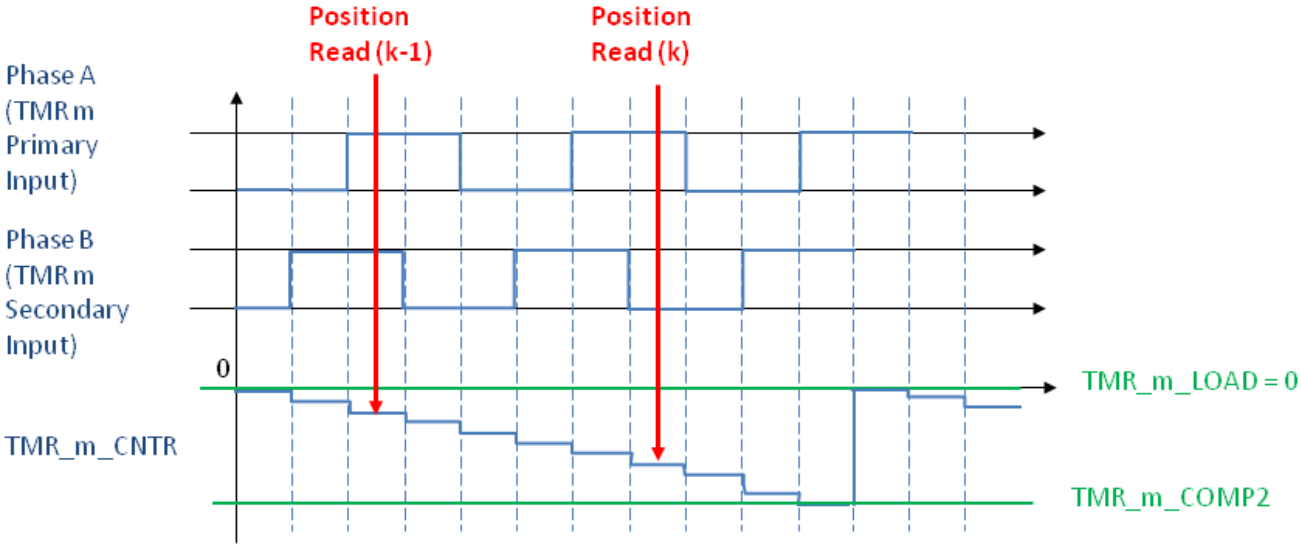

Figure 3. Position down counting using the TMR m sub-module

For the speed measurement, also capture the position of the TMR_m_CNTR to the TMR_m_CAPT register. This requires the following initialization:

- TMR_m_SCTRL[CAPTURE_MODE] = 11 = Load capture register on both edges of the secondary input

This code is used to initialize the capture setting:

```
TMR_m_SCTRL = TMR_m_SCTRL_CAPTURE_MODE_1|TMR_m_SCTRL_CAPTURE_MODE_0;
```

With this setting, the position will be captured to TMR_1_CAPT at both edges of the encoder Phase B signal.

The required mechanical rotor position is usually scaled to the fractional number. So the system range is <–1 to 1) which represents the physical range of <–180 to 180) degrees.

Fractional number thetaKMechanical, which is scaled;

$$thetaKMechanical = TMR\_m\_CNTR * thetaScale << thetaScaleShift$$

which is equivalent to:

$$thetaKMechanical = TMR\_m\_CNTR * thetaScale * 2\text{^}thetaScaleShift$$

where:

thetaKMechanical	Mechanical angle variable scaled as a fractional number
*	Fractional multiplication
<<	Determines the number of left shifts
thetaScaleShift	Shift scale coefficient

In this simple way, using the Timer_0 submodule in Quadrature Count mode, and with fractional multiplication and shifts, the rotor position thetaKMechanical in the fractional scale can be obtained.

5 Speed measurement

To measure the rotor speed, the position difference during the time difference must be known.

A simple way for a high rotor speed is to count the position difference using the position counter described in Position detection using encoder, within a defined time period. Thus, the speed can be easily calculated because it is related to the position difference. Since this measurement resolution depends on the number of the encoder pulses over the defined period, the resolution is low and fails in the low speed region.

So, to measure the rotor speed at a low speed range, measure the time difference between two encoder edges. The speed is then inversely related to the measured time difference. But this technique requires a high-frequency clock for timing, and the resolution over a high speed range is low. Therefore, a

technique is used which calculates the speed as a division of the number of encoder pulses and the precisely measured time duration between these pulses. This technique gives a very good speed resolution over the entire speed range.

5.1 Capturing the time of the secondary input signal edges

The time duration between encoder phase pulses can be simply measured by a Quad Timer with the IP bus clock time base. Because of the flexibility of the Quad Timer module, any of the four timer sub-modules can be used in this mode.
Mark the Timer n sub-module (where n can be 0, 1, 2 or 3) for the time capture, as shown in this figure.

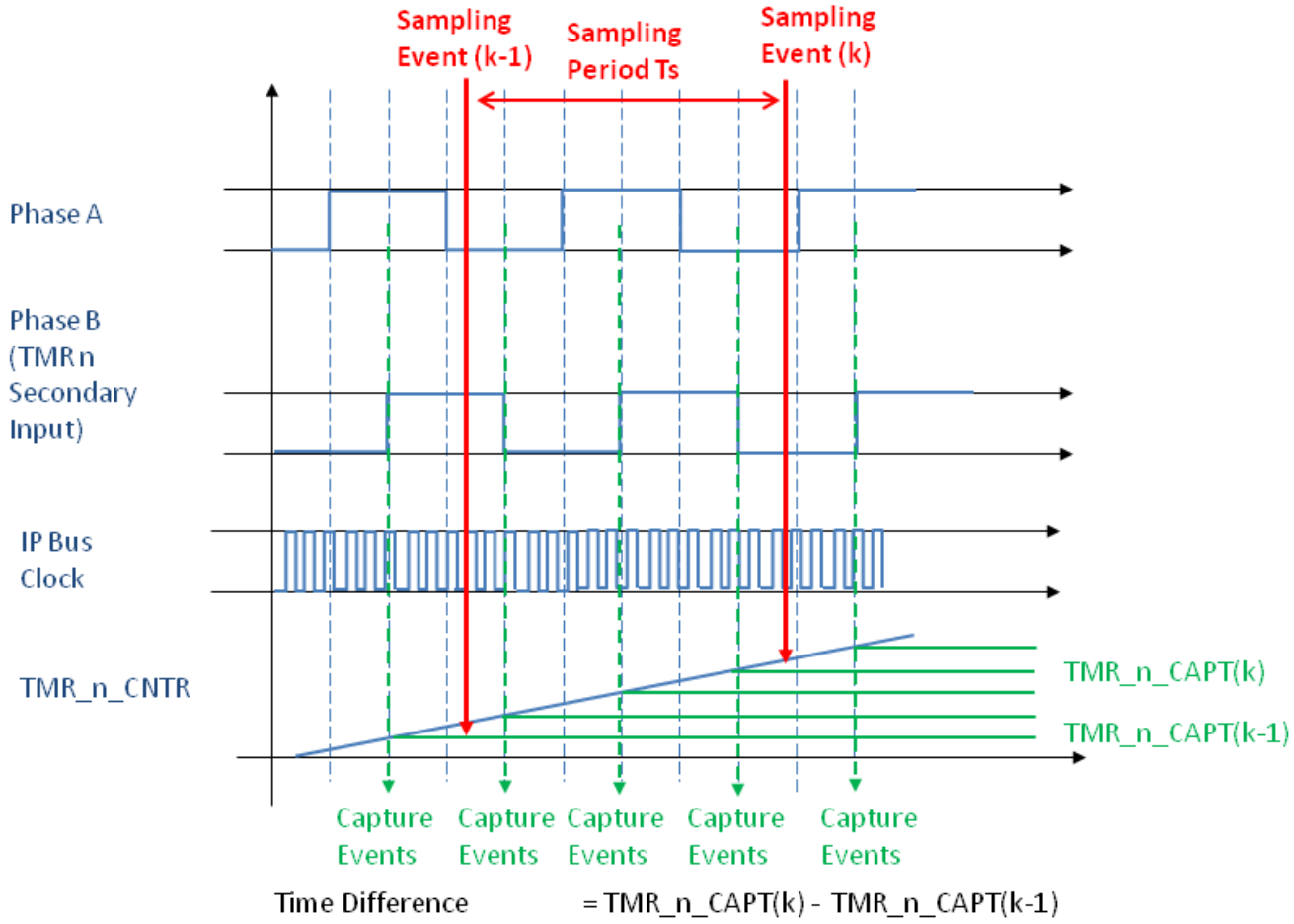

Figure 4. Time capture at the encoder Phase B edges using TMR n sub-module

The timer primary input is the IP bus clock. The secondary input utilizes the encoder phase B signal. The time is measured using the mode: count rising edges of the primary IP bus clock input. So the TMR_n_CTRL register needs to be initialized with the following bit groups:

Register fields	Value	Description
TMR_n_CTRL[CM]	001	Determines a count of the rising edges of the primary source, where the counter TMR_n_CNTR is updated according to the primary input signals.
TMR_n_CTRL[PCS]	1010	The primary timer source, the IP bus clock is divided by a prescaler, 4.
TMR_n_CTRL[SCS]	01	The secondary timer source is timer input 1 (encoder Phase B).
TMR_n_CTRL[LENGTH]	0	The counter TMR_n_CNTR counts until a roll over at 0xFFFF and continues from 0x0000.

This code line is used to initialize TMR_n_CTRL.

```
TMR_n_CTRL=(TMR_n_CTRL_CM_0|TMR_n_CTRL_PCS3|TMR_n_CTRL_PCS1|TMR_n_CTRL_SCS0)
```

The encoder phase B secondary source is used as the capture event to load the TMR_n_CAPT with the TMR_n_CNTR counter. This is determined by setting the Timer Channel Status and Control register (TMR_n_SCTRL) as follows.

- TMR_n_SCTRL[CAPTURE_MODE] = 11 = Load capture register on both edges of the secondary input

The syntax is:

```
TMR_n_SCTRL = TMR_n_SCTRL_CAPTURE_MODE_1|TMR_n_SCTRL_CAPTURE_MODE_0;
```

Now, the Sampling Event must be introduced. In control systems, there is usually a periodical event with a constant period Ts. This is asynchronous to the encoder signals of Phase B. The Sampling Event is usually provided as an interrupt, which calls an interrupt service subroutine with a constant sampling period. This interrupt can be generated with any periphery, but the most elegant solution is to use the Timer n sub-module compare functionality as will be described in Quad Timer and periodical interrupt generation. At the Sampling Event, the software provides reading of the capture registers TMR_n_CAPT. This means, we get the time of the last Phase B signal edge before the Sampling Event.

The time difference between the Phase B signal edges of the sampling events k and k–1 then needs to be calculated by the software. The time difference can be calculated by subtracting the value of TMR_n_CAPT at (k–1)th Sampling Event from the value of TMR_n_CAPT at kth Sampling Event. Therefore the software process needs to save the previously captured time to a variable, at the Sampling Event:

$$oldTimeAtCapture = newTimeAtCapture;$$
$$newTimeAtCapture = TMR\_n\_CAPT;$$

Therefore,

$$oldTimeAtCapture = TMR\_n\_CAPT(k-1);$$
$$newTimeAtCapture = TMR\_n\_CAPT(k);$$

and finally, the difference can be calculated using the following equation:

$$difTime = newTimeAtCapture - oldTimeAtCapture;$$

Note: A subtraction function with no saturation must be used for proper functionality.

5.2 Quad Timer and periodical interrupt generation

Together with the measurement of the duration between secondary input edges, the sub-module for time capture can also be used to provide the Sampling Event time base as shown in the following figure.

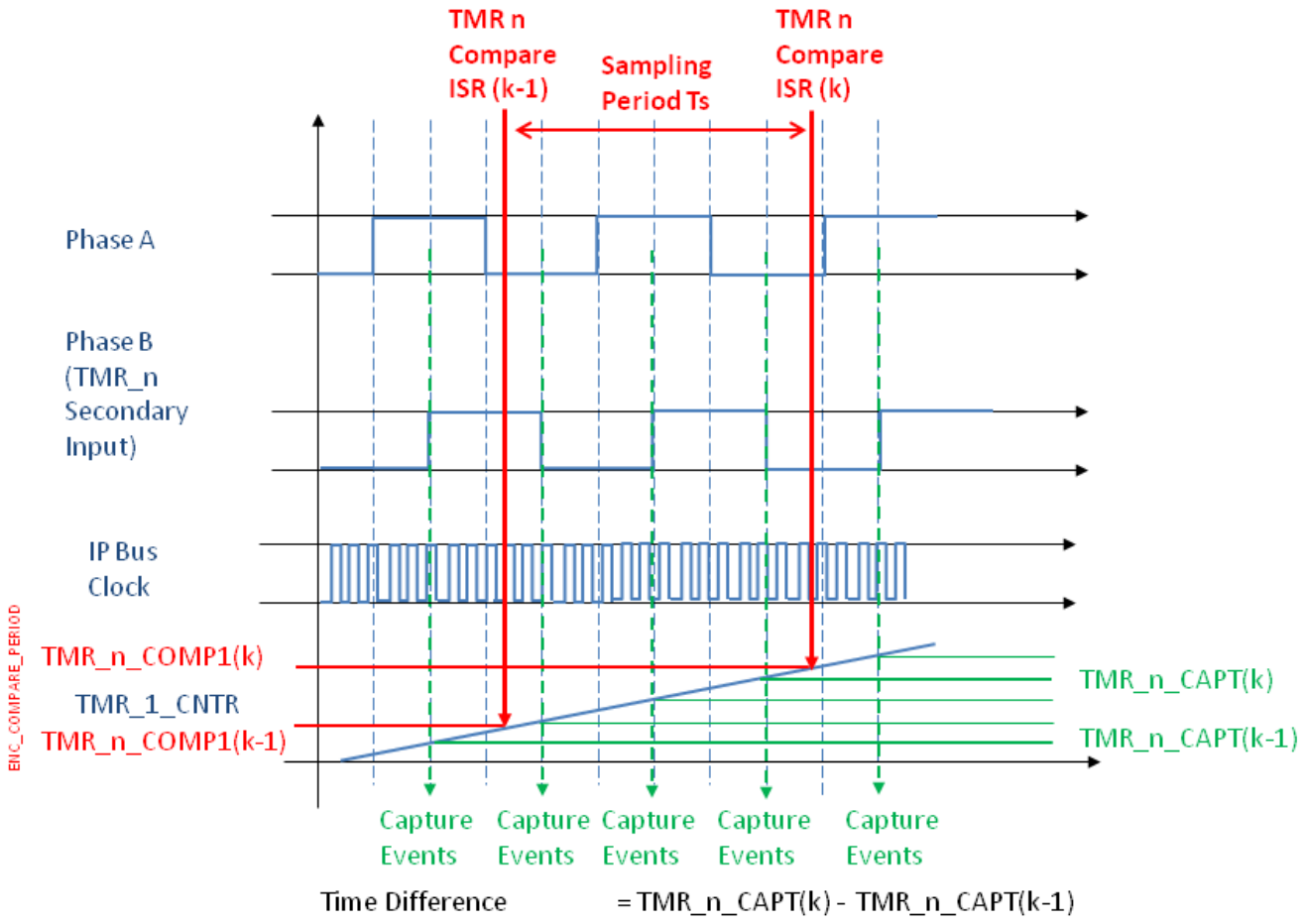

Figure 5. Time capture at the encoder Phase B edges and compare interrupt using TMR n sub-module

The Sampling Event with period Ts will be generated using the Output Compare register TMR_n_COMP1. TMR_n_COMP1 needs to be updated on each compare event to provide a periodical interrupt. So, the value of TMR_n_COMP1 at the kth sample is given by this equation.

$$TMR\_n\_COMP1(k+1) = sampling\ time\ Ts + TMR\_n\_COMP1(k)$$

The loading of TMR_n_COMP1 can be double-buffered in the Timer Comparator Load Control (TMR_n_CMPLD1) register, by using the Compare Load Control (CL) field in the Comparator Status and Control register (TMR_n_CSCTRL) as follows.

- When TMR_n_CSCTRL[CL1] = 01, the COMP1 will be preloaded with the value from TMR_n_CMPLD1 upon successful TMR_n_COMP1 compare event.

The following code line is used.

```
TMR_n_CSCTRL = TMR_n_CSCTRL_CL1_0;
```

At a TMR_n_COMP1 compare event, the TMR n interrupt subroutine is called. In this sampling routine, the software needs to update the TMR_n_CMPLD1 using this code line.

```
#define ENC_COMPARE_PERIOD       /* sampling time Ts in system units */
TMR_n_CMPLD1 = (TMR_n_COMP1+ENC_COMPARE_PERIOD);
```

The compare interrupt needs to be enabled in TMR_n_SCTRL using the following code line.

```
TMR_n_SCTRL |= TMR_n_SCTRL_TCFIE;
```

The functionality is displayed in Figure 5.

5.3 Quad Timer and rotor speed measurement

As already mentioned, speed measurement (which is suitable over a broad speed range) requires detection of the position and time differences between any of two encoder signal edges.

Because of the flexibility of the Quad Timer module, any of the four timer sub-modules can be used for the rotor position or time differences. Let’s use the timer TMR 0 (position measurement) and TMR 1 (time measurement) sub-modules.

In the interrupt subroutine, the processor will read the last encoder phase B (or phase A) edge position from the TMR 0 Capture register TMR_0_CAPT. It will also read the edge time from the TMR 1 Capture register TMR_1_CAPT.

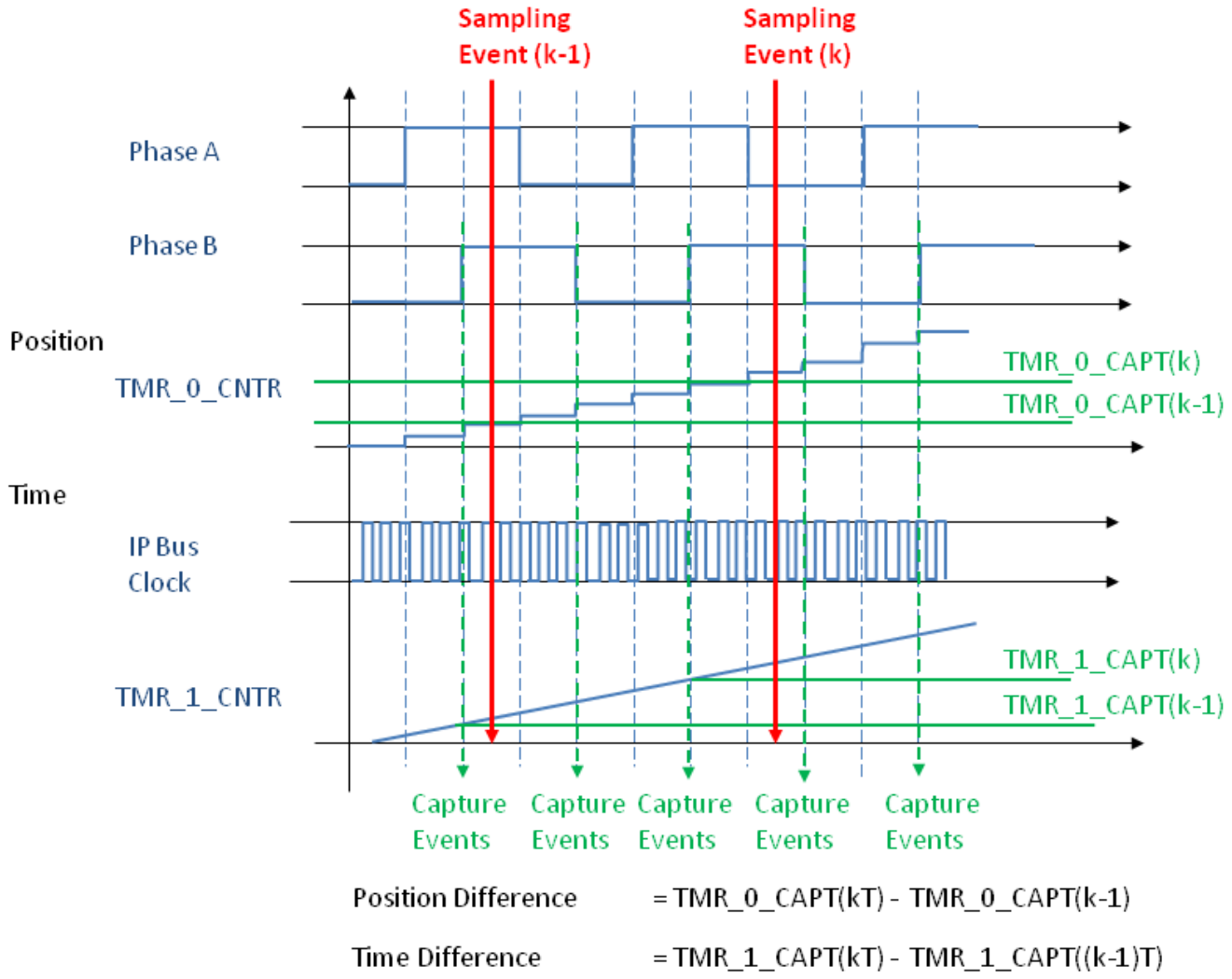

Figure 6. Position and time difference of the encoder Phase B edges using the TMR_0 and TMR_1 sub-modules

For the speed measurement, also capture the position of the TMR_0_CNTR to the TMR_0_CAPT register. This is provided by initialization of the TMR_0_SCTRL register with:

- When TMR_0_SCTRL[CAPTURE_MODE] = 11, the TMR_n_CAPT register is loaded on both edges of the secondary input.

This is the final code line to initialize TMR_0_SCTRL.

```
TMR_0_SCTRL = TMR_0_SCTRL_CAPTURE_MODE_1|TMR_0_SCTRL_CAPTURE_MODE_0;
```

With this setting, the position will be captured to TMR_1_CAPT at both edges of the encoder Phase B signal.
The differences between the previous sampling step (k–1) and current sampling step (k) edges will then be calculated by the software:

$$Position\ Difference\ (k) = TMR\_0\_CAPT(k) - TMR\_0\_CAPT(k-1)$$

$$Time\ Difference(k) = TMR\_1\_CAPT(k) - TMR\_1\_CAPT(k-1)$$

So, on each sampling interrupt, the system variables must be loaded.

$$oldNuEdgesAtCapture\ =\ newNuEdgesAtCapture$$

$$newNuEdgesAtCapture\ =\ TMR\_0\_CAPT$$

$$difNuEdges = oldNuEdgeAtCapture - NewNuEdgesAtCapture$$

$$oldTimeAtCapture\ =\ newTimeAtCapture$$

$$newTimeAtCapture\ =\ TMR\_1\_CAPT$$

$$difTime\ =\ newTimeAtCapture - oldTimeAtCapture$$

Thus, the Position Difference and Time Difference can be obtained as the value of difNuEdges and difTime respectively.
The functionality is displayed in Figure 6.

The speed is calculated using the Position Difference and Time Difference variables:

$$Speed\ =\ ((Speed\ Scale * Position\ Difference) << Speed\ Scale\ Shift)\ /\ Time\ Difference$$

Which is equivalent to:

$$Speed\ =\ ((Speed\ Scale * Position\ Difference) * 2\hat{}Speed\ Scale\ Shift)/\ Time\ Difference$$

Speed	Mechanical speed variable scaled as a fractional number (variable **angularSpeed**)
Position Difference	The position difference between the previous sampling step (k-1) and current sampling step (k) edges (variable **difNuEdges**)
Time Difference	The time difference between the previous sampling step (k-1) and current sampling step (k) edges (variable **difTime**)
*	fractional multiplication
/	Four-quadrant division
Speed Scale	Speed scale fractional coefficient (variable **angularSpeedScale**)

<<	Determines the number of left shifts
Speed Scale Shift	Speed scale shift coefficient (variable **angularSpeedScaleShift**)

So at each sampling time (ISR/Read event in Figure 6), the DSC will calculate the rotor speed with a defined scaling, using intrinsic functions for multiplication (L_mult), shift left (L_shlfts) and division (div_ls4q). The syntax is as follows.

```
TempF32 = L_mult(difNuEdges, angularSpeedScale);
TempF32 = L_shlfts(TempF32, angularSpeedScaleShift);
angularSpeed = div_ls4q(TempF32,difTime);
```

Figure 6 shows a solution where the Sampling Event is provided by any module or event (synchronisation with PWM, PIT timer or other timing module). This is because sometimes, the sampling is already defined in the application before the encoder implementation.

However the TMR 1 can also be used to generate the required sampling interrupt. This is described in Quad Timer and periodical interrupt generation. The time diagram for this Quad timer setting is shown in the following figure. The speed calculation is the same and is provided in the TMR 1 Compare Interrupt Subroutine.

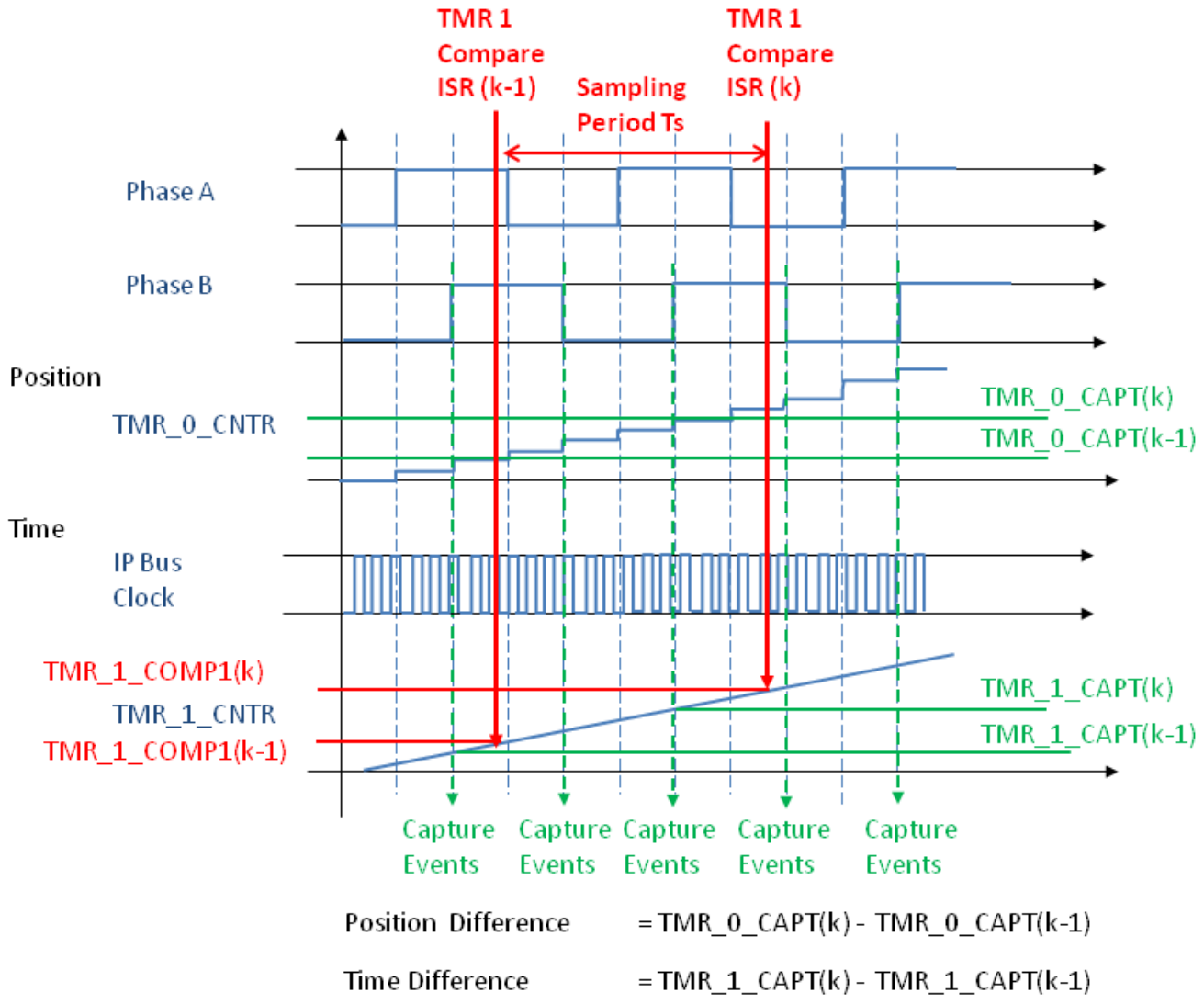

Figure 7. Position and time difference of the encoder edges with the TMR 1 Compare ISR

6 Application example

The following sections elaborate an example software for the speed and position detection using the MC56F82xxx DSC.

The connectivity of the Freescale MC56F82xxx DSC is very universal due to input signal multiplexing, a SIM module with internal peripheral multiplexing control, and the crossbar module XBARA. Therefore, the encoder phase signal can be possibly connected to any Tx_IN (timer input pin) or XB_INy (universal crossbar input) capable input pin.

This example elaborates on the position and speed measurement using the following device and signal connections:

- MC56F82723VLC device
- 32-pin LQFP package
- Core frequency 50 MHz
- An encoder device with 1024 inc/rev
- Encoder Phase A signal connected to pin 15 (GPIOC6) and Crossbar input 3 (XB_IN3).
- Encoder Phase B signal connected to pin 18 (GPIOC10) and Crossbar input 5 (XB_IN5).
- The XBARA module provides connection of XB_IN3 to T0_IN and XB_IN5 to T1_IN
- TMR 0 and TMR 1 are used for position and time detection.

The signal configuration is also shown in Figure 1. The example signals are in brackets.

7 Application example code

All the code lines used in the context of this application are given below. The code incorporates input settings, crossbar settings, interrupt vectors, and first of all, the timer settings. The final code is more complex than the previous samples, since it incorporates hazard states and limitations. Using a smart scaling technique, the mechanical and electrical rotor positions can be scaled to the fractional variables thetaKMechanical and thetaKElectrical respectively.

Interrupt Vector Table

The file MC56F827xx_vector.asm is located in *Project_Settings\Startup_Code*.

```
JSR  >TimeBaseISR                ; /* 0x36 Interrupt no. 27 */
```

Included Header Files

The most important headers used in the following code:

```
#include "MC56F82723.h"          /* MC56F82723 Peripheral description header */
#include <intrinsics_56800E.h>   /* intrinsics arithmetic header */
```

Constants and definitions

```
      /* Encoder position scale */
#define ENC_EL_POSITION_SCALE                (0.500)
#define ENC_EL_POSITION_SCALE_SHIFT          (7)

      /* Encoder position scale */
#define ENC_MECH_POSITION_SCALE              (0.500)
#define ENC_MECH_POSITION_SCALE_SHIFT        (5)

      /* Encoder speed scale (speed calculated from position derivation) */
#define ENC_SPEED_SCALE                      (0.6357829)
#define ENC_SPEED_SCALE_SHIFT                (4)

      /* QTimer A1 compare register #1 sampling time Ts in IP bus clock */
#define ENC_COMPARE_PERIOD                   (12500)
      /* Number of encoder edges per mechanical revolution */
#define ENC_NU_EDGES_REV                     (4096)
      /* Modulo division of encoder edges counter */
```

```
#define ENC_BI_MODULO                          (ENC_NU_EDGES_REV-1)
      /* QTimer A0 - max. number of edges between two consecutive readings */
#define ENC_NU_EDGES_LIMIT                     ((ENC_NU_EDGES_REV)/2-1)

/** Other System Values (NOT USED IN THIS APPLICATION EXAMPLE) **/
      /* All parameters calculated for bus clock frequency [Hz]*/
#define IPBUS_CLOCK_FREQ_HZ                    50000000.0
      /* Application speed maximal range [rpm] */
#define APP_SPEED_MAX_RPM                      18000.0
      /* Number of motor pole pairs */
#define MOTOR_NUMBER_OF_POLE_PAIRS             4
      /* Period of encoder speed sampling [s] */
#define PER_ENCODER_SPEED_SAMP_S               (0.001)
      /* Encoder position range maximum [Degree] */
#define ENC_MAX_POSITION_DEG                   180


typedef struct
{
      Word16       prevCaptureOKFlag;
      Word16       newNuEdgesAtCapture;       /* number of edges at new capture */
      UWord16      newTimeAtCapture;          /* time instant at new capture */
      Word16       oldNuEdgesAtCapture;       /* number of edges previous edge */
      UWord16      oldTimeAtCapture;          /* time instant at previous capture */
      Word16       difNuEdges;                /* difference of edges between captures */
      UWord16      difTime;                   /* difference of time between captures */
      Frac16       tmpAngularSpeed;           /* Angular Speed */

      Frac16       angularSpeedScale;         /* Speed scale */
      Frac16       angularSpeedScaleShift;    /* Speed scale shift */

      Word16       maxNuEdges;                /* encoder max. possible edges difference
                                                 between captures */
      Frac16   encNuEdgesRev;                 /* encoder pulses per revolution */
}ENC_SPEED_STRUCT;



typedef struct
{
      Frac16       positionCounter;           /* Timer A0 counter reg. */
      Frac16       thetaScale;                /* position scale */
      Int16        thetaScaleShift;           /* position scale shift */
}ENC_POSITION_STRUCT;
```

Variables

```
      /* encoder electrical position sensing parameters */
static ENC_POSITION_STRUCT              encElPosParam;
      /* encoder mechanical position sensing parameters */
static ENC_POSITION_STRUCT              encMechPosParam;
      /* encoder speed sensing parameters */
static ENC_SPEED_STRUCT                  encSpeedParam;
      /* Motor speed Sensed with encoder */
static Frac16                           speedMotorSens;
      /* actual motor electrical position from encoder */
static Frac16                            thetaKElectrical;
      /* actual motor mechanical position from encoder */
static Frac16                           thetaKMechanical;
```

Prototypes

```
static void GPIOC_Init(void);
static void XBAR_Init(void);
static void ENC_PositionTimer0Init (void);
static void ENC_TimeBaseTimer1Init (void);

Frac16 ENC_PositionGet(ENC_POSITION_STRUCT *ptr);
void ENC_TimeBaseNew(void);
Frac16 ENC_AngularSpeed(ENC_SPEED_STRUCT *ptr);


void TimeBaseISR (void);
```

Functions

```
static void GPIOC_Init(void)
{
              /* Enable GPIOC clock */
       SIM_PCE0 |= SIM_PCE0_GPIOC;
              /* Encoder Phase A – C6, Phase A – C10 set as peripheral */
      GPIOC_PER |= (GPIOA_PER_PE_10 | GPIOA_PER_PE_6);
              /* Select TMR0 Input from XBAR XB_OUT34 TMR1 Input from XBAR XB_OUT35 */
              /* otherwise the PIOC3 and PIOC4 peripheral inputs are used */
      SIM_IPSn |= (SIM_IPSn_TA0 | SIM_IPSn_TA1);
             /* Set C6 as XB_IN3 XBAR input ALT = 01 */
      SIM_GPSCL  &= ~( SIM_GPSCL_C6_1);
      SIM_GPSCL  |=  ( SIM_GPSCL_C6_0);
            /* Set C10 as XB_IN5 XBAR input ALT = 01 */
      SIM_GPSCH  &= ~( SIM_GPSCH_C10_1);
      SIM_GPSCH  |=  ( SIM_GPSCH_C10_0);
}


static void XBAR_Init(void)
{
             /* XB_IN3 XBAR A input to T0_IN
                 XBARA_SEL17 = (XBARA_SEL17_SEL34_1|XBARA_SEL17_SEL34_0)|\
                                (XBARA_SEL17_SEL35_2|XBARA_SEL17_SEL35_0); */
             /* XB_IN3 (Encoder Phase A) muxed to Timer 0 input T1_IN
                XB_IN5 (Encoder Phase B) muxed to Timer 1 input T1_IN */
       XBARA_SEL17 = (3 << 5) | 3;
}

static void ENC_PositionTimer0Init(void)
{
              /* Enable TMR0 clock */
      SIM_PCE0 |= SIM_PCE0_TA0;
      TMR_0_CTRL = TMR_0_CTRL_CM_2|TMR_0_CTRL_SCS_0|TMR_0_CTRL_LENGTH;
      TMR_0_SCTRL = TMR_0_SCTRL_CAPTURE_MODE_1|TMR_0_SCTRL_CAPTURE_MODE_0;
      TMR_0_CSCTRL = 0;
      TMR_0_COMP1 = (ENC_NU_EDGES_REV-1);
      TMR_0_COMP2 = (-(ENC_NU_EDGES_REV-1));
      TMR_0_CMPLD1 = 0;
      TMR_0_CMPLD2 = 0;
      TMR_0_LOAD = 0;
      TMR_0_CNTR = 0;
}

static void ENC_TimeBaseTimer1Init(void)
{
      /*     Initialization of QTA1 is to be called in main
```

```
            The user should define the QTA1 Compare Interrupt function */
            /* Enable TMR1 clock */
      SIM_PCE0 |= SIM_PCE0_TA1;
      TMR_1_SCTRL = TMR_1_SCTRL_TCFIE|TMR_1_SCTRL_CAPTURE_MODE_1|TMR_1_SCTRL_CAPTURE_MODE_0;
      TMR_1_CSCTRL = TMR_1_CSCTRL_CL1_0;
      TMR_1_CTRL = (TMR_1_CTRL_CM_0|TMR_1_CTRL_PCS3|TMR_1_CTRL_PCS1|  TMR_1_CTRL_SCS0);
      TMR_1_COMP1 = (ENC_COMPARE_PERIOD-1);
      TMR_1_COMP2 = 0;
      TMR_1_CMPLD1 = (2*(ENC_COMPARE_PERIOD)-1);
      TMR_1_CMPLD2 = 0;
      TMR_1_LOAD = 0;
      TMR_1_CNTR = 0;
}

Frac16 ENC_PositionGet(ENC_POSITION_STRUCT *ptr)
{
      ptr->positionCounter = (Frac16)TMR_0_CNTR;
      return((Frac16)extract_h((L_mult(ptr->positionCounter,ptr->thetaScale))\
                              <<(ptr->thetaScaleShift)));
}

void ENC_TimeBaseNew(void)
{
/************ INTERRUPT ON QT1 COMPARE ***************/
      /* Calculate the new compare value for QTA1 */
      TMR_1_CMPLD1 = (TMR_1_COMP1+ ENC_COMPARE_PERIOD);
      TMR_1_SCTRL &= ~(TMR_1_SCTRL_TCF);
/************ INTERRUPT ON QT1 COMPARE ***************/
}

Frac16 ENC_AngularSpeed(ENC_SPEED_STRUCT *ptr)
{
      register Frac32 TempF32;
      if (TMR_1_SCTRL & TMR_1_SCTRL_IEF) /* CAPTURE OCCURED */
      {
/************* READ CAPTURE REGISTERS *****************/
            /* read Number of Encoder Pulses stored in QT0_Capture register */
            ptr->newNuEdgesAtCapture   = (Word16)TMR_0_CAPT;
            /* read Exact time of encoder pulses stored QT1_Capture register */
            ptr->newTimeAtCapture      = (Word16)TMR_1_CAPT;
/************* READ CAPTURE REGISTERS *****************/
            /* Clear capture flag */
            TMR_0_SCTRL &= ~(TMR_0_SCTRL_IEF);
            TMR_1_SCTRL &= ~(TMR_1_SCTRL_IEF);
            /* Avoid a hardware hazard, when capture event occurs
                  only in one timer. This can happen when a capture
                  edge comes in the middle of the flags clear procedure */
            if ((TMR_0_SCTRL & TMR_0_SCTRL_IEF)\
                || (TMR_1_SCTRL & TMR_1_SCTRL_IEF))
            {
                  TMR_0_SCTRL &= ~(TMR_0_SCTRL_IEF);
                  TMR_1_SCTRL &= ~(TMR_1_SCTRL_IEF);
            }
            if(ptr->prevCaptureOKFlag)
            {
                  ptr->difNuEdges=ptr->newNuEdgesAtCapture-ptr->oldNuEdgesAtCapture;
                  ptr->difTime =ptr->newTimeAtCapture-ptr->oldTimeAtCapture;
/******************* REMOVE ERROR ********************/
            /* Remove error in difNuEdges created by wrong subtraction
                  A wrong subtraction occurs when QT0 passes the ENC_PULSES_REV value
                  because a wrap-around subtraction doesn't work with the ENC_PULSES_REV
                  value*/
```

```
                    if (ptr->difNuEdges>ptr->maxNuEdges)
                    {
                          ptr->difNuEdges     = ptr->difNuEdges - ptr->encNuEdgesRev;
                    }
                    else if (ptr->difNuEdges<(-ptr->maxNuEdges))
                    {
                          ptr->difNuEdges     = ptr->difNuEdges + ptr->encNuEdgesRev;
                    }
/******************* REMOVE ERROR ********************/
/***************** SPEED CALCULATION *****************/
                    turn_on_sat();
                    if ((ptr->difTime<=(UWord16)MAX_16))
                    {
                          TempF32 = L_shlfts(L_mult((Frac16)ptr->difNuEdges,\
                                                    (Frac16)ptr->angularSpeedScale),\
                                             ptr->angularSpeedScaleShift);
                          if (extract_h(TempF32)>=(Frac16)ptr->difTime)
                          {
                                ptr->tmpAngularSpeed=MAX_16;
                          }
                          else if (extract_h(-TempF32)>=(Frac16)ptr->difTime)
                          {
                                ptr->tmpAngularSpeed = MIN_16;
                          }
                          else
                          {
                               ptr->tmpAngularSpeed = ((div_ls4q(TempF32,
                                                          ((Frac16)(ptr->difTime))));
                          }
                    }
                    else
                    {
                          ptr->tmpAngularSpeed = 0;
                    }
                    turn_off_sat();
              }
/***************** SPEED CALCULATION *****************/

/************ STORE NEWLY CAPTURED VALUES ***************/
             ptr->oldNuEdgesAtCapture   = ptr->newNuEdgesAtCapture;
             ptr->oldTimeAtCapture      = ptr->newTimeAtCapture;
             ptr->prevCaptureOKFlag = 1;//Debug 08.11.19
/************ STORE NEWLY CAPTURED VALUES ***************/
       }
       else   /* CAPTURE DID NOT OCCUR */
       {
             ptr->newTimeAtCapture = (UWord16)TMR_1_CNTR;
             ptr->difTime  = ptr->newTimeAtCapture - ptr->oldTimeAtCapture;

             if (ptr->difTime>(UWord16)MAX_16)
             {
                   /* Set calculated speed to zero,
                      and wait for the next two successive captures */
                   ptr->prevCaptureOKFlag = 0;
                   ptr->tmpAngularSpeed = 0;
             }
       }
       return(ptr->tmpAngularSpeed);
}
```

Main function and initializations

In our example, the position is periodically read in the s/w main loop, but the position can be read from any interrupt subroutine (for example from TimeBaseISR):

```
thetaKElectrical = ENC_PositionGet(&encElPosParam);
thetaKMechanical = ENC_PositionGet(&encMechPosParam);
```

The initialization and main s/w loop is below:

```
void main (void)
{

      GPIOC_Init();
      XBAR_Init();
      ENC_PositionTimer0Init();
      ENC_TimeBaseTimer1Init();

       encElPosParam.thetaScale         = FRAC16(ENC_EL_POSITION_SCALE);
       encElPosParam.thetaScaleShift    = ENC_EL_POSITION_SCALE_SHIFT;

       encMechPosParam.thetaScale       = FRAC16(ENC_MECH_POSITION_SCALE);
       encMechPosParam.thetaScaleShift  = ENC_MECH_POSITION_SCALE_SHIFT;

       encSpeedParam.encNuEdgesRev      = ENC_NU_EDGES_REV;
       encSpeedParam.maxNuEdges         = ENC_NU_EDGES_LIMIT;
       encSpeedParam.angularSpeedScale  = FRAC16(ENC_SPEED_SCALE);
       encSpeedParam.angularSpeedScaleShift  = ENC_SPEED_SCALE_SHIFT;

       while(1)
       {
              /* get the electrical position using the ENC_PositionGet function */
              thetaKElectrical = ENC_PositionGet(&encElPosParam);
              /* get the mechanical position using the ENC_PositionGet function */
              thetaKMechanical = ENC_PositionGet(&encMechPosParam);
       }
}
```

Time base interrupt subroutine

```
#pragma interrupt saveall
void TimeBaseISR(void)
{
    /* Calculate mechanical motor speed */
    ENC_TimeBaseNew();

    /* Calculate mechanical motor speed */
    speedMotorSens = ENC_AngularSpeed(&encSpeedParam);
}
```

8 Definitions and acronyms

ADC	Analogue-to-Digital Converter
AOI	And/Or/Invert Module
CW	CodeWarrior

DSC	Digital Signal Controller
FOC	Field Oriented Control
GPIO	General Port Input Output
ISR	Interrupt Service Routine
(k-1)	Previous Sampling Step
k	Sampling Step
(k+1)	Next Sampling Step
PWM	Pulse-Width Modulation
SIM	System Integration Module
Motor control	In this application note, this means a process which controls an electrical motor such as a BLDC PMSM, AC-induction or other
XBAR	Cross-Bar Switch
Ts	The time base - sampling period

9 Revision history

Revision number	Date	Substantial changes
0	10/2013	Initial release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number AN4813
Revision 0, October 2013