Texas Instruments, Inc.
C2000 Systems and Applications

Digital Motor Control

Software Library:
F2833x Drivers

2012

Contents

Introduction

The digital motor control library is composed of C functions (or macros) developed for C2000 motor control users. These modules are represented as modular blocks in C2000 literature in order to explain system-level block diagrams clearly by means of software modularity. The DMC library modules cover nearly all of the target-independent mathematical macros and target-specific peripheral configuration macros, which are essential for motor control. These modules can be classified as:

Transformation and Observer Modules	Clarke, Park, Phase Voltage Calculation, Sliding Mode Observer, BEMF Commutation, Direct Flux Estimator, Speed Calculators and Estimators, Position Calculators and Estimators etc.
Signal Generators and Control Modules	PID, Commutation Trigger Generator, V/f Controller, Impulse Generator, Mod 6 Counter, Slew Rate Controllers, Sawtooth & Ramp generators, Space Vector Generators etc.
Peripheral Drivers	PWM abstraction for multiple topologies and techniques, ADC Drivers, Hall Sensor Driver, QEP Driver, CAP Driver etc.
Real-Time Debugging Modules	DLOG module for CCS graph window utility, PWMDAC module for monitoring the control variables through socilloscope

In the DMC library, each module is separately documented with source code, use, and background technical theory. All DMC modules allow users to quickly build, or customize their own systems. The library supports three principal motor types (induction motor, BLDC and PM motors) but is not limited to these motors.

The DMC library components have been used by TI to provide system-level motor control examples. In the motor control code, all DMC library modules are initialized according to the system specific parameters, and the modules are inter-connected to each other. At run-time the modules are called in order. Each motor control system is built using an incremental build approach, which allows some sections of the code to be built at a time, so that the developer can verify each section of the application one step at a time. This is critical in real-time control applications, where so many different variables can affect the system, and where many different motor parameters need to be tuned.

Description

This module allows the user to configure analog-to-digital conversion (ADC) channels. The conversions are triggered on end of conversion (EOC) which is set to 4th conversion by default. In DMC projects, the converted results represent load currents and DC-bus voltage but not limited to these.

Availability

C interface version

Module Properties

Type: Target Dependent, Application Independent

Target Devices: 28x Floating Point

C Version File Names: f2833xileg_vdc.h (for x2833x)

IQmath library files for C: N/A

C Interface

Module Usage

Instantiation
There is no instantiation for ADC configuration.

Initialization

```
// Default ADC initialization
int ChSel[16]    = Default_ch_sel;
int TrigSel[16]  = Default_trig_sel;
int ACQPS[16] = Default_ACQPS;
```

where

ChSel [ ] stores which ADC pin is used for conversion when a Start of Conversion (SOC) trigger is received for the respective channel

TrigSel [ ] stores what trigger input starts the conversion of the respective channel

ACQPS [ ] stores the acquisition window size used for the respective channel

Invoking the computation macro

```
ChSel[0]=x;
ChSel[1]=y;
ChSel[2]=z;

ADC_MACRO_INIT(ChSel,TrigSel,ACQPS);
```

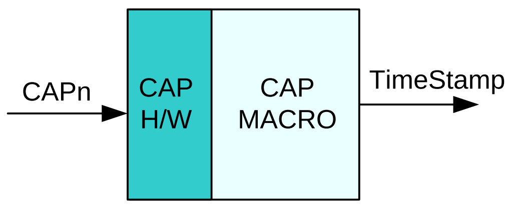
Example

The following pseudo code provides the information about the module usage.

```
main()
{
        ChSel[1]=1;      // ChSelect: ADC A1-> Phase A Current
        ChSel[2]=9;      // ChSelect: ADC B1-> Phase B Current
        ChSel[7]=10;     // ChSelect: ADC B2-> DC Bus  Voltage

        ADC_MACRO_INIT(ChSel,TrigSel,ACQPS)          // Call init macro for ADC INIT
}

void interrupt periodic_interrupt_isr()
{
clarke1.As=((AdcMirror.ADCRESULT0)*0.00024414-offsetA)*2*0.909; // Phase A curr.
clarke1.Bs=((AdcMirror.ADCRESULT1)*0.00024414-offsetB)*2*0.909; // Phase B curr.
// phase current meas: ((ADCmeas(q12)/2^12)-offset)*2*(3.0/3.3)

volt1.DcBusVolt = ((AdcMirror.ADCRESULT2)*0.00024414)*0.909;
// DC Bus voltage meas: (ADCmeas(q12)/2^12)*(3.0V/3.3V)
}
```

Description

This module provides the instantaneous value of the selected time base (GP Timer) captured on the occurrence of an event. Such events can be any specified transition of a signal applied at ECAP input pins of 2833x devices.



Availability

C interface version

Module Properties

Type: Target Dependent, Application Independent

Target Devices: 28x Floating Point

IQmath library files for C: IQmathLib.h, IQmath.lib

C Version File Names: f2833xcap.h (for x2833x)

C Interface

Object Definition

The structure of CAPTURE object is defined by following structure definition for x2833x series

```
typedef struct { int32 EventPeriod;       // Output: Timer value difference between two edges (Q0)
                 Uint16 CapReturn;        // Output: Status of one entry of first event of ECAP pin (Q0)
               } CAPTURE;
```

Item	Name	Description	Format	Range(Hex)
Inputs	CAPn	Capture input signals to 28x device	N/A	0-3.3 v
Outputs	EventPeriod (x2833x)	Timer value difference between two edges detected.	0	80000000-7FFFFFFF
	CapReturn	Ecap first event status	0	0-1

Special Constants and Data types

CAPTURE

The module definition is created as a data type. This makes it convenient to instance an interface to the CAPTURE driver. To create multiple instances of the module simply declare variables of type CAPTURE.

CAPTURE_DEFAULTS

Structure symbolic constant to initialize CAPTURE module. This provides the initial values to the terminal variables as well as method pointers.

This default definition of the object implements two methods – the initialization and the runtime compute macro for CAPTURE generation. This is implemented by means of a macro pointer, and the initializer sets this to CAP_INIT_MACRO and CAP_MACRO macros for x2833x. The argument to this macro is the address of the CAPTURE object.

Module Usage

Instantiation
The following example instances one CAPTURE object
CAPTURE cap1;

Initialization
To Instance pre-initialized objects
CAPTURE cap1 = CAPTURE_DEFAULTS;

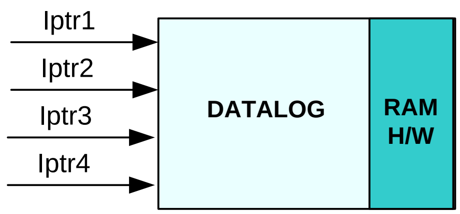
Invoking the computation macro
CAP_INIT_MACRO (1); // For cap#1
CAP_MACRO (1,cap1);

Example
The following pseudo code provides the information about the module usage.

```
main()
{

        CAP_INIT_MACRO(1);                    // Call init macro for cap1

}


void interrupt periodic_interrupt_isr()
{
   Uint16 CapReturn;
   Uint32 EventPeriod;

   CapReturn = CAP_MACRO(1,cap1);

   //  if status==1 then a time stamp was not read,
   //   if status==0 then a time stamp was read

   if(status==0)
   {
      EventPeriod=(int32)(cap1.EventPeriod);            // Read out new time stamp
   }

}
```

Description

This module stores the real-time values of four user selectable software Q15 variables in the data RAM provided on the 28xx DSP. Four variables are selected by configuring four module inputs, *iptr1, iptr2, iptr3* and *iptr4*, point to the address of the four variables. The starting addresses of the four RAM locations, where the data values are stored, are set to DLOG_4CH_buff1, DLOG_4CH_buff2, DLOG_4CH_buff3, and DLOG_4CH_buff4. The DATALOG buffer size, prescalar, and trigger value are also configurable.



Availability

This 16-bit module is available in one interface format:

The CcA interface version

Module Properties

Type: Target Dependent, Application Independent

Target Devices: Fixed Point x280x or floating point x2833x devices

CcA Version File Names: dlog4chc.asm, dlog4ch.h

IQmath library files for C: N/A

C Interface

Object Definition

The structure of DLOG_4CH object is defined by following structure definition

```
typedef struct {    long  task;         // Variable:  Task address pointer
                    int  *iptr1;        // Input: First input pointer (Q15)
                    int  *iptr2;        // Input: Second input pointer (Q15)
                    int  *iptr3;        // Input: Third input pointer (Q15)
                    int  *iptr4;        // Input: Fourth input pointer (Q15)
                    int  trig_value;    // Input: Trigger point (Q15)
                    int  prescalar;     // Parameter: Data log prescale
                    int  skip_cntr;     // Variable:  Data log skip counter
                    int  cntr;          // Variable:  Data log counter
                    long write_ptr;     // Variable:  Graph address pointer
                    int  size;          // Parameter: Maximum data buffer
                    int  (*init)();     // Pointer to init function
                    int  (*update)();   // Pointer to update function
                  } DLOG_4CH;
```

Item	Name	Description	Format	Range(Hex)
Inputs	iptr1	Input pointer for the first Q15 variable	Q0	00000000-FFFFFFFF
	iptr2	Input pointer for the second Q15 variable	Q0	00000000-FFFFFFFF
	iptr3	Input pointer for the third Q15 variable	Q0	00000000-FFFFFFFF
	iptr4	Input pointer for the fourth Q15 variable	Q0	00000000-FFFFFFFF
Outputs	N/A	Data RAM	N/A	N/A
DATALOG Parameter	prescalar	Data log prescaler	Q0	0000-7FFF
	trig_value	Trigger point based on the fist Q15 variable	Q15	8000-7FFF
	size	Maximum data buffer	Q0	0000-7FFF
Internal	skip_cntr	Data log skip counter	Q0	0000-7FFF
	cntr	Data log counter	Q0	0000-7FFF
	write_ptr	Graph address pointer	Q0	00000000-FFFFFFFF
	task	Task address pointer	Q0	00000000-FFFFFFFF

Note: The trigger value is with reference to the input *iptr1. In accordance with this, the input connected to channel 1 should be time varying, and the trigger value should be set up such that input crosses the trigger value.

The other channels are captured synchronous to the channel 1. There is no trigger mechanism on channels 2 through 4.

Special Constants and Data types

DLOG_4CH
The module definition is created as a data type. This makes it convenient to instance an interface to the DATALOG driver. To create multiple instances of the module simply declare variables of type DLOG_4CH.

DLOG_4CH_handle
User defined Data type of pointer to DATALOG module

DLOG_4CH_DEFAULTS
Structure symbolic constant to initialize DATALOG module. This provides the initial values to the terminal variables as well as method pointers.

Methods

int DLOG_4CH_init(DLOG_4CH *);
int DLOG_4CH_update(DLOG_4CH *);

This default definition of the object implements two methods – the initialization and the runtime update function for DATALOG. This is implemented by means of a function pointer, and the initializer sets this to DLOG_4CH_init and DLOG_4CH_update functions for x281x/x280x. The argument to this function is the address of the DATALOG object.

Module Usage

Instantiation
The following example instances one DATALOG object
DLOG_4CH dlog1;

Initialization
To Instance pre-initialized objects
DLOG_4CH dlog1 = DLOG_4CH_DEFAULTS;

Invoking the computation function
dlog1.init(&dlog1);
dlog1.update(&dlog1);

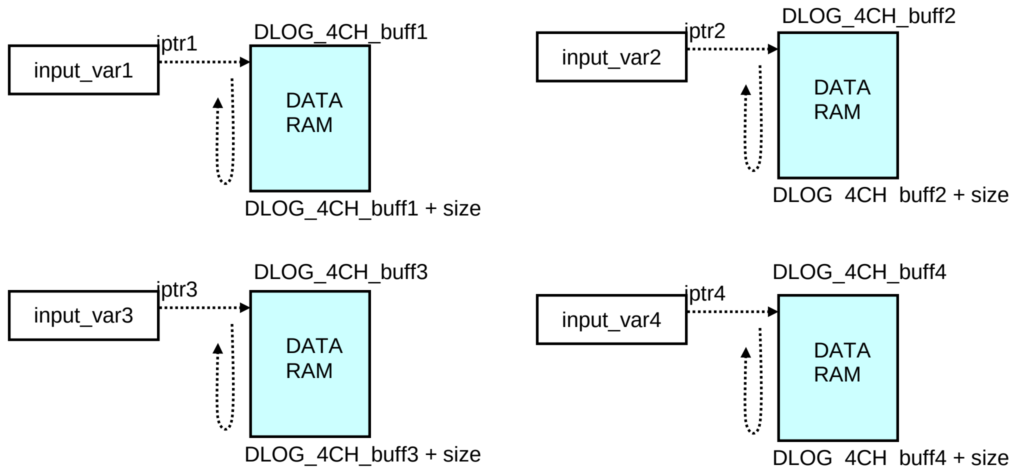
Example

The following pseudo code provides the information about the module usage.

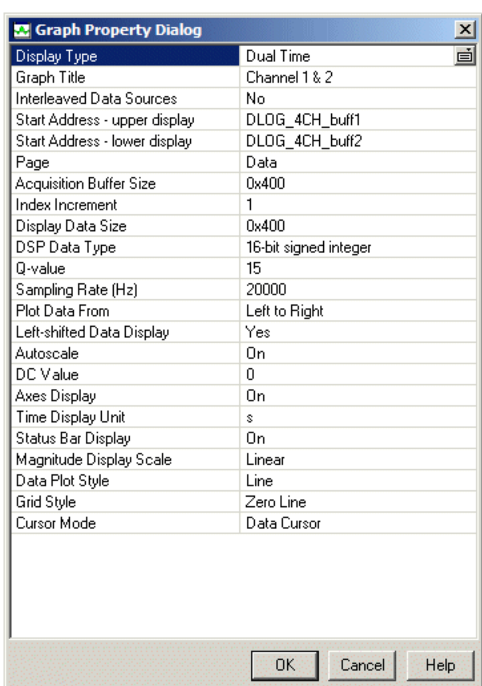
```
main()
{

    dlog1.iptr1 = &Q15_var1;        // Pass input to DATALOG module
    dlog1.iptr2 = &Q15_var2;        // Pass input to DATALOG module
    dlog1.iptr3 = &Q15_var3;        // Pass input to DATALOG module
    dlog1.iptr4 = &Q15_var4;        // Pass input to DATALOG module
    dlog1.trig_value = 0x0;         // Pass input to DATALOG module
    dlog1.size = 0x400;             // Pass input to DATALOG module
    dlog1.prescalar = 1;            // Pass input to DATALOG module
    dlog1.init(dlog1);              // Call init function for dlog1
}

void interrupt periodic_interrupt_isr()
{

    dlog1.update(&dlog1);           // Call update function for dlog1

}
```

Technical Background

This software module stores up to four real-time Q15 values of each of the selected input variables in the data RAM as illustrated in the following figures. The starting addresses of four RAM sections, where the data values are stored, are set to DLOG_4CH_buff1, DLOG_4CH_buff2, DLOG_4CH_buff3, and DLOG_4CH_buff4.



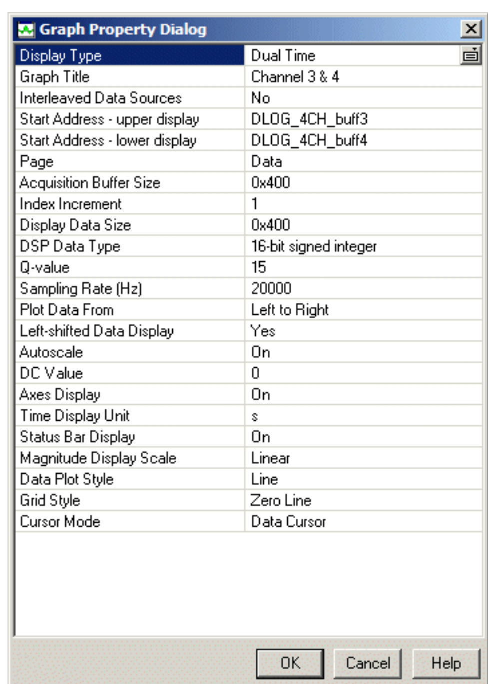
To show four stored Q15 variables in CCS graphs properly, the properties of two dual time graphs for these variables should be configured as shown in the following figures. In the software, the default buffer size is 0x400. The sampling rate is usually same as ISR frequency. In this case, it is 20 kHz with the prescalar of 1.



Graph Property Dialog

Display Type	Dual Time
Graph Title	Channel 1 & 2
Interleaved Data Sources	No
Start Address - upper display	DLOG_4CH_buff1
Start Address - lower display	DLOG_4CH_buff2
Page	Data
Acquisition Buffer Size	0x400
Index Increment	1
Display Data Size	0x400
DSP Data Type	16-bit signed integer
Q-value	15
Sampling Rate (Hz)	20000
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s
Status Bar Display	On
Magnitude Display Scale	Linear
Data Plot Style	Line
Grid Style	Zero Line
Cursor Mode	Data Cursor

OK Cancel Help



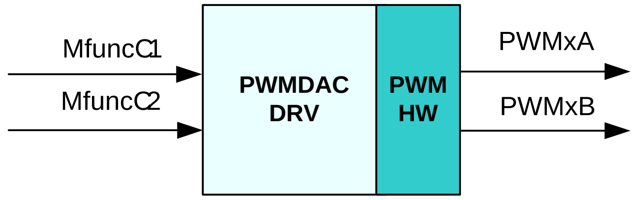
Graph Property Dialog

Display Type	Dual Time
Graph Title	Channel 3 & 4
Interleaved Data Sources	No
Start Address - upper display	DLOG_4CH_buff3
Start Address - lower display	DLOG_4CH_buff4
Page	Data
Acquisition Buffer Size	0x400
Index Increment	1
Display Data Size	0x400
DSP Data Type	16-bit signed integer
Q-value	15
Sampling Rate (Hz)	20000
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s
Status Bar Display	On
Magnitude Display Scale	Linear
Data Plot Style	Line
Grid Style	Zero Line
Cursor Mode	Data Cursor

OK Cancel Help

Description

This module converts any s/w variables into the PWM signals in EPWMxA/B for 2833x. Thus, it can be used to view the signal, represented by the variable, at the outputs of the PWMxA, PWMxB, pins through the external low-pass filters.



Availability

C interface version

Module Properties

Type: Target Dependent, Application Independent

Target Devices: 28x Floating Point

IQmath library files for C: IQmathLib.h, IQmath.lib

C Version File Names: f2833xpwmdac.h (for x2833x)

C Interface

Object Definition

The structure of PWMDAC object is defined by following structure definition

```
typedef struct {
    _iq MfuncC1;              // Input: PWM 1 Duty cycle ratio (Q24)
    _iq MfuncC2;              // Input: PWM 2 Duty cycle ratio (Q24)
  Uint16 PeriodMax;           //  Parameter: PWMDAC half period in number of clocks  (Q0)
  Uint16 HalfPerMax;          // Parameter: Half of PeriodMax
  } PWMDAC;
```

Item	Name	Description	Format	Range(Hex)
Inputs	MfuncCx	PWM duty cycle ratio	Q24	$(-2^{24},2^{24})$
Outputs	PWMxA/B	Output signals from the PWMxA/B pins	N/A	0-3.3 V
PWMDAC parameter	PeriodMax	PWMDAC half period in number of clocks	Q0	8000-7FFF
	HalfPerMax	Half of PeriodMax	Q0	8000-7FFF

Special Constants and Data types

PWMDAC

The module definition is created as a data type. This makes it convenient to instance an interface to the PWMDAC driver. To create multiple instances of the module simply declare variables of type PWMDAC.

PWMDAC_DEFAULTS

Structure symbolic constant to initialize PWMDAC module. This provides the initial values to the terminal variables as well as method pointers.

This default definition of the object implements two methods – the initialization and the runtime compute macro for PWMDAC generation. This is implemented by means of a macro pointer, and the initializer sets this to PWMDAC_INIT_MACRO and PWMDAC_MACRO macros for x2833x. The argument to this macro is the address of the PWMDAC object.

Module Usage

Instantiation

The following example instances one PWMDAC object

```
PWMDAC  pwmdac1;
```

Initialization

To Instance pre-initialized objects

```
PWMDAC pwmdac1 = PWMDAC_DEFAULTS;
```

Invoking the computation macro

```
PWMDAC_INIT_MACRO(pwmdac1);
PWMDAC_MACRO(pwmdac1);
```

Example

The following pseudo code provides the information about the module usage.

```
main()
{
  pwmdac1.PeriodMax=2500;                  // @150Mhz clock freq, PWM freq = 30kHz
  PWMDAC_INIT_MACRO (6, pwmdac1)           // PWM 6A,6B
}

void interrupt periodic_interrupt_isr()
{
   pwmdac1.MfuncC1 = variable1;     // variable1 is in GLOBAL_Q
   pwmdac1.MfuncC2 = variable2;     // variable2 is in GLOBAL_Q
   PWMDAC_MACRO (6,pwmdac1)  // update macro for pwmdac1 for PWM ch #6
}
```

Technical Background

The external low-pass filters are necessary to view the actual signal waveforms as seen in Figure 1. The (1^{st}-order) RC low-pass filter can be simply used for filter out the high frequency component embedded in the actual low frequency signals. To select R and C values, its time constant can be expressed in term of cut-off frequency (f_c) as follow:

$$\tau = RC = \frac{1}{2\pi f_c} \qquad (1)$$

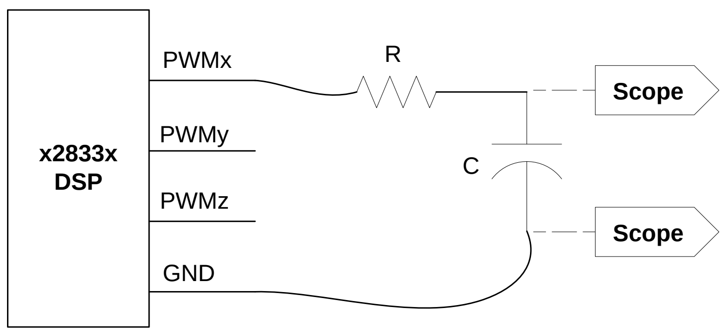
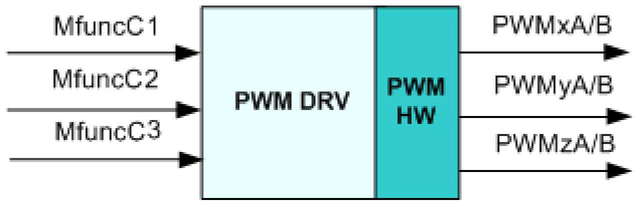


Figure 1: External RC low-lass filter connecting to PWMx pin in x2833x DSP

Description This module uses the duty ratio information and calculates the compare values for generating PWM outputs. The compare values are used in the full compare EPWM unit in 2833x. This also allows PWM period modulation.



Availability C interface version

Module Properties **Type:** Target Dependent, Application Independent

Target Devices: 28x Floating Point

IQmath library files for C: IQmathLib.h, IQmath.lib

C Version File Names: f2833xpwm.h (for x2833x)

C Interface

Object Definition

The structure of PWMGEN object is defined by following structure definition

```
typedef struct {  Uint16 PeriodMax;     // Parameter: PWM Half-Period in CPU clock cycles (Q0)
                  Uint16 HalfPerMax;    // Parameter: Half of PeriodMax (Q0)
                  Uint16 Deadband;      // Parameter: PWM deadband in CPU clock cycles (Q0)
                  _iq MfuncC1;          // Input: PWM 1 Duty cycle ratio (Q24)
                  _iq MfuncC2;          // Input: PWM 2 Duty cycle ratio (Q24)
                  _iq MfuncC3;          // Input: PWM 3 Duty cycle ratio (Q24)
                } PWMGEN;
```

Item	Name	Description	Format	Range(Hex)
Inputs	MfuncCx	PWM duty cycle ratio	Q24	$(-2^{24},2^{24})$
Outputs	PWMx	Output signals from the 6 PWM pins	N/A	0-3.3 V
PWMGEN parameter	PeriodMax	PWM Half-Period in CPU clock cycles	Q0	8000-7FFF
	HalfPerMax	Half of PeriodMax	Q0	8000-7FFF
	Deadband	PWM deadband in CPU clock cycles	Q0	8000-7FFF

Special Constants and Data types

PWMGEN

The module definition is created as a data type. This makes it convenient to instance an interface to the PWMGEN driver. To create multiple instances of the module simply declare variables of type PWMGEN.

PWMGEN_DEFAULTS

Structure symbolic constant to initialize PWMGEN module. This provides the initial values to the terminal variables as well as method pointers.

This default definition of the object implements two methods – the initialization and the runtime compute macro for PWMGEN generation. This is implemented by means of a pointer, and the initializer sets this to PWM_INIT_MACRO and PWM_MACRO macros for x2833x. The argument to this macro is the address of the PWMGEN object.

Module Usage

Instantiation
The following example instances one PWMGEN object
PWMGEN pwm1;

Initialization
To Instance pre-initialized object
PWMGEN pwm1 = PWMGEN_DEFAULTS;

Invoking the computation macro
PWM_INIT_MACRO (pwm1);
PWM_MACRO (pwm1);

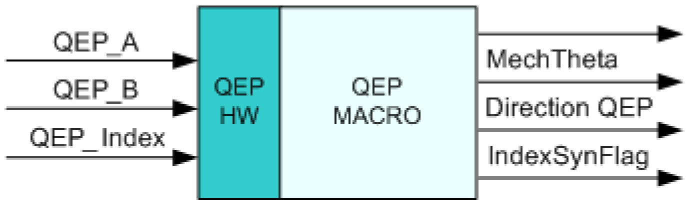
Example

The following pseudo code provides the information about the module usage.

```
main()
{

   pwm1.PeriodMax = 7500;       // PWM frequency = 10 kHz, clock = 150 MHz
   pwm1.HalfPerMax = pwm1.PeriodMax/2;
   PWM_INIT_MACRO (pwm1); // Call init macro for pwm1

}

void interrupt periodic_interrupt_isr()
{

        pwm1.MfuncC1 = svgen_dq1.Ta;    // svgen_dq1.Ta is in GLOBAL_Q
        pwm1.MfuncC2 = svgen_dq1.Tb;    // svgen_dq1.Tb is in GLOBAL_Q
        pwm1.MfuncC3 = svgen_dq1.Tc;    // svgen_dq1.Tc is in GLOBAL_Q
        PWM_MACRO (1,2,3,pwm1);         // Call update macro for pwm1 for PWM ch #1,2,3

}
```

Description

This module determines the rotor position and generates a direction (of rotation) signal from the shaft position encoder pulses.



Availability

C interface version

Module Properties

Type: Target Dependent, Application Independent

Target Devices: 28x Floating Point

IQmath library files for C: IQmathLib.h, IQmath.lib

C Version File Names: f2833xqep.h (for x2833x)

C Interface

Object Definition

The structure of QEP object is defined by following structure definition

```
typedef struct { int32 ElecTheta;          // Output: Motor Electrical angle (Q24)
                 int32 MechTheta;          // Output: Motor Mechanical Angle (Q24)
                 Uint16 DirectionQep;      // Output: Motor rotation direction (Q0)
                 Uint16 QepPeriod;         // Output: Capture period of QEP signal (Q0)
                 Uint32 QepCountIndex;     // Variable: Encoder counter index (Q0)
                 int32 RawTheta;           // Variable: Raw angle from EQEP position counter (Q0)
                 Uint32 MechScaler;        // Parameter: 0.9999/total count  (Q30)
                 Uint16 LineEncoder;       // Parameter: Number of line encoder (Q0)
                 Uint16 PolePairs;         // Parameter: Number of pole pairs (Q0)
                 int32 CalibratedAngle;    // Parameter: Raw offset between encoder and ph-a (Q0)
                 Uint16 IndexSyncFlag;     // Output: Index sync status (Q0)
               }  QEP;
```

Item	Name	Description	Format	Range(Hex)
Inputs	QEP_A	QEP_A signal applied to QEP1-A	N/A	0-3.3 v
	QEP_B	QEP_A signal applied to QEP1-B	N/A	0-3.3 v
	QEP_Index	QEP_Index signal applied to QEP1-I	N/A	0-3.3 v
Outputs	ElecTheta	Motor Electrical angle	Q24 (28xx)	00000000-7FFFFFFF (0 – 360 degree)
	MechTheta	Motor Mechanical Angle	Q24 (28xx)	00000000-7FFFFFFF (0 – 360 degree)
	DirectionQep	Motor rotation direction	Q0	0 or 1
	IndexSyncFlag	Index sync status	Q0	0 or F0
QEP parameter	MechScaler*	MechScaler = 1/total count, total count = 4*no_lines_encoder	Q30	00000000-7FFFFFFF
	PolePairs	Number of pole pairs	Q0	1,2,3,…
	CalibratedAngle	Raw offset between encoder and phase a	Q0	8000-7FFF
Internal	QepCountIndex	Encoder counter index	Q0	8000-7FFF
	RawTheta	Raw angle from Timer 2	Q0	8000-7FFF

*MechScaler in Q30 is defined by a 32-bit word-length

Special Constants and Data types

QEP

The module definition is created as a data type. This makes it convenient to instance an interface to the QEP driver. To create multiple instances of the module simply declare variables of type QEP.

QEP_DEFAULTS

Structure symbolic constant to initialize QEP module. This provides the initial values to the terminal variables as well as method pointers.

This default definition of the object implements three methods – the initialization, the runtime compute, and interrupt macros for QEP generation. This is implemented by means of a macro pointer, and the initializer sets this to QEP_INIT_MACRO, and QEP_MACRO macros for x2833x. The argument to this macro is the address of the QEP object.

Module Usage

Instantiation
The following example instances one QEP object
QEP qep1;

Initialization
To Instance pre-initialized objects
QEP qep1 = QEP_DEFAULTS;

Invoking the computation macro
QEP_INIT_MACRO (1, qep1);
QEP_MACRO (1, qep1);

The index event handler resets the QEP counter, and synchronizes the software/hardware counters to the index pulse. Also it sets the QEP IndexSyncFlag variable to reflect that an index sync has occurred.

The index handler is invoked in an interrupt service routine. Of course the system framework must ensure that the index signal is connected to the correct pin and the appropriate interrupt is enabled and so on.

Example
The following pseudo code provides the information about the module usage.

```
main()
{

    QEP_INIT_MACRO(1,qep1);             // Call init macro for qep1

}

void interrupt periodic_interrupt_isr()
{

    QEP_MACRO(1,qep1);          // Call compute macro for qep1

}
```

Technical Background

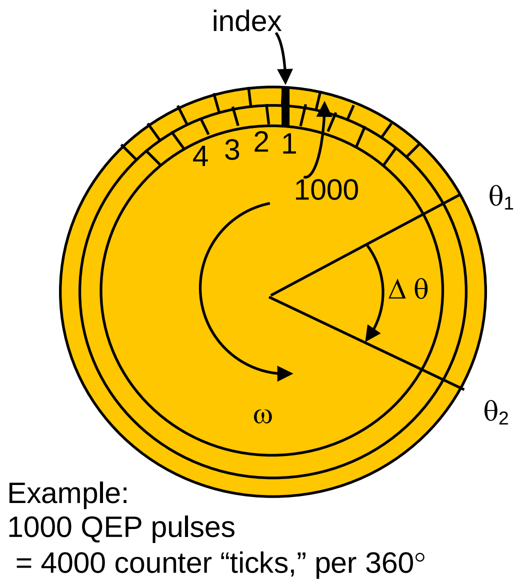


Figure 1. Speed sensor disk

Figure 1 shows a typical speed sensor disk mounted on a motor shaft for motor speed, position and direction sensing applications. When the motor rotates, the sensor generates two quadrature pulses and one index pulse. These signals are shown in Figure 2 as QEP_A, QEP_B and QEP_index.

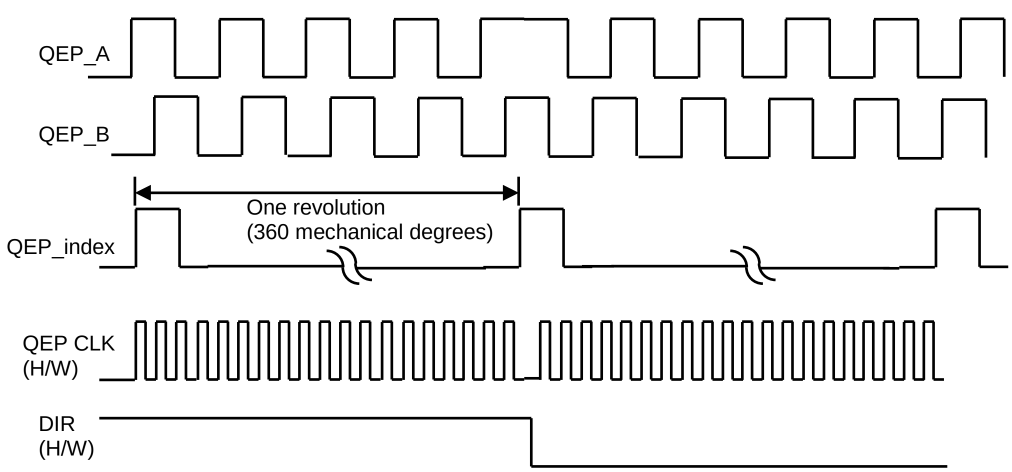


Figure 2. Quadrature encoder pulses, decoded timer clock and direction signal.

For the x280x and 2833x devices, QEP_A and QEP_B signals are applied to the EQEP1A and EQEP1B pins, respectively. The QEP_index signal is applied to the EQEP1I pin. And the position counter is obtained by QPOSCNT register.

Now the number of pulses generated by the speed sensor is proportional to the angular displacement of the motor shaft. In Figure 1, a complete 360° rotation of motor shaft generates 1000 pulses of each of the signals QEP_A and QEP_B. The QEP circuit in 281x counts both edges of the two QEP pulses. Therefore, the frequency of the counter clock, QEP_CLK, is four times that of each input sequence. This means, for 1000 pulses for each of QEP_A and QEP_B, the number of counter clock cycles will be 4000. Since the counter value is proportional to the number of QEP pulses, therefore, it is also proportional to the angular displacement of the motor shaft.

For the x280x and 2833x devices, the QDF bit in QEOSTS register is used to check the rotational direction. The index pulse resets the timer counter T2CNT and sets the index synchronization flag *IndexSyncFlag* to 00F0. Thus the counter T2CNT gets reset and starts counting the QEP_CLK pulses every time a QEP_index high pulse is generated.

To determine the rotor position at any instant of time, the counter value(T2CNT) is read and saved in the variable *RawTheta.* This value indicates the clock pulse count at that instant of time. Therefore, *RawTheta* is a measure of the rotor mechanical displacement in terms of the number of clock pulses. From this value of *RawTheta*, the corresponding per unit mechanical displacement of the rotor, *MechTheta*, is calculated as follows:

Since the maximum number of clock pulses in one revolution is 4000, i.e., maximum count value is 4000, then a coefficient, *MechScaler*, can be defined as,

$$
\begin{aligned}
& MechScaler \times 4000 = 360^0\, mechanical = 1\ per\ unit(pu)\ mechanical\ displacement \\
& \Rightarrow MechScaler = (1/4000)\ pu\ mech\ displacement / count \\
& = 16777\ pu\ mech\ displacement / count (in\ Q30)
\end{aligned}
$$

Then, the pu mechanical displacement, for a count value of *RawTheta*, is given by,

$$MechTheta = MechScaler \times RawTheta$$

If the number of pole pair is *polepairs*, then the pu electrical displacement is given by,

$$ElecTheta = PolePairs \times MechTheta$$

IMPORTANT NOTICE