Digital Motor Control

Software Library

Digital Control Systems (DCS) Group

SPRU485A
August 2001
Revised October 2003

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Contents

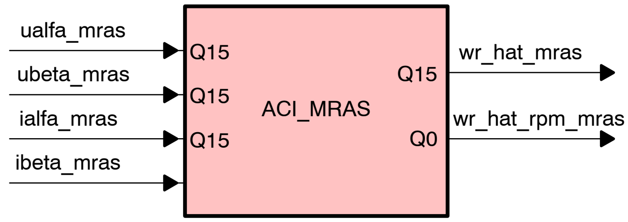
Digital Motor Control Software Library

The Digital Motor Control Software Library is a collection of digital motor control (DMC) software modules (or functions). These modules allow users to quickly build, or customize, their own systems. The Library supports the three motor types: ACI, BLDC, PMSM, and comprises both peripheral dependent (software drivers) and TMS320C24xx™ CPU-only dependent modules.

- The features of the Digital Motor Control Software Library are:
- Complete working software
- Majority offered both in Assembly and in “CcA” (C callable assembly)
- CcA modules are xDAIS-ready
- Fully documented usage and theory
- Used to build the DMC reference systems

Description

This software module implements a speed estimator for the 3-ph induction motor based on reactive power model reference adaptive system (MRAS). In this technique, there are two subsystems called reference and adaptive models, which compute the reactive power of the induction motor. Since both pure integrators and stator resistance are not associated in the reference model, the reactive power MRAS is independent of initial conditions and insensitive to variation of stator resistance.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: aci_mras.asm

ASM Routines: ACI_MRAS, ACI_MRAS_INIT

Parameter calculation excel file: aci_mras_init.xls

C-callable ASM filenames: aci_mras.asm, aci_mras.h

Item	ASM Only	C-Callable ASM	Comments
Code size	416 words	495 words†	
Data RAM	43 words	0 words†	
xDAIS module	No	No	
xDAIS component	No	No	
Multiple instances	No	Yes	

† Each pre-initialized ACIMRAS structure instance consumes 33 words in the data memory and 35 words in the .cinit section.

Direct ASM Interface

Table 1. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	ualfa_mras	Stationary alfa-axis stator voltage (pu)	Q15	−1 –> 0.999
	ubeta_mras	Stationary beta-axis stator voltage (pu)	Q15	−1 –> 0.999
	ialfa_mras	Stationary alfa-axis stator current (pu)	Q15	−1 –> 0.999
	ibeta_mras	Stationary beta-axis stator current (pu)	Q15	−1 –> 0.999
Outputs	wr_hat_mras	Estimated rotor speed (pu)	Q15	−1 –> 0.999
	wr_hat_rpm_mras	Estimated rotor speed (rpm)	Q0	−32768 –> 32767
Init / Config†	K1	K1 = (Ls−Lm^2/Lr)*Ib/(T*Vb)	Q10	−32–> 31.999
	K2	K2 = Lm^2*Ib/(Lr*Tr*Vb)	Q15	−1 –> 0.999
	K3	K3 = Tr*Wb	Q8	−128 –> 127.996
	K4	K4 = (Wb*T)^2/2	Q15	−1 –> 0.999
	K5	K5 = 1−T/Tr+T^2/(2*Tr^2)	Q15	−1 –> 0.999
	K6	K6 = Wb*(T−T^2/Tr)	Q15	−1 –> 0.999
	K7	K7 = T/Tr−T^2/(2*Tr^2)	Q15	−1 –> 0.999
	base_rpm	base_rpm = 120*base_freq/no_poles	Q3	−4096 –> 4095.9

† These constants are computed using the machine parameters (Ls, Lr, Lm, Tr), base quantities (Ib, Vb, Wb), and sampling period (T).

Routine names and calling limitation:
There are two routines involved:

ACI_MRAS, the main routine; and
ACI_MRAS_INIT, the initialization routine.

The initialization routine must be called during program initialization. The ACI_MRAS routine must be called in the control loop.

Variable Declaration:
In the system file, including the following statements before calling the subroutines:

```
.ref   ACI_MRAS, ACI_MRAS_INIT         ; Function calls
.ref   wr_hat_mras, wr_hat_rpm_mras    ; Outputs
.ref   ualfa_mras, ubeta_mras          ; Inputs
.ref   ialfa_mras, ibeta_mras          ; Inputs
```

Memory map:
All variables are mapped to an uninitialized named section, mras_aci, which can be allocated to any one data page.

Example:

During system initialization specify the ACI_MRAS parameters as follows:

```
LDP    #K1
SPLK   #K1_,K1              ; K1 = (Ls-Lm^2/Lr)*Ib/(T*Vb)  (Q10)
SPLK   #K2_,K2              ; K2 = Lm^2*Ib/(Lr*Tr*Vb)      (Q15)
SPLK   #K3_,K3              ; K3 = Tr*Wb                   (Q8)
SPLK   #K4_,K4              ; K4 = (Wb*T)^2/2              (Q15)
SPLK   #K5_,K5              ; K5 = 1-T/Tr+T^2/(2*Tr^2)     (Q15)
SPLK   #K6_,K6              ; K6 = Wb*(T-T^2/Tr)           (Q15)
SPLK   #K7_,K7              ; K7 = T/Tr-T^2/(2*Tr^2)       (Q15)
SPLK   #BASE_RPM_,base_rpm  ; Base motor speed in rpm      (Q3)
```

Then in the interrupt service routine call the module and read results as follows:

```
LDP  #ualfa_mras              ; Set DP for module inputs
BLDD #input_var1,ualfa_mras   ; Pass input variables to module inputs
BLDD #input_var2,ubeta_mras   ; Pass input variables to module inputs
BLDD #input_var3,ialfa_mras   ; Pass input variables to module inputs
BLDD #input_var4,ibeta_mras   ; Pass input variables to module inputs

CALL   ACI_MRAS

LDP  #output_var1                 ; Set DP for module output
BLDD #wr_hat_mras,output_var1     ; Pass output to other variables
BLDD #wr_hat_rpm_mras,output_var2 ; Pass output to other variables
```

C/C-Callable ASM Interface

Object Definition

The structure of the ACIMRAS object is defined in the header file, aci_mras.h, as below:

```
typedef struct{ int  ualfa_mras;      /* Input: alfa-axis phase voltage at k (Q15) */
                int  ubeta_mras;      /* Input: beta-axis phase voltage at k (Q15) */
                int  ialfa_mras;      /* Input: alfa-axis line current at k (Q15) */
                int  ibeta_mras;      /* Input: beta-axis line current at k (Q15) */
                int  ialfa_old;       /* History: alfa-axis line current at k-1 (Q15) */
                int  ibeta_old;       /* History: beta-axis line current at k-1 (Q15) */
                int  imalfa_old_high;/* History: alfa-axis magnetizing current at k-1 (Q31) */
                int  imalfa_old_low; /* History: alfa-axis magnetizing current at k-1 (Q31) */
                int  imbeta_old_high;/* History: beta-axis magnetizing current at k-1 (Q31) */
                int  imbeta_old_low; /* History: beta-axis magnetizing current at k-1 (Q31) */
                int  imalfa_high;     /* Variable: alfa-axis magnetizing current at k (Q31) */
                int  imalfa_low;      /* Variable: alfa-axis magnetizing current at k (Q31) */
                int  imbeta_high;     /* Variable: beta-axis magnetizing current at k (Q31) */
                int  imbeta_low;      /* Variable: beta-axis magnetizing current at k (Q31) */
                int  ealfa;           /* Variable: alfa-axis back emf at k (Q15) */
                int  ebeta;           /* Variable: beta-axis back emf at k (Q15) */
                int  q;               /* Variable: reactive power in reference model (Q15) */
                int  q_hat;           /* Variable: reactive power in adaptive model  (Q15) */
                int  error;           /* Variable: reactive power error (Q15) */
                int  K1;              /* Parameter: constant using in reference model (Q10) */
                int  K2;              /* Parameter: constant using in adaptive model (Q15) */
                int  K3;              /* Parameter: constant using in adaptive model (Q8) */
                int  K4;              /* Parameter: constant using in adaptive model (Q15) */
                int  K5;              /* Parameter: constant using in adaptive model (Q15) */
                int  K6;              /* Parameter: constant using in adaptive model (Q15) */
                int  K7;              /* Parameter: constant using in adaptive model (Q15) */
                int  Kp;              /* Parameter: proportioanl gain  (Q15) */
                int  Ki_high;         /* Parameter: integral gain (Q31) */
                int  Ki_low;          /* Parameter: integral gain (Q31) */
                int  base_rpm;        /* Parameter: base motor speed in rpm (Q3) */
                int  wr_hat_mras;     /* Output: estimated (per-unit) motor speed (Q15) */
                int  wr_hat_rpm_mras;/* Output: estimated (rpm) motor speed (Q0) */
                int  (*calc)();       /* Pointer to calculation function */
              } ACIMRAS;
```

Special Constants and Datatypes

ACIMRAS

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

ACIMRAS_DEFAULTS

Initializer for the ACIMRAS object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, aci_mras.h.

Methods

void calc(ACIMRAS *);

This default definition of the object implements just one method – the runtime compute function for MRAS speed estimator. This is implemented by means of a function pointer, and the default initializer sets this to aci_mras_calc function. The argument to this function is the address of the ACIMRAS object. Again, this statement is written in the header file, aci_mras.h.

Module Usage

Instantiation:

The following example instances two such objects:

```
ACIMRAS  mras1, mras2;
```

Initialization:

To instance a pre-initialized object:

```
ACIMRAS mras1 = ACIMRAS_DEFAULTS;
ACIMRAS mras2 = ACIMRAS_DEFAULTS;
```

Invoking the compute function:

```
mras1.calc(&mras1);
mras2.calc(&mras2);
```

Example:

Lets instance two ACIMRAS objects, otherwise identical, and run two MRAS speed estimators. The following example is the c source code for the system file.

```
ACIMRAS mras1 = ACIMRAS_DEFAULTS;   /* instance the first object */
ACIMRAS mras2 = ACIMRAS_DEFAULTS;   /* instance the second object */

main()
{

  mras1.ualfa_mras=volt1.Vdirect;     /* Pass inputs to mras1 */
  mras1.ubeta_mras=volt1.Vquadra;     /* Pass inputs to mras1 */
  mras1.ialfa_mras=current_dq1.d;     /* Pass inputs to mras1 */
  mras1.ibeta_mras=current_dq1.q;     /* Pass inputs to mras1 */

  mras2.ualfa_mras=volt2.Vdirect;     /* Pass inputs to mras2 */
  mras2.ubeta_mras=volt2.Vquadra;     /* Pass inputs to mras2 */
  mras2.ialfa_mras=current_dq2.d;     /* Pass inputs to mras2 */
  mras2.ibeta_mras=current_dq2.q;     /* Pass inputs to mras2 */

}

void interrupt periodic_interrupt_isr()
{

  mras1.calc(&mras1);          /* Call compute function for mras1 */
  mras2.calc(&mras2);          /* Call compute function for mras2 */

  speed_pu1=mras1.wr_hat_mras;     /* Access the outputs of mras1 */
  speed_rpm1=mras1.wr_hat_rpm_mras;/* Access the outputs of mras1 */
  speed_pu2=mras2.wr_hat_mras;     /* Access the outputs of mras2 */
  speed_rpm2=mras2.wr_hat_rpm_mras;/* Access the outputs of mras2 */

}
```

Background Information

The reactive power MRAS speed estimator is shown in Figure 1. The information required for this module is stator voltages and stator current components in the $\alpha-\beta$ stationary reference frame. Two sets of equations are developed to compute reactive power of induction motor in the reference and adaptive models. The reference model does not involve the rotor speed while the adaptive model needs the estimated rotor speed to adjust the computed reactive power to that computed from the reference model. The system stability had been proved by Popov's hyperstability theorem [1]−[2]. The equations for the reactive power in both models can be derived in the continuous and discrete time domains, as shown below. Notice that the representation of complex number is defined for the stator voltages and currents in the stationary reference frame, i.e., $\bar{v}_s = v_{s\alpha} + jv_{s\beta}$ and $\bar{i}_s = i_{s\alpha} + ji_{s\beta}$.

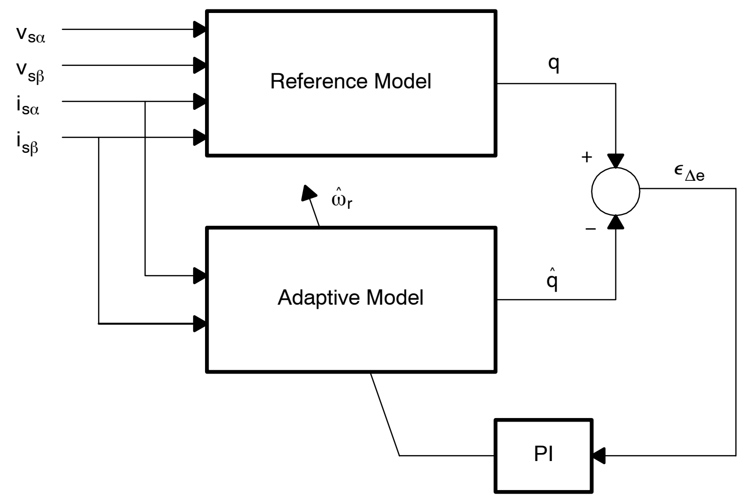


Figure 1. The Simplified Block Diagram of Reactive Power MRAS Speed Estimator

Continuous time representation

Reference model

The back emf of Induction motor can be expressed in the stationary reference frame as follows:

$$\hat{e}_{(s\alpha)} = \frac{L_m}{L_r}\frac{\left(d\psi_{(r\alpha)}\right)}{dt} = v_{(s\alpha)} - R_s i_{(s\alpha)} - \sigma L_s \frac{di_{(s\alpha)}}{dt} \tag{1}$$

$$\hat{e}_{(s\beta)} = \frac{L_m}{L_r}\frac{\left(d\psi_{(r\beta)}\right)}{dt} = v_{(s\beta)} - R_s i_{(s\beta)} - \sigma L_s \frac{di_{(s\beta)}}{dt} \tag{2}$$

$$\bar{e} = e_{(s\alpha)} + je_{(s\beta)} \tag{3}$$

The reactive power of the Induction motor can be computed from cross product of stator currents and back emf vectors as follows:

$$q = \bar{i}_s \times \bar{e} = \bar{i}_s \times \left(\bar{v}_s - R_s\bar{i}_s - \sigma L_s \frac{d\bar{i}_s}{dt}\right) = \bar{i}_s \times \bar{v}_s - \bar{i}_s \times \sigma L_s \frac{d\bar{i}_s}{dt} \tag{4}$$

where $\bar{i}_s \times \bar{i}_s = i_{s\alpha} i_{s\beta} - i_{s\beta} i_{s\alpha} = 0$ and $\sigma = 1 - \frac{L_m^2}{L_s L_r}$ (leakage coefficient)

As a result, the reactive power shown in (4) can be further derived as

$$q = i_{s\alpha} v_{s\beta} - i_{s\beta} v_{s\alpha} - \sigma L_s \left(i_{s\alpha} \frac{di_{s\beta}}{dt} - i_{s\beta} \frac{di_{s\alpha}}{dt} \right) \tag{5}$$

Adaptive model

The estimated back emf computed in the adaptive model can be expressed as follows:

$$\hat{e}_{(s\alpha)} = \frac{(L^2)_m}{L_r} \frac{di_{(m\alpha)}}{dt} = \frac{(L^2)_m}{(L_r \tau_r)} \left(-\tau_r \hat{\omega} i_{(m\beta)} - i_{(m\alpha)} + i_{(s\alpha)} \right) \tag{6}$$

$$\hat{e}_{(s\beta)} = \frac{(L^2)_m}{L_r} \frac{di_{(m\beta)}}{dt} = \frac{(L^2)_m}{(L_r \tau_r)} \left(-\tau_r \hat{\omega} i_{(m\alpha)} - i_{(m\beta)} + i_{(s\beta)} \right) \tag{7}$$

$$\hat{\bar{e}} = \hat{e}_{(s\alpha)} + j\hat{e}_{(s\beta)} \tag{8}$$

where $\tau_r = \frac{L_r}{R_r}$ is rotor time constant, and $i_{m\alpha}$, $i_{m\beta}$ are computed from the following equations:

$$\frac{di_{m\alpha}}{dt} = -\hat{\omega}_r i_{m\beta} - \frac{1}{\tau_r} i_{m\alpha} + \frac{1}{\tau_r} i_{s\alpha} \tag{9}$$

$$\frac{di_{m\beta}}{dt} = \hat{\omega}_r i_{m\alpha} - \frac{1}{\tau_r} i_{m\beta} + \frac{1}{\tau_r} i_{s\beta} \tag{10}$$

Once the estimated back emf, $\hat{\bar{e}}$, computed by using (6)–(10), the estimated reactive power can be computed as follows:

$$\hat{q} = \bar{i}_s \times \hat{\bar{e}} = i_{s\alpha} \hat{e}_{s\beta} - i_{s\beta} \hat{e}_{s\alpha} \tag{11}$$

Then, the PI controller tunes the estimated rotor speed, $\hat{\omega}_r$, such that the reactive power generated by adaptive model, $\hat{q}$, matches that generated by reference model, q. The speed tuning signal, $\epsilon_{\Delta e}$, is the error of reactive power that can be expressed as follows:

$$\varepsilon_{\Delta e} = \bar{i}_s \times (\bar{e} - \hat{\bar{e}}) = q - \hat{q} \tag{12}$$

Discrete time representation

For implementation on DSP based system, the differential equations need to be transformed to difference equations. Due to high sampling frequency compared to bandwidth of the system, the simple approximation of numerical integration, such as forward, backward, or trapezoidal rules, can be adopted. Consequently, the reactive power equations in both reference and adaptive models are discretized as follows:

Reference model

According to (5), using backward approximation, then

$$\begin{aligned} q(k) &= i_{s\alpha}(k) v_{s\beta}(k) - i_{s\beta}(k) v_{s\alpha}(k) - \\ &\sigma L_s \left(i_{s\alpha}(k) \frac{i_{s\beta}(k) - i_{s\beta}(k-1)}{T} - i_{s\beta}(k) \frac{i_{s\alpha}(k) - i_{s\alpha}(k-1)}{T} \right) \end{aligned} \tag{13}$$

Equation (13) can be further simplified as.

$$q(k) = i_{s\alpha}(k)v_{s\beta}(k) - i_{s\beta}(k)v_{s\alpha}(k) - \frac{\sigma L_s}{T}\left(i_{s\beta}(k)i_{s\alpha}(k-1) - i_{s\alpha}(k)i_{s\beta}(k-1)\right) \tag{14}$$

where T is the sampling period

Adaptive model

According to (11),

$$\hat{q}(k) = i_{(s\alpha)}(k)\hat{e}_{(s\beta)}(k) - i_{(s\beta)}(k)\hat{e}_{(s\alpha)}(k) \tag{15}$$

where $\hat{e}_{s\beta}(k), \hat{e}_{s\alpha}(k)$ are computed as follows:

$$\hat{e}_{(s\alpha)}(k) = \frac{(L^2)_m}{(L_r\tau_r)}\Big(-\tau_r\hat{\omega}_r(k)i_{(\beta)}(k) - i_{(m\alpha)}(k) + i_{(s\alpha)}(k)\Big) \tag{16}$$

$$\hat{e}_{(s\beta)}(k) = \frac{(L^2)_m}{(L_r\tau_r)}\Big(-\tau_r\hat{\omega}_r(k)i_{(\alpha)}(k) - i_{(m\beta)}(k) + i_{(s\beta)}(k)\Big) \tag{17}$$

and $i_{m\alpha}(k)$, $i_{m\beta}(k)$ can be solved by using trapezoidal integration method, it yields

$$\begin{aligned} i_{(m\alpha)}(k) = {} & i_{(m\alpha)}(k-1)\left[-\left(\frac{T^2}{2}\right)\left(\hat{\omega}^2\right)_r(k) + 1 - \left(\frac{T}{\tau_r}\right) + \left(\frac{T^2}{(\tau^2)_r}\right)\right] - \\ & i_{(m\beta)}(k-1)\hat{\omega}_r(k)\left[T - \frac{T^2}{\tau_r}\right] + \\ & i_{(s\alpha)}(k)\left[\frac{T}{\tau_r} - \frac{T^2}{\left(2(\tau^2)_r\right)}\right] - \\ & i_{(s\beta)}(k)\hat{\omega}_r(k)\left[\frac{T^2}{(2\tau_r)}\right] \end{aligned} \tag{18}$$

$$\begin{aligned} i_{(m\beta)}(k) = {} & i_{(m\beta)}(k-1)\left[\frac{T^2}{2}\left(\hat{\omega}^2\right)_r(k) + 1 - \frac{T}{\tau_r} + \frac{T^2}{\left(2(\tau^2)_r\right)}\right] + \\ & i_{(m\alpha)}(k-1)\hat{\omega}_r(k)\left[T - \frac{T^2}{\tau_r}\right] + \\ & i_{(s\beta)}(k)\left[\frac{T}{\tau^r} - \frac{T^2}{(2\tau_r)}\right] + \\ & i_{(s\alpha)}(k)\hat{\omega}_r(k)\left[\frac{T^2}{(2\tau_r)}\right] + \end{aligned} \tag{19}$$

Per unit, discrete time representation

For the sake of generality, the per unit concept is used in all equations. However, for simplicity, the same variables are also used in the per unit representations.

Reference model

Dividing (14) by base power of V_bI_b, then its per unit representation is as follows:

$$q(k) = i_{s\alpha}(k)v_{s\beta}(k) - i_{s\beta}(k)v_{s\alpha}(k) - K_1\left(i_{s\beta}(k)i_{s\alpha}(k-1) - i_{s\alpha}(k)i_{s\beta}(k-1)\right) \text{ pu} \tag{20}$$

Rearranging (20), then another form can be shown

$$q(k) = i_{s\alpha}(k)\left(v_{s\beta}(k) - K_1 i_{s\beta}(k-1)\right) - i_{s\beta}(k)\left(v_{s\alpha}(k) + K_1 i_{s\alpha}(k-1)\right) \text{ pu} \tag{21}$$

where $K_1 = \frac{\sigma L_s I_b}{T V_b}$, V_b is base voltage, and I_b is base current.

Adaptive model

Dividing (16) and (17) by base voltage V_b, then yields

$$\hat{e}_{s\alpha}(k) = K_2\left(-K_3 \hat{\omega}_r(k) i_{m\beta}(k) - i_{m\alpha}(k) + i_{s\alpha}(k)\right) \text{ pu} \tag{22}$$

$$\hat{e}_{s\beta}(k) = K_2\left(K_3 \hat{\omega}_r(k) i_{m\alpha}(k) - i_{m\beta}(k) + i_{s\beta}(k)\right) \text{ pu} \tag{23}$$

where $K_2 = \frac{L_m^2 I_b}{L_r \tau_r V_b}$, $K_3 = \tau_r \omega_b = \frac{L_r \omega_b}{R_r}$, and $\omega_b = 2\pi f_b$ is base electrically angular velocity. Similarly, dividing (18)–(19) by base current I_b, then yields

$$i_{(m\alpha)}(k) = i_{(m\alpha)}(k-1)\left(-K_4\left(\hat{\omega}^2\right)_r(k) + K_5\right) - i_{(m\beta)}(k-1)\hat{\omega}_r(k)K_6 +_{pu} i_{(s\alpha)}(k)K_7 - i_{(s\beta)}(k)\hat{\omega}_r(k)K_8 \tag{24}$$

$$i_{(m\beta)}(k) = i_{(m\beta)}(k-1)\left(-K_4\left(\hat{\omega}^2\right)_r(k) + K_5\right) - i_{(m\alpha)}(k-1)\hat{\omega}_r(k)K_6 +_{pu} i_{(s\beta)}(k)K_7 - i_{(s\alpha)}(k)\hat{\omega}_r(k)K_8 \tag{25}$$

where $K_4 = \frac{\omega_b^2 T^2}{2}$, $K_5 = 1 - \frac{T}{\tau_r} + \frac{T^2}{2\tau_r^2}$, $K_6 = \omega_b\left(T - \frac{T^2}{\tau_r}\right)$, $K_7 = \frac{T}{\tau_r} - \frac{T^2}{2\tau_r^2}$, and $K_8 = \omega_b \frac{T^2}{2\tau_r}$. After $i_{m\alpha}(k)$ and $i_{m\beta}(k)$ in per unit are calculated from (24) and (25), the back emf in per unit can also be computed by using (22) and (23), and then the per unit estimated reactive power in adaptive model can be simply calculated from (15).

Notice that the K8 is practically ignored because it is extremely small. The excel file aci_mras_init.xls is used to compute these seven constants (i.e., K1,0,K7) in the appropriately defined Q system. This file can directly compute the hexadecimal/decimal values of these K's in order to put them into the ACI_MRAS_INIT module easily. The PI controller gains Kp and Ki are also translated into the hexadecimal/decimal values in this excel file. Moreover, the base motor speed is computed in the hexadecimal/decimal values as well. The required parameters for this module are summarized as follows:

The machine parameters:

- number of poles
- rotor resistance (R_r)
- stator leakage inductance (L_{sl})
- rotor leakage inductance (L_{rl})
- magnetizing inductance (L_m)

The based quantities:

- base current (I_b)
- base voltage (V_b)
- base electrically angular velocity (ω_b)
- The sampling period:
- sampling period (T)

Notice that the rotor self inductance is $L_r = L_{rl} + L_m$, and the stator self inductance is $L_s = L_{sl} + L_m$.

Next, Table 2 shows the correspondence of notations between variables used here and variables used in the program (i.e., aci_mras.asm). The software module requires that both input and output variables are in per unit values (i.e., they are defined in Q15).

Table 2. Correspondence of Notations

	Equation Variables	Program Variables
Inputs	$v_{s\alpha}$	ualfa_mras
	$v_{s\beta}$	ubeta_mras
	$i_{s\alpha}$	ialfa_mras
	$i_{s\beta}$	ibeta_mras
Outputs	$\hat{\omega}_r$	wr_hat_mras
Others	$\hat{e}_{s\alpha}$	ealfa
	$\hat{e}_{s\beta}$	ebeta
	$i_{m\alpha}$	imalfa_high, imalfa_low
	$i_{m\beta}$	imbeta_high, imbeta_low
	q	q
	$\hat{q}$	q_hat
	$\epsilon_{\Delta e}$	error

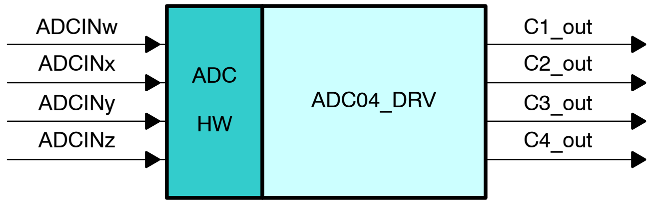
References:

3) P. Vas, *Sensorless Vector and Direct Torque Control*, Oxford University Press, 1998.

4) F-Z Peng and T. Fukao, “Robust speed identification for speed-sensorless vector control of Induction motors”, *IEEE Trans. Ind. Appl.*, vol. 30, no. 5, pp. 1234–1240, 1994.

Description

This module performs 4-channel AD conversion on bipolar signals. The channels are specified by *A4_ch_sel*.



Availability

This module is available in the direct-mode assembly-only interface (Direct ASM).

Module Properties

Type: Target dependent, Application dependent

Target Devices: x24x/x24xx

Assembly File Name: adc04drv.asm

Item	ASM Only	Comments
Code size	101 words	
Data RAM	15 words	
xDAIS module	No	
xDAIS component	No	IALG layer not implemented

Direct ASM Interface

Table 3. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	ADCINw/x/y/z	ADC pins in 24x/24xx device. w,x,y,z correspond to the channel numbers selected by A4_ch_sel	N/A	N/A
Outputs	Cn_out (n=1,2,3,4)	Conversion result for channel corresponding to Cn	Q15	0−7FFF
Init / Config	A4_ch_sel	ADC channel select variable. Specify appropriate channels using this variable. Input format = C4C3C2C1, Ex, A4_ch_sel = FC83 implies selected channels are, Ch3 as C1, Ch8 as C2, Ch12 as C3 and Ch15 as C4.	Q0	N/A

Variable Declaration:
In the system file include the following instructions:

```
.ref   ADC04_DRV, ADC04_DRV_INIT                        ;function call
.ref   A4_ch_sel, C1_gain, C2_gain, C3_gain, C4_gain   ;input
.ref   C1_offset, C2_offset, C3_offset, C4_offset      ;input
.ref   C1_out, C2_out, C3_out, C4_out                  ;output
```

Memory Map:
All variables are mapped to an uninitialized named section ‘adc04drv’.

Example:
During system initialization specify the inputs as follows:

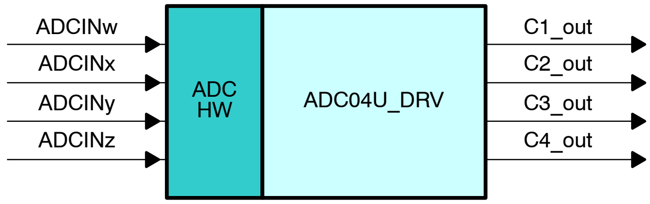
```
ldp  #A4_ch_sel           ;Set DP for module inputs
splk #04321h, A4_ch_sel   ;Select ADC channels. In this example
                          ;channels selected are 4, 3, 2, and 1.
splk #GAIN1, C1_gain      ;Specify gain value for each channel
splk #GAIN2, C2_gain
splk #GAIN3, C3_gain
splk #GAIN4, C4_gain
splk #OFFSET1, C1_offset  ;Specify offset value for each channel
splk #OFFSET2, C2_offset
splk #OFFSET3, C3_offset
splk #OFFSET4, C4_offset
```

Then in the interrupt service routine call the module and read results as follows:

```
CALL ADC04_DRV
ldp  #output_var1          ;Set DP for output variables
bldd #C1_out, output_var1  ;Pass module outputs to output variables
bldd #C2_out, output_var2
bldd #C3_out, output_var3
bldd #C4_out, output_var4
```

Description

This module performs 4-channel AD conversion on unipolar signals. The channels are specified by *A4_ch_sel* .



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: adc4udrv.asm

C-Callable Version File Names: F243ADC1.ASM, F243ADC2.ASM, F243_ADC.H, F2407ADC1.ASM, F2407ADC2.ASM, F2407ADC.H

Item	ASM Only	C-Callable ASM	Comments
Code size	93/73 words	91/71 words†	
Data RAM	11 words	0 words†	
Multiple instances	No	See note	

† Each pre-initialized ADCVALS structure instance consumes 11 words in the data memory and 13 words in the .cinit section.

Note: Multiple instances must point to distinct interfaces on the target device. Multiple instances pointing to the same ADC interface in hardware may produce undefined results. So the number of interfaces on the F241/3 is limited to one, while there can be upto two such interfaces on the LF2407.

Direct ASM Interface

Table 4. Module Terminal Variables/Functions

	Name	Description	Format	Range
H/W Inputs	ADCINw/x/y/z	ADC pins in 24x/24xx device. w,x,y,z correspond to the channel numbers selected by A4_ch_sel	N/A	N/A
Outputs	Cn_out (n=1,2,3,4)	Conversion result for channel corresponding to Cn	Q15	0–7FFF
Init / Config	A4_ch_sel	ADC channel select variable. Use this to specify appropriate ADC channels. Input format = C4C3C2C1, for example, A4_ch_sel = FC83 implies selected channels are, Ch3 as C1, Ch8 as C2, Ch12 as C3 and Ch15 as C4.	Q0	N/A
	Cn_gain (n=1,2,3,4)	Gain control for channel corresponding to Cn. Use this to adjust gain for each channel for appropriately scaled signals.	Q13	0–7FFF
	24x/24xx	Select appropriate 24x/24xx device in the x24x_app.h file.		

Variable Declaration:
In the system file include the following instructions:

```
.ref   ADC04U_DRV, ADC04U_DRV_INIT                    ;function call
.ref    A4_ch_sel, C1_gain, C2_gain, C3_gain, C4_gain ;input
.ref    C1_out, C2_out, C3_out, C4_out                ;output
```

Memory map:
All variables are mapped to an uninitialized named section 'adc4udrv'

Example:
During system initialization specify the inputs as follows:

```
ldp  #A4_ch_sel          ;Set DP for module inputs
splk #04321h, A4_ch_sel  ;Select ADC channels. In this example
                         ;channels selected are 4, 3, 2, and 1.
splk #GAIN1, C1_gain     ;Specify gain value for each channel
splk #GAIN2, C2_gain
splk #GAIN3, C3_gain
splk #GAIN4, C4_gain
```

Then in the interrupt service routine call the module and read results as follows:

```
CALL ADC04U_DRV
```

```
ldp  #output_var1          ;Set DP for output variables
bldd #C1_out, output_var1  ;Pass module outputs to output variables
bldd #C2_out, output_var2
bldd #C3_out, output_var3
bldd #C4_out, output_var4
```

C/C-Callable ASM Interface

Object Definition

The structure of the ADCVALS Interface Object is defined by the following structure definition

```
typedef struct {
       int c1_gain; /* Gain control for channel 1[Q13] */
       int c2_gain; /* Gain control for channel 2[Q13] */
       int c3_gain; /* Gain control for channel 3[Q13] */
       int c4_gain; /* Gain control for channel 4[Q13] */

       int c1_out;  /* Conversion result for channel 1[Q15]*/
       int c2_out;  /* Conversion result for channel 2[Q15]*/
       int c3_out;  /* Conversion result for channel 3[Q15]*/
       int c4_out;  /* Conversion result for channel 4[Q15]*
       int a4_ch_sel; /* ADC channel select variable[Q0] */
       int (*init)();  /* Initialization func pointer  */
       int (*update)(); /* Update function              */
       } ADCVALS;
```

Table 5. Module Terminal Variables/Functions

	Name	Description	Format	Range
H/W Inputs	ADCINw/x/y/z	ADC pins in 24x/24xx device. w,x,y,z correspond to the channel numbers selected by A4_ch_sel	N/A	N/A
Outputs	Cn_out (n=1,2,3,4)	Conversion result for channel corresponding to Cn	Q15	0−7FFF
Init / Config	A4_ch_sel	ADC channel select variable. Use this to specify appropriate ADC channels. Input format = C4C3C2C1, for example, A4_ch_sel = FC83 implies selected channels are, Ch3 as C1, Ch8 as C2, Ch12 as C3 and Ch15 as C4.	Q0	N/A
	Cn_gain (n=1,2,3,4)	Gain control for channel corresponding to Cn. Use this to adjust gain for each channel for appropriately scaled signals.	Q13	0−7FFF
	24x/24xx	Select appropriate 24x/24xx device in the x24x_app.h file.		

Special Constants and Datatypes

ADCVALS
The module definition itself is created as a data type. This makes it convenient to instance an interface to the ADC Driver module.

ADCVALS_DEFAULTS
Initializer for the ADCVALS Object. This provides the initial values to the terminal variables as well as method pointers.

ADCVALS_handle
Typedef'ed to ADCVALS *

F243_ADC_DEFAULTS
Constant initializer for the F243ADC Interface.

F2407_ADC_DEFAULTS
Constant initializer for the F2407 ADC Interface

Methods

void init (ADCVALS_handle)
Initializes the ADC Driver unit hardware.

void update(ADCVALS_handle)
Updates the ADC Driver hardware with the data from the ADCVALS Structure.

Module Usage

Instantiation:
The interface to the ADC Driver Unit is instanced thus:

```
ADCVALS  adc;
```

Initialization:
To instance a pre-initialized object

```
ADCVALS adc =ADC_DEFAULTS
```

Hardware Initialization:

```
adc.init(&adc);
```

Invoking the update function:

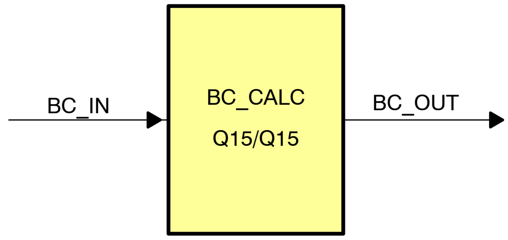
```
adc.update(&adc);
```

Example:
Lets instance one ADCVALS object

```
ADCVALS adc =ADC_DEFAULTS;

main()
{
  adc.a4_ch_sel = 0x5432 ;                /* Initialize */
  adc.c1_gain   = 0x1FFF;
  adc.c2_gain   = 0x1FFF;
  adc.c3_gain   = 0x1FFF;
  adc.c4_gain   = 0x1FFF;

  (*adc.init)(& adc);                       /* Call the function */
}
  void interrupt periodic_interrupt_isr()
{
(*adc.update)(& adc);
x = adc.c1_out;
y = adc.c2_out;
z = adc.c3_out;
p = adc.c4_out;
}
```

Description

This software module calculates the average value of a s/w variable. The output can be rescaled and the size of buffer used for storing the averaging data is selectable.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Asembly File Name: box_car.asm

ASM Routines: BC_CALC, BC_INIT

C-Callable ASM File Names: box_car.asm, box_car.h

Item	ASM Only	C-Callable ASM	Comments
Code size	47 words	46 words‡	
Data RAM	69† words	69† words‡	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† For 64-word buffer size.
‡ Each pre-initialized BOXCAR structure occupies (5+BC_SIZE) words in the data memory and (7+BC_SIZE) words in the .cinit section.

Direct ASM Interface

Table 6. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	BC_IN	Input to be averaged	Q15	−1 –> 0.999
Output	BC_OUT	Averaged output with the selectable buffer size	Q15	−1 –> 0.999
Init / Config	BC_SIZE	The buffer size	Q0	2, 4, 8, 16, …
	bc_scaler	The scaling factor	Q15	−1 –> 0.999

Routine names and calling limitation:
There are two routines involved:

BC_CALC, the main routine; and
BC_INIT, the initialization routine.

The initialization routine must be called during program initialization. The BC_CALC routine must be called in the control loop.

Variable Declaration:
In the system file, including the following statements before calling the subroutines:

```
.ref   BC_INIT, BC_CALC      ;function call
.ref   BC_IN, BC_OUT        ;Inputs/Outputs
```

Memory map:
All variables are mapped to an uninitialized named section, bc, which can be allocated to any one data page. However, the buffer data is mapped to an uninitialized named section, farmem.

Example:
In the interrupt service routine call the module and read results as follows:

```
LDP  #BC_IN                    ; Set DP for module inputs
BLDD #input_var1,BC_IN         ; Pass input variables to module inputs

CALL BC_CALC

LDP  #output_var1               ; Set DP for module output
BLDD #BC_OUT, output_var1       ; Pass output to other variables
```

C/C-Callable ASM Interface

Object Definition

The structure of the BOXCAR object is defined in the header file, box_car.h, as seen in the following:

```
#define    BC_SIZE  64

  typedef struct  { int  BC_IN;               /* Input: Box-Car input (Q15) */
                    int  BC_PTR;              /* Variable: Box-car buffer pointer */
                    int  BC_BUFFER[BC_SIZE];  /* Variable: Box-car buffer (Q15) */
                    int  BC_OUT;              /* Output: Box-car output (Q15) */
                    int  bc_scaler;           /* Parameter: Box-car scaler (Q15) */
                    int  (*calc)();           /* Pointer to calculation function */
                  } BOXCAR;
```

Special Constants and Datatypes

BOXCAR

The module definition itself is created as a data type. This makes it convenient to instance a BOXCAR object. To create multiple instances of the module simply declare variables of type BOXCAR.

BOXCAR_DEFAULTS

Initializer for the BOXCAR object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, box_car.h.

Methods

void calc(BOXCAR *);

This default definition of the object implements just one method – the runtime compute function for averaging. This is implemented by means of a function pointer, and the default initializer sets this to bc_calc function. The argument to this function is the address of the BOXCAR object. Again, this statement is written in the header file, box_car.h.

Module Usage

Instantiation:

The following example instances two such objects:

```
BOXCAR bc1, bc2;
```

Initialization:

To instance a pre-initialized object:

```
BOXCAR bc1 = BOXCAR_DEFAULTS;
BOXCAR bc2 = BOXCAR_DEFAULTS;
```

Invoking the compute function:

```
bc1.calc(&bc1);
bc2.calc(&bc2);
```

Example:

Lets instance two BOXCAR objects, otherwise identical, and compute the averaging values of two different s/w variables. The following example is the c source code for the system file.

```
BOXCAR bc1= BOXCAR_DEFAULTS;      /* instance the first object */
BOXCAR bc2= BOXCAR_DEFAULTS;      /* instance the second object */

main()
{

    bc1.BC_IN = input1;            /* Pass inputs to bc1 */
    bc2.BC_IN = input2;            /* Pass inputs to bc2 */

}

void interrupt periodic_interrupt_isr()
{

    bc1.calc(&bc1);            /* Call compute function for bc1 */
    bc2.calc(&bc2);            /* Call compute function for bc2 */

    output1 = bc1.BC_OUT;      /* Access the outputs of bc1 */
    output2 = bc2.BC_OUT;      /* Access the outputs of bc2 */

}
```

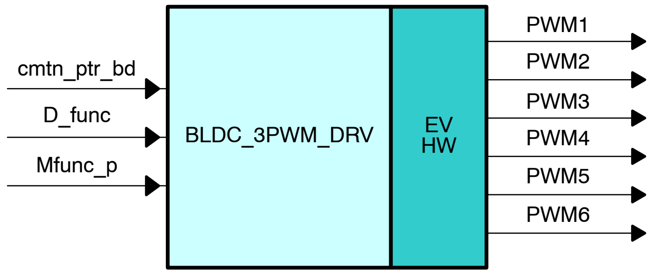
Background Information

This s/w module computes the average of the runtime values of the selected input variable. The size of the buffer used to keep the data is selectable with the power of two, i.e., 2, 4, 8, 16, 32, 64, …. The default buffer size is 64. For different buffer size modify the code (valid for both ASM and CcA versions) as required. The following instruction is added or deleted, according to the buffer size, at the location indicated in the code. This divides the number in accumulator by two.

```
SFR                      ; Number of times SFR need to be executed
                         ; is, log2(BC_SIZE)
```

Description

This module generates the 6 switching states of a 3-ph power inverter used to drive a 3-ph BLDC motor. These switching states are determined by the input variable *cmtn_ptr_bd*. The module also controls the PWM duty cycle by calculating appropriate values for the full compare registers CMPR1, CMPR2 and CMPR3. The duty cycle values for the PWM outputs are determined by the input *D_func*.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: bldc3pwm.asm

C-Callable Version File Names: f2407bldcpwm1.c, f2407bldcpwm2.asm, f2407bldcpwm.h, f243bldcpwm1.c, f243bldcpwm2.asm, f243_bldcpwm.h

Item	ASM Only	C-Callable ASM	Comments
Code size	82 words	89 words†	
Data RAM	6 words	0 words†	
Multiple instances	No	See note	

† Each pre-initialized PWMGEN structure instance consumes 6 words in the data memory and 8 words in the .cinit section.

Note: Multiple instances must point to distinct interfaces on the target device. Multiple instances pointing to the same PWM interface in hardware may produce undefined results. So the number of interfaces on the F241/3 is limited to one, while there can be upto two such interfaces on the LF2407.

Direct ASM Interface

Table 7. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	cmtn_ptr_bd	Commutation(or switching) state pointer input	Q0	0–5
	D_func	Duty ratio of the PWM outputs	Q15	0–7FFF
	Mfunc_p	PWM period modulation input	Q15	0–7FFF
H/W Outputs	PWMx (x=1,2,3,4,5,6)	Full compare PWM outputs from 24x/24xx device	N/A	N/A
Init / Config	FPERIOD	PWM frequency select constant. Default value is set for 20kHz. Modify this constant for different PWM frequency.	Q0	Application dependent
	24x/24xx	Select appropriate 24x/24xx device in the x24x_app.h file.	N/A	N/A

Variable Declaration:
In the system file include the following statements:

```
.ref   BLDC_3PWM_DRV, BLDC_3PWM_DRV_INIT   ;function call
.ref    cmtn_ptr_bd, D_func, Mfunc_p             ;input
```

Memory map:
All variables are mapped to an uninitialized named section ‘bldc3pwm’

Example:

```
ldp  #cmtn_ptr_bd            ;Set DP for module inputs
bldd #input_var1, cmtn_ptr_bd ;Pass input variables to module inputs
bldd #input_var2, D_func
CALL BLDC_3PWM_DRV
```

Note:

Since this is an output driver module it does not have any user configurable s/w outputs and, therefore, does not need any output parameter passing. This s/w module calculates the compare values, which are used in the full compare unit internal to 24x/24xx device. From the compare values the device generates the PWM outputs.

C/C-Callable ASM Interface

Object Definition

The structure of the PWMGEN Interface Object is defined by the following structure definition

```
typedef struct  {
    int cmtn_ptr_bd;     /* Commutation(or switching) state pointer input[Q0] */
    int mfunc_p;         /* Duty ratio of the PWM outputs[Q15]                */
    int period_max;      /* Maximum period                                    */
    int d_func;          /* PWM period modulation input[Q15]                  */
    int (*init)();       /* Function pointer to INIT function                 */
    int (*update)();     /* Function pointer to UPDATE function               */
            } PWMGEN;
```

Table 8. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	cmtn_ptr_bd	Commutation(or switching) state pointer input	Q0	0−5
	D_func	Duty ratio of the PWM outputs	Q15	0−7FFF
	Mfunc_p	PWM period modulation input	Q15	0−7FFF
H/W Outputs	PWMx (x=1,2,3,4,5,6)	Full compare PWM outputs from 24x/24xx device	N/A	N/A
Init / Config	FPERIOD	PWM frequency select constant. Default value is set for 20kHz. Modify this constant for different PWM frequency.	Q0	Application dependent
	24x/24xx	Select appropriate 24x/24xx device in the x24x_app.h file.	N/A	N/A

Special Constants and Datatypes

PWMGEN
The module definition itself is created as a data type. This makes it convenient to instance an interface to the PWM Generator module.

PWMGEN _DEFAULTS
Initializer for the PWMGEN Object. This provides the initial values to the terminal variables as well as method pointers.

PWMGEN_handle
Typedef'ed to PWMGEN *

F243_PWMGEN_DEFAULTS
Constant initializer for the F243 PWM Interface.

F2407_PWMGEN_DEFAULTS
Constant initializer for the F2407 PWM Interface

Methods

void init (PWMGEN_handle)
Initializes the PWM Gen unit hardware.

void update(PWMGEN_handle)
Updates the PWM Generation hardware with the data from the PWM Structure.

Module Usage

Instantiation:
The interface to the PWM Generation Unit is instanced thus:

```
PWMGEN  pwm;
```

Initialization:
To instance a pre-initialized object

```
PWMGEN  pwm =PWMGEN_DEFAULTS
```

Hardware Initialization:

```
pwm.init(&pwm);
```

Invoking the update function

```
pwm.update(&pwm);
```

Example:
Lets instance one PWMGEN object

```
PID2 pid =PID2_DEFAULTS;
PWMGEN  pwm =PWMGEN_DEFAULTS;

main()
  {
  pid.k0_reg2 = 0x080;        /* Initialize */
  pid.k1_reg2 = 0x0140;
  pid.kc_reg2 = 0x0506;

  pwm.cmtn_ptr_bd = 3;        /* Initialize */
  pwm.mfunc_p =0x1777;
  pwm.d_func = 0x6fff;
  pwm.period_max =0x5fff;

(*pwm.init)(&pwm);            /* Call the compute function for pwm */
  }
  void interrupt periodic_interrupt_isr()
  {

(*pid.update)(&pid);          /*call compute function for pid */

/* Lets output pid.out_reg2 */

  pwm.d_func = bldc.pid2.out_reg2;

  (*pwm.update)(&pwm);

}
```

Background Information

Figure 2 shows the 3-phase power inverter topology used to drive a 3-phase BLDC motor. In this arrangement, the motor and inverter operation is characterized by a *two phases ON* operation. This means that two of the three phases are always energized, while the third phase is turned off. This is achieved by controlling the inverter switches in a periodic 6 switching or commutation states. The bold arrows on the wires in Figure 2 indicate the current flowing through two motor stator phases during one of these commutation states. The direction of current flowing into the motor terminal is considered as positive, while the current flowing out of the motor terminal is considered as negative. Therefore, in Figure 2, Ia is positive, Ib is negative and Ic is 0.

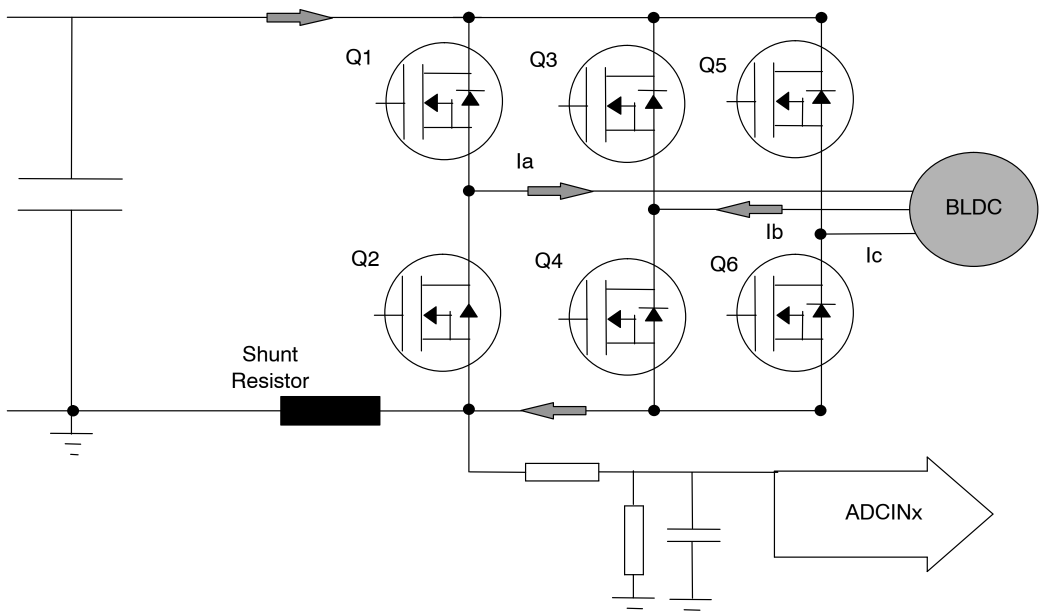


Figure 2. Three Phase Power Inverter for a BLDC Motor Drive

In this control scheme, torque production follows the principle that current should flow in only two of the three phases at a time and that there should be no torque production in the region of Back EMF zero crossings. Figure 3 depicts the phase current and Back EMF waveforms for a BLDC motor during the *two phases ON* operation. All the 6 switching states of the inverter in Figure 2 are indicated in Figure 3 by S1 through S6. As evident from Figure 3, during each state only 2 of the 6 switches are active, while the remaining 4 switches are turned OFF. Again, between the 2 active switches in each state, the odd numbered switch (Q1or Q3 or Q5) are controlled with PWM signal while the even numbered switch (Q2 or Q4 or Q6) is turned fully ON. This results in motor current flowing through only two of the three phases at a time. For example in state S1, Ia is positive, Ib is negative and Ic is 0. This is achieved by driving Q1 with PWM signals and turning Q4 fully ON. This state occurs when the value in the commutation state pointer variable, cmtn_ptr_bd, is 0. Table 9 summarizes the state of the inverter switches and the corresponding values of the related peripheral register, the commutation pointer and the motor phase currents.

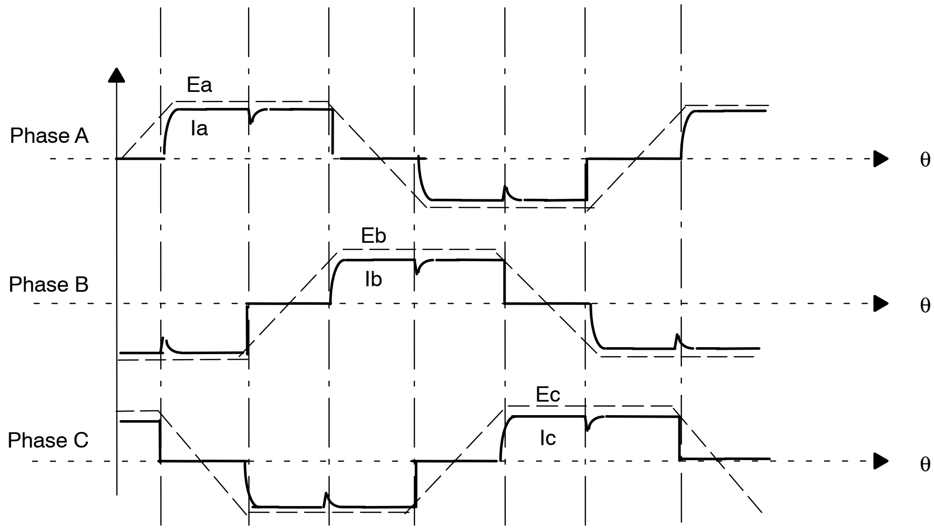


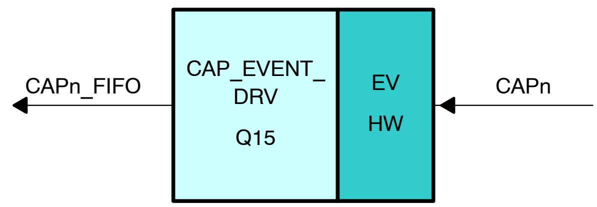
Figure 3. Phase Current and Back EMF Waveforms in 3-ph BLDC Motor Control

Table 9. Commutation States in 3-ph BLDC Motor Control

State	cmtn_ ptr_bd	ACTR	Q1	Q2	Q3	Q4	Q5	Q6	Ia	Ib	Ic
S1	0	00C2	**PWM**	OFF	OFF	ON	OFF	OFF	+ve	−ve	0
S2	1	0C02	**PWM**	OFF	OFF	OFF	OFF	ON	+ve	0	−ve
S3	2	0C20	OFF	OFF	**PWM**	OFF	OFF	ON	0	+ve	−ve
S4	3	002C	OFF	ON	**PWM**	OFF	OFF	OFF	−ve	+ve	0
S5	4	020C	OFF	ON	OFF	OFF	**PWM**	OFF	−ve	0	+ve
S6	5	02C0	OFF	OFF	OFF	ON	**PWM**	OFF	0	−ve	+ve

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 the event manager (EV) capture input pins of 24x/24xx devices.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: cap_drv.asm

C-Callable Version File Names: F243CAP.h, F243CAPx.c, F2407CAPx.c, F2407CAP.H, CAPTURE.H

Item	ASM Only	C-Callable ASM	Comments
Code size	32 words	54 words (49 words .text, 5 words .cinit)	
Data RAM	1 words	6 words	
Multiple instances	No	Yes†	Multiple instances must be initialized to point to different capture pin routines.

† Creating multiple instances pointing to the same capture pin can cause undefined results.

Direct ASM Interface

Table 10. Module Terminal Variables/Functions

	Name	Description	Format	Range
H/W Inputs	CAPn (n=1,2,3,4)	Capture input signals to 24x/24xx device	N/A	N/A
Outputs	CAPnFIFO (n=1,2,3,4)	Capture unit FIFO registers.	N/A	N/A
Init / Config	24x/24xx	Select appropriate 24x/24xx device in the x24x_app.h file.	N/A	N/A
	CLK_prescaler_bits	Initialize this clock prescaler variable. The default value is set to 4. To use this value call the CAP_EVENT_DRV_INIT routine only. For a different value modify this variable and also call the other initialization routine CAP_EVENT_DRV_CLKPS_INIT. The correct value for this parameter is calculated in the Excel file with the user input of the desired clock prescaler (1,2,4,8,16,32,64,128).	Q0	0−7

Variable Declaration:
In the system file include the following statements:

```
.ref   CAP_EVENT_DRV, CAP_EVENT_DRV _INIT    ;function call

.ref   CAP_EVENT_DRV_CLKPS_INIT              ;function call

.ref   CLK_prescaler_bits                    ;parameter
```

Memory map:
Not required.

Example:

```
CALL CAP_EVENT_DRV_INIT
ldp  #CLK_prescaler_bits
splk #7, CLK_prescaler_bits   ;To specify a prescaler of 128
CALL CAP_EVENT_DRV_CLKPS_INIT

ldp  #output_var1          ;Set DP for output variable
bldd #CAP1FIFO,output_var1 ;Pass module o/ps to output vars
bldd # CAP2FIFO, output_var2
bldd # CAP3FIFO, output_var3
```

C/C-Callable ASM Interface

Object Definition

The structure of the CAPTURE object is defined by the following struct

```
/*-----------------------------------------------------------------
Define the structure of the Capture Driver Object
-----------------------------------------------------------------*/
typedef struct { int time_stamp;
                 int (*init)(); /*Pointer to the init function */
                 int (*read)(); /*Pointer to the init function */
               } CAPTURE;
```

Table 11. Module Terminal Variables/Functions

	Name	Description	Format	Range
H/W Input Pins	–	–		Inputs are logic levels on hardware pins.
Output	Time_stamp	An Integer value read from timer assigned to the capture unit.	Q0	−32768 to 32767

Special Constants and Datatypes

CAPTURE
The module definition itself is created as a data type. This makes it convenient to instance an interface to the CAPTURE pin(s).

CAPTURE_DEFAULTS
Initializer for the CAPTURE Object. This provides the initial values to the terminal variables as well as method pointers.

CAPTURE_handle
This is typedef'ed to CAPTURE *.

Methods

void init(CAPTURE_handle)
Initializes the CAPTURE unit on the device to activate the capture function.

int read(CAPTURE_handle)
Reads a time stamp value from the timer associated with the capture unit. Note that the time stamp is placed in the capture object. The return value of the function is either 0 or 1. If the function read a value from the hardware, i.e. if a capture event has occurred, then the function returns 0. Otherwise the return value is 1.

Module Usage

Instantiation:
The interface to the Capture unit on the device is instanced thus:

```
CAPTURE cap1;
```

Initialization:
To instance a pre-initialized object

```
CAPTURE cap1=CAP1_DEFAULTS;
```

Invoking the initialization function:

```
cap1.init(&cap1);
```

Reading a time stamp from the capture unit:

```
cap1.read(&cap1);
```

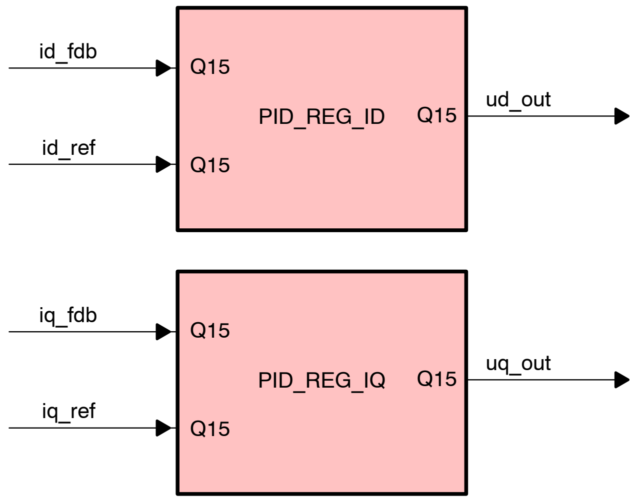
Example:

Lets instance one CAPTURE object, init it and invoke the read function to fetch a time stamp.

```
CAPTURE cap1 CAP1_DEFAULTS; /*Instance the Capture interface object    */

main()
{

     cap1.init(&cap1);

}
void interrupt periodic_interrupt_isr()
{

     int status;
     int time_of_event;

     status=cap1.read(&cap1);

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

     if(status==0)
     {
     time_of_event=cap1.time_stamp;
     }
}
```

Description

These s/w modules implement two PI regulators with integral windup correction.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: pid.asm

ASM Routines: PID_REG_ID, PID_REG_ID_INIT, PID_REG_IQ, PID_REG_IQ_INIT

Parameter calculation excel file: pid.xls

Item	ASM Only	C-Callable ASM	Comments
Code size	134 words	?? words	
Data RAM	24 words	?? words	
xDAIS ready	No	Yes	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

Direct ASM Interface

Table 12. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	id_fdb	Feedback signal for PI regulator PID_REG_ID	Q15	8000−7FFF
	id_ref	Reference signal for PI regulator PID_REG_ID	Q15	8000−7FFF
	iq_fdb	Feedback signal for PI regulator PID_REG_IQ	Q15	8000−7FFF
	iq_ref	Reference signal for PI regulator PID_REG_IQ	Q15	8000−7FFF
Outputs	ud_out	Output for PI regulator PID_REG_ID	Q15	Umin_d_− Umax_d_
	uq_out	Output for PI regulator PID_REG_IQ	Q15	Umin_q_− Umax_q_
Init / Config	Kp_d†	Proportional gain coefficient	Q11	System dependent
	Ki_d†	Integral coefficient	Q25	System dependent
	Kc_d†	Integral windup correction coefficient	Q14	System dependent
	Kp_q†	Proportional gain coefficient	Q11	System dependent
	Ki_q†	Integral coefficient	Q25	System dependent
	Kc_q†	Integral windup correction coefficient	Q14	System dependent

† From the system file, initialize these PI regulator coefficients.

Variable Declaration:
In the system file include the following statements:

```
.ref pid_reg_id,pid_reg_id_init       ; function call
.ref id_fdb,id_ref,Kp_d,Ki_d,Kc_d     ; Inputs|
.ref ud_int                           ; Input
.ref ud_out                           ; Outputs
.ref pid_reg_iq,pid_reg_iq_init       ; function call
.ref iq_fdb,iq_ref,Kp_q,Ki_q,Kc_q     ; Inputs
.ref uq_int                           ; Input
.ref uq_out                           ; Outputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘pid’

Example:

```
ldp  #id_fdb                 ;Set DP for module inputs
bldd #input_var1, id_fdb     ;Pass input variables to module inputs
bldd #input_var2, id_ref

CALL pid_reg_id

ldp  #output_var1            ;Set DP for output variable
bldd #ud_out, output_var1    ;Pass module output to output variable

ldp    #iq_fdb               ;Set DP for module inputs
bldd #input_var3, iq_fdb     ;Pass input variables to module inputs
bldd #input_var4, iq_ref

CALL pid_reg_iq

ldp    #output_var2          ;Set DP for output variable
bldd #uq_out, output_var2    ;Pass module output to output variable
```

C/C-Callable ASM Interface

TBD

Background Information

The discrete-time equations used for the PI controller with anti-windup correction can be summarized as follows:

$$e_n = i*_n - i_n$$

$$U_n = X_{(n-1)} + K_p e_n$$

$$U_{out} = U_{\max} \text{ if } U_n > U_{\max}$$

$$U_{out} = U_{\min} \text{ if } U_n < U_{\min}$$

$$U_{out} = U_n$$

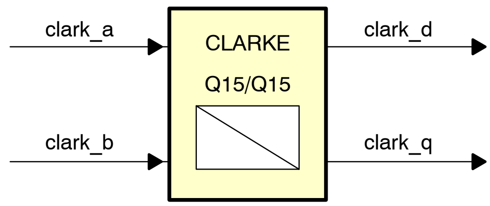
otherwise

$$X_n = X_{(n-1)} + K_i e_n + K_c(U_{out} - U_n)$$

where $K_c = \frac{Ki}{Kp}$

Description

Converts balanced three phase quantities into balanced two phase quadrature quantities.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent/Application Independent

Target Devices: x24x/x24xx

Direct ASM Version File Name: clarke.asm

C-Callable Version File Name: clark.asm

Item	ASM Only	C-Callable ASM	Comments
Code size	19 words	29 words†	
Data RAM	6 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† The Clarke transform operates on structures allocated by the calling function.

Direct ASM Interface

Table 13. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	clark_a	Phase ‘a’ component of the balanced three phase quantities.	Q15	8000–7FFF
	clark_b	Phase ‘b’ component of the balanced three phase quantities	Q15	8000–7FFF
Outputs	clark_d	Direct axis(d) component of the transformed signal	Q15	8000–7FFF
	clark_q	Quadrature axis(q) component of the transformed signal	Q15	8000–7FFF
Init / Config	none			

Variable Declaration:
In the system file include the following statements:

```
.ref   CLARKE, CLARKE_INIT                   ;function call

.ref   clark_a, clark_b, clark_d, clark_q    ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section ‘clarke’

Example:

```
ldp  #clark_a                ;Set DP for module input
bldd #input_var1, clark_a    ;Pass input variable to module input
bldd #input_var2, clark_b

CALL CLARKE

ldp  #output_var1            ;Set DP for output variable
bldd #clark_d, output_var1   ;Pass module output to output
                             ; variable
bldd #clark_q, output_var2
```

C/C-Callable ASM Interface

This function is implemented as a function with two arguments, each a pointer to the input and output structures.

```
struct   { int a;
           int b;
           int c;
         } clark_in;

struct   { int d;
           int q;
         } clark_out;

void clark(&clark_in,&clark_out);
```

The inputs are read from the clarke_in structure and the outputs are placed in the clarke_out structure.

Table 14. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	a	Phase ‘a’ component of the balanced three phase quantities.	Q15	8000−7FFF
	b	Phase ‘b’ component of the balanced three phase quantities	Q15	8000−7FFF
	c	Phase ‘c’ component of the balanced three phase quantities	Q15	8000−7FFF
Outputs	d	Direct axis(d) component of the transformed signal	Q15	8000−7FFF
	q	Quadrature axis(q) component of the transformed signal	Q15	8000−7FFF
Init / Config	none			

Example:
In the following example, the variables intput_a, input_b, input_c are transformed to the quadrature components output_d, output_q.

```
typedef struct { int a,b,c ; } triad;

triad threephase;
triad quadrature;

int threephase_a, threephase_a, threephase_a;
int output_d,output_q;

void some_func(void)
{

threephase.a=input_a;
threephase.b=input_b;
threephase.c=input_c;

clark(&threephase,&quadrature);
```

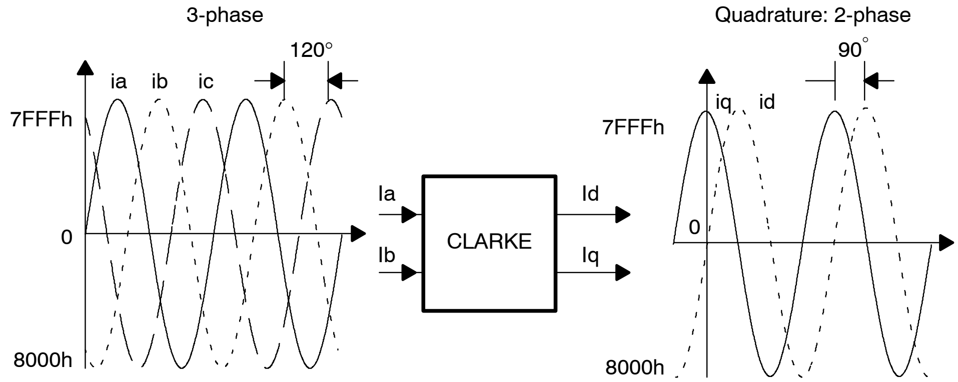
```
output_d=quadrature.a;
output_q=quadrature.b;

}
```

Background Information

Implements the following equations:

$$\begin{cases} Id = Ia \\ Iq = (2Ib + Ia)/\sqrt{3} \end{cases}$$

This transformation converts balanced three phase quantities into balanced two phase quadrature quantities as shown in figure below:



The instantaneous input and the output quantities are defined by the following equations:

$$ia = I \times \sin(\omega t)$$
$$ib = I \times \sin(\omega t + 2\pi/3)$$
$$ic = I \times \sin(\omega t - 2\pi/3)$$
$$\begin{cases} id = I \times \sin(\omega t) \\ iq = I \times \sin(\omega t + \pi/2) \end{cases}$$

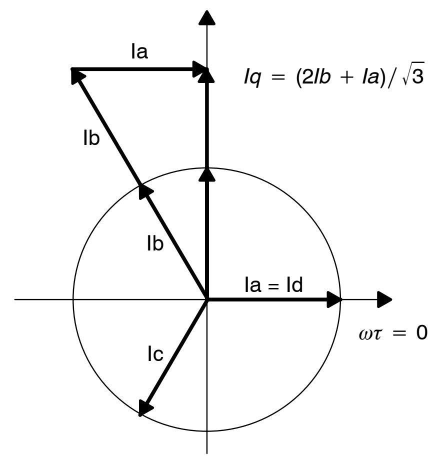
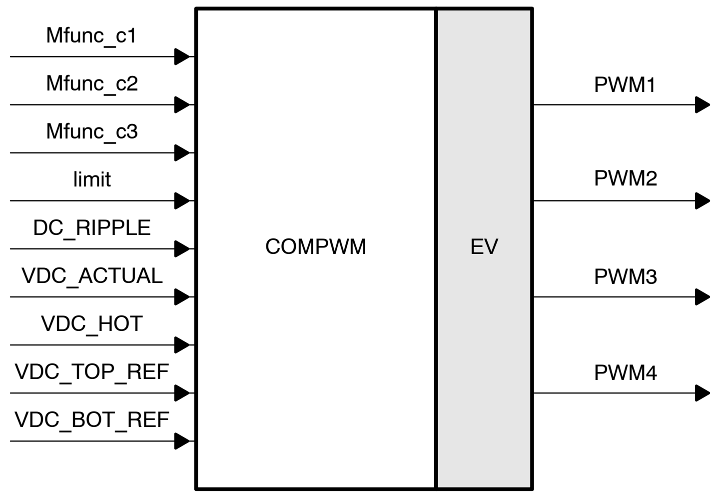


Table 15. Variable Cross Ref Table

Variables in the Equations	Variables in the Code
Ia	clark_a
Ib	clark_b
Id	clark_d
Iq	clark_q

Description

The software module “COMPWM” compensates and/or modifies the PWM output based on system inputs. Although this module is applied for a single phase AC induction motor drive, the same can be applied for a three phase AC induction motor drive.



Availability

This module is available in the direct-mode assembly-only interface (Direct ASM).

Module Properties

Type: Target dependent, Application Dependent

Target Devices: x24x/x24xx

Item	ASM Only	Comments
Code size	311 words	
Data RAM	30 words	
xDAIS module	No	
xDAIS component	No	IALG layer not implemented

Direct ASM Interface

Table 16. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	VDC_ACTUAL	Total DC bus voltage. Measured across both DC bus capacitors.	Q15	0–7FFFh
	VDC_HOT	Half of the DC bus voltage. Measured across the lower DC bus capacitor.	Q15	0000h–7FFFh
	VDC_TOP_REF	The ideal voltage across the top capacitor.	Q15	0000h–7FFFh
	VDC_BOT_REF	The ideal voltage across the lower capacitor. Ideally both the reference voltages are same.	Q15	0000h–7FFFh
	Mfunc_c1	PWM duty ratio	Q15	8000h–7FFFh
	Mfunc_c2	PWM duty ratio	Q15	8000h–7FFFh
	Mfunc_c3	PWM duty ratio	Q15	8000h–7FFFh
	limit	Determines the level of over-modulation	Q0	0 – T1PER/2
	DC_RIPPLE	Software switch to activate riple compensation	Q0	0 (OFF) or 1 (ON)
Outputs	CMPR1	Compensated value for compare 1	Q0	0 – T1PER
	CMPR2	Compensated value for compare 2	Q0	0 – T1PER
Init / Config	ADC_BOT_REF	The reference voltage of the lower DC bus capacitor	Q15	0 – 7FFFh (half of total DC bus)
	ADC_TOP_REF	The reference voltage of the upper DC bus capacitor	Q15	0 – 7FFFh (half of total DC bus)

Variable Declaration:
In the system file include the following statements:

```
.ref   COMPWM                                  ;function call
.ref   COMPWM_INIT                             ;function call
.ref   Mfunc_c1, Mfunc_c2, Mfunc_c3, Mfunc_p   ;Inputs
.ref   limit                                   ;limit
.ref   DC_RIPPLE,VDC_TOP_REF, VDC_BOT_REF      ;inputs
.ref   VDC_ACTUAL, VDC_HOT                     ;inputs
```

Memory map:
All variables are mapped to an uninitialized named section “compwm”

Example:

```
LDP    #DC_RIPPLE

BLDD #ripple_on,  DC_RIPPLE
BLDD #total_bus,  VDC_ACTUAL
BLDD #half_bus,   VDC_HOT
BLDD #ADCref1,    VDC_TOP_REF
BLDD #ADCref2,    VDC_BOT_REF

CALL COMPWM
```

Background Information

This software module modifies a particular system variable based on other system variable feedback. One obvious application of this module is to modify PWM output based on DC bus voltage variation of the voltage source inverter. Ideally, the PWM duty ratio is calculated assuming that the DC bus voltage is stiff with no variation. However, in a practical system there is always DC bus voltage variation based on the load. If this variation is not taken into account than the voltage output of the inverter will get distorted and lower order harmonics will be introduced. The inverter voltage output can be immune to the DC bus variation if the PWM duty ratio is modified according to the bus voltage variation. The following equation shows the mathematical relationship between various variables –

At any PWM cycle the ideal voltage applied across Phase A is,

Va = (T1PER – compare_1)*VDC_TOP_REF – VDC_BOT_REF*compare_1 (1)

In an actual system, voltages across the capacitors will have ripple and the actual voltage applied across Phase A is,

Va_actual = (T1PER – Ta_new)*V1 – Ta_new*V2 (2)

Where,
V1 = measured voltage across the upper capacitor (VDC_ACTUAL – VDC_HOT)
V2 = measured voltage across the lower capacitor (VDC_HOT)

The compensated compare values for Phase A (Ta_new) can be calculated by solving equations (1) and (2) and is given by,

Va = Va_actual

Ta_new = (T1PER*V1 – Va)/ (V1+V2) (3)

Similar, calculation can be performed for Phase B and the compensated compare value becomes,

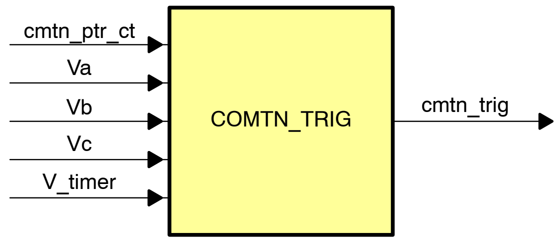
Tb_new = (T1PER*V1 – Vb)/ (V1+V2) (4)

Where,
Vb = (T1PER – compare_2)*VDC_BOT_REF – VDC_TOP_REF*compare_2

It is clear from equations (3) and (4) that the compensation routine depends on accurate measurement of DC bus voltages. Moreover, the user will have to provide protection so that the power devices do not stay ON for a long period to create a short circuit in the motor phase.

Description

This module determines the Bemf zero crossing points of a 3-ph BLDC motor based on motor phase voltage measurements and then generates the commutation trigger points for the 3-ph power inverter switches.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Independent

Target Devices: x24x / x24xx

Assembly File Name: COM_TRIG.asm

C-Callable Version File Name: COM_TRIG.asm, cmtn.h

Item	ASM Only	C-Callable ASM	Comments
Code size	195 words	237 words†	
Data RAM	21 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized CMTN structure instance consumes 19 words in the data memory and 21 words in the .cinit section.

Direct ASM Interface

Table 17. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	cmtn_ptr_ct	Commutation state pointer input. This is used for Bemf zero crossing point calculation for the appropriate motor phase.	Q0	0–5
	Va, Vb, Vc	Motor phase voltages referenced to GND	Q15	0–7FFFh
	V_timer	A virtual timer used for commutation delay angle calculation.	Q0	0–7FFFh
Output	cmtn_trig	Commutation trigger output	Q0	0 or 7FFFh
Init / Config	none			

Variable Declaration:
In the system file include the following statements:

```
.ref   COMTN_TRIG, COMTN_TRIG_INIT    ;function call
.ref   Va, Vb, Vc, cmtn_trig, cmtn_ptr_ct    ;input/output
```

Note: One of the module inputs, V_timer, is a global resource. This should be declared as a GLOBAL variable in the system file.

Memory map:
All variables, except V_timer, are mapped to an uninitialized named section 'com_trig'

V_timer is mapped to the same memory section as the other variables in the main system

Example:

```
ldp  #Va                        ;Set DP for module inputs
bldd #input_var1, Va            ;Pass input variables to module inputs
bldd #input_var2, Vb
bldd #input_var3, Vc
bldd #input_var4, cmtn_ptr_ct
CALL COMTN_TRIG

ldp  #output_var1               ;Set DP for output variable
bldd #cmtn_trig, output_var1  ;Pass module output to output variable
```

C/C-Callable ASM Interface

Object Definition

The structure of the CMTN Object is defined by the following structure definition:

```
/*-----------------------------------------------------------------------------
Define the structure of the CMTN
(Commutation trigger)
-----------------------------------------------------------------------------*/

typedef struct  { int trig; /* Commutation trig output                    */
                  int va; /* Motor phase voltage to GND  for phase A      */
                  int vb; /* Motor phase voltage to GND  for phase B      */
                  int vc ; /* Motor phase voltage to GND  for phase C     */
                  int zc_trig;
                  int ptr_ct; /* Commutation state pointer input          */
                  int debug_Bemf;
                  int noise_windowCntr;
                  int d30_doneFlg;
                  int time_stampNew;
                  int time_stampOld;
                  int v_timer; /* Virtual timer used for commmutaion delay angle
           calculation */
                  int delay;
                  int dt_taskFlg ;
                  int noise_windowMax;
                  int delay_cntr;
                  int cdnw_delta;
                  int nw_dynThold;
                  int (*calc) (); /* Function pointer */
                } CMTN;
```

Table 18. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	ptr_ct	Commutation state pointer input. This is used for Bemf zero crossing point calculation for the appropriate motor phase.	Q0	0−5
	va, vb, vc	Motor phase voltages referenced to GND	Q15	0−7FFFh
	v_timer	A virtual timer used for commutation delay angle calculation.	Q0	0−7FFFh
Output	trig	Commutation trigger output	Q0	0 or 7FFFh

Special Constants and Datatypes

CMTN
The module definition itself is created as a data type. This makes it convenient to instance a Commutation trigger module.To create multiple instances of the module simply declare variables of type CMTN

CMTN_handle
Typedef'ed to CMTN*

CMTN_DEFAULTS
Initializer for the CMTN Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(CMTN_handle)
The default definition of the object implements just one method – the runtime implementation of the Commutation trigger module. This is implemented by means of a function pointer, and the default initializer sets this to cmtn_calc. The argument to this function is the address of the CMTN object.

Module Usage

Instantiation:
The following example instances one such objects:

```
CMTN p1,p2;
```

Initialization:
To instance a pre-initialized object

```
CMTN p1 = CMTN_DEFAULTS, p2 = CMTN_DEFAULTS;
```

Invoking the compute function:

```
p1.calc(&p1);
```

Example:
Lets instance two CMTN objects ,othewise identical but running with different freq values

```
CMTN p1 = CMTN_DEFAULTS; /* Instance the first object */
CMTN p2 = CMTN_DEFAULTS; /* Instance the second object */

main()
{
    p1.ptr_ct  = 5;
    p1.va      = 7;
    p1.vb      = 0;
    p1.vc      =8;
    p1.v_timer =2;
    p1.nw_dynThold = 90;
    p1.dt_taskFlg = 0;
    p1.cdnw_delta = 0;
    p1.d30_doneFlg =0;
    p1.time_stampNew =14;
```

```
    p2.ptr_ct  = 1;
    p2.va      =6;
    p2.vb      =7;
    p2.vc      = 2;
    p2.v_timer = 78;
    p2.nw_dynThold = 3;
    p2.dt_taskFlg = 0;
    p2.cdnw_delta = 7;
    p2.d30_doneFlg = 15;
    p2.time_stampNew = 30;
}

void interrupt periodic_interrupt_isr()
{
       (*p1.calc)(&p1);    /* Call compute function for p1 */
       (*p2.calc)(&p2);    /* Call compute function for p2 */

       x = p1.trig;            /* Access the output */
       y = p1.time_stampNew;   /* Access the output */
       z = p1.time_stampOld;   /* Access the output */
       q = p2.trig;            /* Access the output */
       r = p2.time_stampNew;   /* Access the output */
       s = p2.time_stampOld;   /* Access the output */

    /* Do something with the outputs */
}
```

Background Information

Figure 4 shows the 3-phase power inverter topology used to drive a 3-phase BLDC motor. In this arrangement, the motor and inverter operation is characterized by a two phase ON operation. This means that two of the three phases are always energized, while the third phase is turned off.

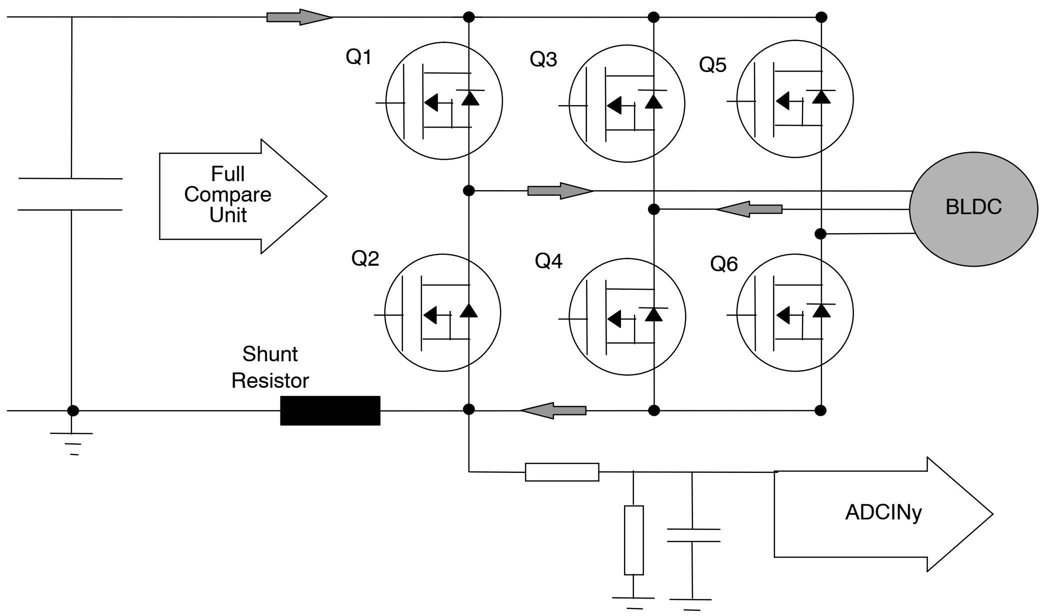


Figure 4. Three Phase Power Inverter for a BLDC Motor Drive

The bold arrows on the wires indicate the Direct Current flowing through two motor stator phases. For sensorless control of BLDC drives it is necessary to determine the zero crossing points of the three Bemf voltages and then generate the commutation trigger points for the associated 3-ph power inverter switches.

Figure 5 shows the basic hardware necessary to perform these tasks.

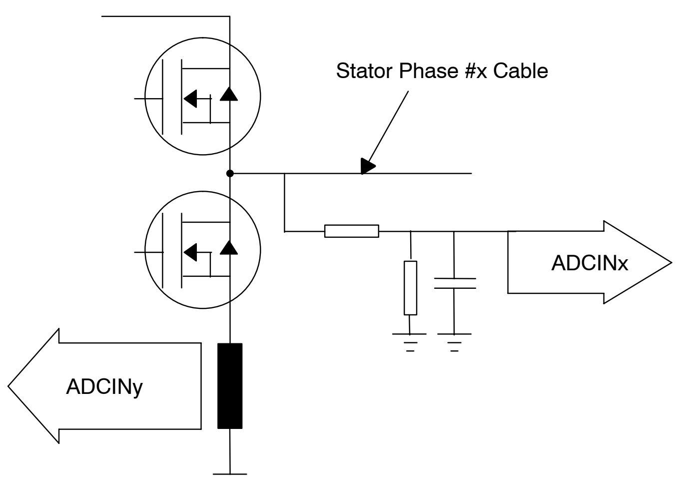


Figure 5. Basic Sensorless Additional Hardware

The resistor divider circuit is specified such that the maximum output from this voltage sensing circuit utilizes the full ADC conversion range. The filtering capacitor should filter the chopping frequency, so only very small values are necessary (in the range of nF). The sensorless algorithm is based only on the three motor terminal voltage measurements and thus requires only four ADC input lines.

Figure 6 shows the motor terminal model for phase A, where L is the phase inductance, R is the phase resistance, Ea is the back electromotive force, Vn is the star connection voltage referenced to ground and Va is the phase voltage referenced to ground. Va voltages are measured by means of the DSP controller ADC Unit and via the voltage sense circuit shown in Figure 5.

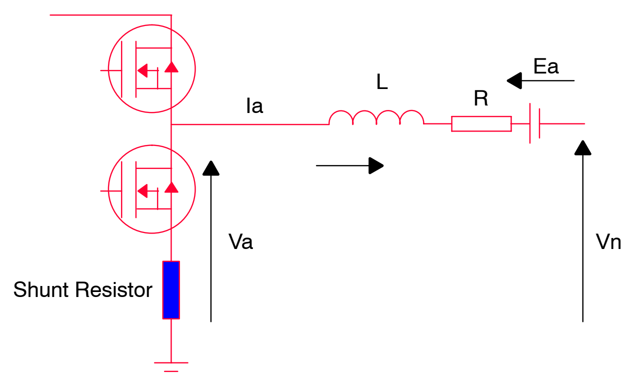


Figure 6. Stator Terminal Electrical Model

Assuming that phase C is the non-fed phase it is possible to write the following equations for the three terminal voltages:

$$Va = RIa + L\frac{dIa}{dt} + Ea + Vn.$$

$$Vb = RIb + L\frac{dIb}{dt} + Eb + Vn$$

$$Vc = Ec + Vn$$

As only two currents flow in the stator windings at any one time, two phase currents are equal and opposite. Therefore,

$$Ia = -Ib$$

Thus, by adding the three terminal voltage equations we have,

$$Va + Vb + Vc = Ea + Eb + Ec + 3Vn$$

The instantaneous Bemf waveforms of the BLDC motor are shown in Figure 7. From this figure it is evident that at the Bemf zero crossing points the sum of the three Bemfs is equal to zero. Therefore the last equation reduces to,

$$Va + Vb + Vc = 3Vn$$

This equation is implemented in the code to compute the neutral voltage. In the code, the quantity 3Vn is represented by the variable called *neutral*.

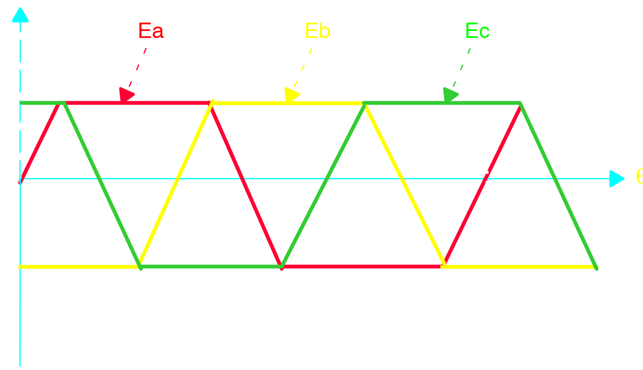


Figure 7. Instantaneous Bemf Waveforms

Bemf Zero Crossing Point Computation

For the non-fed phase (zero current flowing), the stator terminal voltage can be rewritten as follows:

$3Ec = 3Vc - 3Vn.$

This equation is used in the code to calculate the Bemf zero crossing point of the non-fed phase C. Similar equations are used to calculate the Bemf zero crossing points of other Bemf voltages Ea and Eb. As we are interested in the zero crossing of the Bemf it is possible to check only for the Bemf sign change; this assumes that the Bemf scanning loop period is much shorter than the mechanical time constant. This function is computed after the three terminal voltage samples, once every 16.7μs (60kHz sampling loop).

Electrical Behaviour at Commutation Points

At the instants of phase commutation, high dV/dt and dI/dt glitches may occur due to the direct current level or to the parasitic inductance and capacitance of the power board. This can lead to a misreading of the computed neutral voltage. This is overcomed by discarding the first few scans of the Bemf once a new phase commutation occurs. In the code this is implemented by the function named 'NOISE_WIN'. The duration depends on the power switches, the power board design, the phase inductance and the driven direct current. This parameter is system-dependent and is set to a large value in the low speed range of the motor. As the speed increases, the s/w gradually lowers this duration since the Bemf zero crossings also get closer at higher speed.

Commutation Instants Computation

In an efficient sensored control the Bemf zero crossing points are displaced 30° from the instants of phase commutation. So before running the sensorless BLDC motor with help of the six zero crossing events it is necessary to compute the time delay corresponding to this 30° delay angle for exact commutation points. This is achieved by implementing a position interpolation function. In this software it is implemented as follows: let T be the time that the rotor spent to complete the previous revolution and α be the desired delay angle. By dividing α by 360° and multiplying the result by T we obtain the time duration to be spent before commutating the next phase pair. In the code this delay angle is fixed to 30°. The corresponding time delay is represented in terms of the number of sampling time periods and is stored in the variable *cmtn_delay*. Therefore,

$$\text{Time delay} = cmtn\_delay \cdot Ts = T(\alpha/360) = v\_timer \cdot Ts(\alpha/360) = v\_timer \cdot Ts/12$$

Where, Ts is the sampling time period and *v_timer* is a timer that counts the number of sampling cycles during the previous revolution of the rotor.

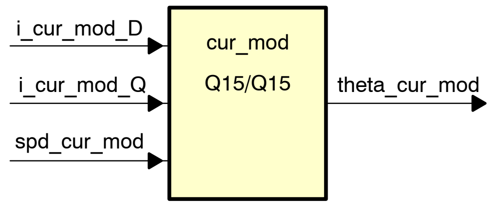
The above equation is further simplified as,

$$cmtn\_delay = v\_timer /12$$

This equation is implemented in the code in order to calculate the time delay corresponding to the 30° commutation delay angle.

Description

This module estimates the rotor flux position based on three inputs. These are the quadrature(isq) and direct(isd) axis components of the stator current in the orthogonal rotating reference frame(output of PARK transform) and the rotor mechanical speed.



Availability

This module is available in direct-mode assembly-only interface (Direct ASM).

Module Properties

Type: Peripheral Independent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: cur_mod.asm

Item	ASM Only	Comments
Code size	122 words	
Data RAM	13 words	
xDAIS ready	No	
xDAIS component	No	
Multiple instances	No	

Direct ASM Interface

Table 19. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	i_cur_mod_D	Direct axis component of current in rotating reference frame (D component of PARK transform)	Q15	0 – 7FFF
	i_dur_mod_Q	Quadrature axis component of current in rotating reference frame (Q component of PARK transform)	Q15	0 – 7FFF
	spd_cur_mod	Per unit motor speed.	Q15	0 – 7FFF
Outputs	theta_cur_mod	rotor flux position	Q15	0 – 7FFF (0–360 degrees)
Init / Config	p	Number of pole pairs	Q0	User specified
	Kr, Kt, K	Parameters depending on the motor used	Q12	User specified

Variable Declaration:
In the system file include the following statements:

```
.ref   CURRENT_MODEL,CURRENT_MODEL_INIT ;function call
.ref   i_cur_mod_D,i_cur_mod_Q          ;Inputs
.ref   spd_cur_mod                      ;Input
.ref   theta_cur_mod                    ;Outputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘cur_mod.’

Example:

```
LDP      #spd_cur_mod               ;Set DP for current module input
BLDD     #speed_frq,spd_cur_mod     ; variables
BLDD     #park_D,i_cur_mod_D
BLDD     #park_Q,i_cur_mod_Q
CALL     CURRENT_MODEL

ldp   #output_var1                  ;Set DP for output variable
bldd  #theta_cur_mod, output_var1   ;Pass module output to output
                                 variable
```

Background Information

With asynchronous drive, the mechanical rotor angular speed is not, by definition, equal to the rotor flux angular speed. This implies that the necessary rotor flux position cannot be detected directly by themechanical position sensor used with the asynchronous motor (QEP or tachometer). The current model module must be added to the generic structure in the regulation block diagram to perform a current and speed closed loop for a three phase ACI motor in FOC control.

The current model consists of implementing the following two equations of the motor in d, q reference frame:

$$i_{ds} = T_R \frac{di_{mR}}{dt} + i_{mR}$$

$$fs = \frac{1}{\omega_b}\frac{(d\theta)}{dt} = n + \frac{i_{qS}}{(T_R i_{mR}\,\omega_b)}$$

where we have:

θ is the rotor flux position

i_{mR} is the magnetizing current

$T_R = \frac{L_R}{R_R}$ is the rotor time constant with L_R the rotor inductance and R_R the rotor resistance.

fs is the rotor flux speed

ω_b is the electrical nominal flux speed.

Knowledge of the rotor time constant is critical to the correct functioning of the current model. This system outputs the rotor flux speed that is integrated to calculate the rotor flux position.

Assuming that $i_{qS_{(k+1)}} \approx i_{qS_k}$ the above equations can be discretized as follows:

$$i_{mR_{(k+1)}} = i_{mR_k} + \frac{T}{T_R}\left(i_{dS_k} - i_{mR_k}\right)$$

$$f_{S_{(k+1)}} = n_{k+1} + \frac{1}{(T_R\omega\omega_b)}\,\frac{i_{qS_k}}{i_{mR_{(k+1)}}}$$

In these equations, T represents the main control loop period.

Let the two constants $\frac{T}{T_R}$ and $\frac{1}{(T_R\omega_b)}$, in the last equations, be renamed as K_R and K_t respectively. These two constants need to be calculated according to the motor parameters and then initialized into the cur_mod.asm file.

Let us take an example with the specific motor parameters:

$$K_R = \frac{T}{T_R} = \frac{T}{(L_R/R_R)} = \frac{100.10^{-6}}{(162.10^{-3}/5.365)} = 3.3117.10^{-3} \Leftrightarrow 0eh \ \ 4.12f$$

$$K_t = \frac{1}{(T_R 2\pi\pi_n)} = \frac{1}{(30.195.10^{-3} \times 2\pi \times 50)} = 105.42.10^{-3} \Leftrightarrow 01b0h \ \ 4.12f$$

Once the motor flux speed (fs) has been calculated, the necessary rotor flux position (θ_{cm}) is computed by the integration formula:

$$\theta_{cm} = \theta_{cm_k} + \omega_b \ f_{s_k} T$$

As the rotor flux position range is $[0; 2\pi]$, 16 bit integer values have been used to achieve the maximum resolution. However, the cur_mod module output, theta_cur_mod, is a 15 bit integer value (0–32765). This is done to make this output signal compatible with the input of the I_PARK and PARK modules.

In the above equation, let us denote $\omega_b \ f_s T$ as $\theta incr$. This is the angle variation within one sample period. At nominal speed (in other words, when fs = 1, mechanical speed nominal needs to be determined by the user, here the description of the current model takes 1500 rpm as a nominal speed), $\theta incr$ is thus equal to 0.031415 rad. In one mechanical revolution performed at nominal speed, there are $2\frac{\pi}{0.031415} \approx 200$ increments of the rotor flux position. Let K be defined as the constant, which converts the $[0; 2\pi]$ range into the range (0;655355) range. K is calculated as follows:

$$K = \frac{65536}{200} = 327.68 \Leftrightarrow 0148h$$

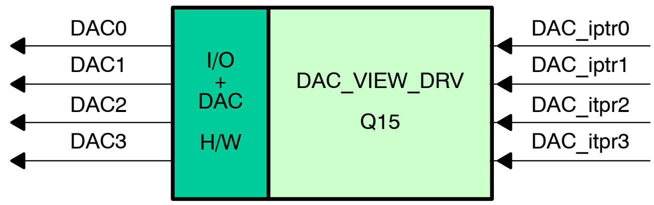
Using this constant, the rotor flux position computation and its formatting becomes:

$$\theta_{cm_{(k=1)}} = \theta_{cm_k} + Kf_{S_k}$$

The θ_{cm_k} is thus represented as 16 bits integer value. As already mentioned ablve, this variable is the computed rotor flux position. It is then passed to the module variable output, theta_cur_mod and scaled for the range (0-32765). The user should be aware that the current model module constants depend on the motor parameters and need to be calculated for each type of motor. The information needed for this are the rotor resistance and the rotor inductance (which is the sum of the magnetizing inductance and the rotor leakage inductance $(L_R = L_H + L_{6R})$).

Description

This module converts any s/w variable with Q15 representation into its equivalent Q0 format that spans the full input range of a 12-bit DAC. Thus, the module output can be directly applied to the input of a 12-bit DAC. This allows the user to view the signal, represented by the variable, at the output of the 12-bit DAC on the 24x/24xx EVM.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target dependent, Application dependent

Target Devices: x24x/x24xx EVM only

Direct ASM Version File Name: dac_view.asm

C-Callable Version File Names: evmdac.asm, evmdac.h

Item	ASM Only	C-Callable ASM	Comments
Code size	54 words	50 words‡	
Data RAM	7 words	0 words‡	
Multiple instances	No	No†	

† Since there is only one DAC on the EVM, creating multiple instances of the interface may produce undefined results.

‡ Each pre−initialized EVMDAC struction instance consumes 6 words in the data memory and 8 words in the .cinit section.

Direct ASM Interface

Table 20. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	DAC_iptrx (x=0,1,2,3)	These input variables contain the addresses of the desired s/w variables.	N/A	N/A
H/W Outputs	DACx (x=0,1,2,3)	Output signals from the 4 channel DAC on the 24x/24xx EVM.	N/A	0−Vcc
Init / Config	DAC_iptrx (x=0,1,2,3)	Initialize these input variables with the addresses of the desired s/w variables. However, this initialization is optional, since these input variables can also be loaded with the addresses of any s/w variables from the Code Composer watch window.	N/A	N/A

Variable Declaration:
In the system file include the following statements:

```
.ref   DAC_VIEW_DRV, DAC_VIEW _DRV _INIT            ;function call
.ref   DAC_iptr0, DAC_iptr1, DAC_iptr2, DAC_iptr3   ;inputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘dac_view’

Example:
During the initialization part of the user code, initialize the module inputs with the address of the desired variables as shown below:

```
CALL DAC_VIEW_DRV_INIT      ;Initializes DAC parameters
ldp  #DAC_iptr0             ;Set DP for module inputs
                            ;Pass input variables to module inputs
splk #input_var0,DAC_iptr0
splk #input_var1, DAC_iptr1
splk #input_var2, DAC_iptr2
splk #input_var3, DAC_iptr3
```

Then in the interrupt routine just call the driver module to view the intended signals at the DAC output.

```
CALL DAC_VIEW_DRV
```

Note:

Since this is an output driver module it does not have any user configurable s/w outputs and, therefore, does not need any output parameter passing.

C/C-Callable ASM Interface

Object Definition

The structure of the EVMDAC object is defined by the following structure definition

```
/*-----------------------------------------------------------------------------
EVMDAC.H:
      Interface header file for the F24X/F240x EVM DAC interface(s).
-----------------------------------------------------------------------------*/
typedef struct {
  int *qptr0;/* Pointer to source data output on DAC channel 0 */
  int *qptr1;/* Pointer to source data output on DAC channel 1 */
  int *qptr2;/* Pointer to source data output on DAC channel 2 */
  int *qptr3;/* Pointer to source data output on DAC channel 3 */
  int scale;
  int (*update)();
  } EVMDAC ;
```

Table 21. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	DAC_qptrx (x=0,1,2,3)	These input variables contain the addresses of the s/w variables to be output on the DAC Channels.	int *	Must be pointed to legal data mem locations.
	scale	Contains the hardware scaling constant Dmax.	Q0 integer	–
H/W Outputs	DACx (x=0,1,2,3)	Output signals from the 4 channel DAC on the 24x/24xx EVM.	Analog voltages	0−Vcc

Special Constants and Datatypes

EVMDAC
The module definition itself is created as a data type. This makes it convenient to instance an interface to the DAC on the EVM.

EVMDAC_DEFAULTS
Initializer for the EVMDAC Object. This provides the initial values to the terminal variables as well as method pointers.

Methods

void update (EVMDAC *)
The only method implemented for this object is the up-date function.

Module Usage

Instantiation:
The interface to the DAC on the EVM is instanced thus:

```
EVMDAC dac;
```

Initialization:
To instance a pre-initialized object

```
EVMDAC dac=EVMDAC_DEFAULTS
```

Invoking the update function:

```
dac.update(dac);
```

Example:

Lets instance one EVMDAC object and one SVGENMF object, (For details on SVGENMF see the SVGEN_MF.DOC.). The outputs of SVGENMF are output via the F24x EVM DAC.

```
SVGENMF sv1=SVGEN_DEFAULTS;    /*Instance the space vector gen object */
EVMDAC  dac=EVMDAC_DEFAULTS;   /*Instance the DAC interface object    */

main()
{
  sv1.freq=1200;  /* Set properties for sv1 */

  dac.qptr0=&sv1.va;
  dac.qptr1=&sv1.vb;
  dac.qptr2=&sv1.vc;
  dac.qptr3=&sv1.vc;

}
void interrupt periodic_interrupt_isr()
{
  sv1.calc(&sv1); /* Call compute function for sv1 */

  /* Lets display sv1.va,sv1.vb, and sv1.vc */

  dac.update(&dac); /* Call the update function */

  }
```

Background Information

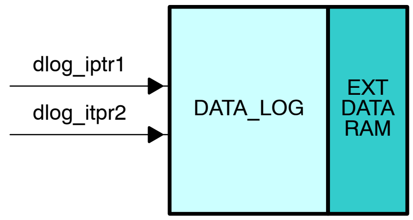
This s/w module converts a variable with Q15 representation, into its equivalent Q0 format that spans the full input range of a 12-bit DAC. If the variable in Q15 is U, and the DAC maximum digital input word is D_{max} (=4095 for a 12-bit DAC), then the equivalent Q0 variable D_{in} (representing U) applied to the DAC input is calculated by the following equation:

$$D_{in} = \frac{D_{max}}{2} + U * \frac{D_{max}}{2}$$

This means that, as U varies from –1 to +1, the digital word input to the DAC varies from 0 to D_{max}. Thus U is converted to a Q0 variable that spans the full input range of the DAC.

Description

This module stores the realtime values of two user selectable s/w variables in the external data RAM provided on the 24x/24xx EVM. Two s/w variables are selected by configuring two module inputs, *dlog_iptr1* and *dlog_iptr2*, point to the address of the two variables. The starting addresses of the two RAM locations, where the data values are stored, are set to 8000h and 8400h. Each section allows logging of 400 data values.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Independent

Target Devices: x24x/x24xx

Assembly File Name: data_log.asm

ASM Routines: DATA_LOG, DATA_LOG_INIT

C-Callable ASM File Names: data_log1.c, data_log2.asm, data_log.h

Item	ASM Only	C-Callable ASM	Comments
Code size	80 words	118 words†	
Data RAM	8 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized DATALOG structure instance consumes 14 words in the data memory and 16 words in the .cinit section.

Direct ASM Interface

Table 22. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	dlog_iptrx (x=1,2)	These inputs contain the addresses of the desired variables.	N/A	N/A
Outputs	none			
Init / Config	dlog_iptrx (x=1,2)	Initialize these inputs with the addresses of the desired variables. However, this initialization is optional, since these input variables can also be loaded with the addresses of any s/w variables from the Code Composer watch window.		

Variable Declaration:
In the system file include the following statements:

```
.ref   DATA_LOG, DATA_LOG _INIT       ;function call
.ref   dlog_iptr1, dlog_iptr2         ;inputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘data_log’

Example:
During the initialization part of the user code, initialize the module inputs with the address of the desired variables as shown below:

```
CALL DATA_LOG_INIT   ;Initializes DAC parameters

ldp  #dlog_iptr1              ;Set DP for module inputs
splk #input_var1, dlog_iptr1  ;Pass input variables to module inputs
splk #input_var2, dlog_iptr2
```

Then in the interrupt routine just call the module to store the values of the intended variables in the external RAM.

```
CALL DATA_LOG
```

Note:

This module does not have any user configurable s/w outputs and, therefore, does not need any output parameter passing.

C/C-Callable ASM Interface

Object Definition

The structure of the DATALOG object is defined in the header file, data_log.h, as shown in the following:

```
typedef struct { int  *dlog_iptr1;      /* Input: First input pointer (Q15) */
                 int  *dlog_iptr2;      /* Input: Second input pointer (Q15) */
                 int  trig_value;       /* Input: Trigger point (Q15) */
                 int  graph_ptr1;       /* Variable: First graph address */
                 int  graph_ptr2;       /* Variable: Second graph address */
                 int  dlog_skip_cntr;   /* Variable: Data log skip counter */
                 int  dlog_cntr;        /* Variable: Data log counter */
                 int  task_ptr;         /* Variable: Task address */
                 int  dlog_prescale;    /* Parameter: Data log prescale */
                 int  dlog_cntr_max;    /* Parameter: Maximum data buffer */
                 int  dl_buffer1_adr;   /* Parameter: Buffer starting address 1 */
                 int  dl_buffer2_adr;   /* Parameter: Buffer starting address 2 */
                 int  (*init)();        /* Pointer to init function */
                 int  (*update)();      /* Pointer to update function */
               } DATALOG;
```

Special Constants and Datatypes

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

DATALOG_DEFAULTS
Initializer for the DATALOG object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, data_log.h.

Methods

void init(DATALOG *);
void update(DATALOG *);
This default definition of the object implements two methods – the initialization and runtime update function for data logging. This is implemented by means of a function pointer, and the default initializer sets these to data_log_init and data_log_update functions. The argument to these functions is the address of the DATALOG object. Again, this statement is written in the header file, data_log.h.

Module Usage

Instantiation:
The following example instances two such objects:

```
DATALOG  dlog1, dlog2;
```

Initialization:
To instance a pre-initialized object:

```
DATALOG dlog1 = DATALOG_DEFAULTS;
DATALOG dlog2 = DATALOG_DEFAULTS;
```

Invoking the compute function:

```
dlog1.update(&dlog1);
dlog2.update(&dlog2);
```

Example:

Lets instance two DATALOG objects, otherwise identical, and run four data logging variables. The following example is the c source code for the system file.

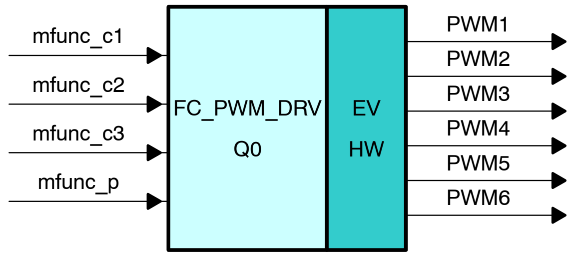
```
DATALOG dlog1= DATALOG_DEFAULTS;       /* instance the first object */
DATALOG dlog2 = DATALOG_DEFAULTS;      /* instance the second object */

main()
{

     dlog1.init(&dlog1);               /* Initialize the data_log function for dlog1 */
     dlog2.init(&dlog2);               /* Initialize the data_log function for dlog2 */

/* Since dlog1 already occupied the data buffer addressed (by default) from 0x8000 to
0x87FF, the starting buffer address for dlog2 need to set to other empty space of memory */

     dlog2.dl_buffer1_adr = 0x08800;   /* Set new starting buffer address of dlog2 */
     dlog2.dl_buffer2_adr = 0x08C00;   /* Set new starting buffer address of dlog2 */

     dlog1.dlog_iptr1 = &input1;       /* Pass inputs to dlog1 */
     dlog1.dlog_iptr2 = &input2;       /* Pass inputs to dlog1 */

     dlog2.dlog_iptr1 = &input3;       /* Pass inputs to dlog2 */
     dlog2.dlog_iptr2 = &input4;       /* Pass inputs to dlog2 */

}

void interrupt periodic_interrupt_isr()
{

     dlog1.update(&dlog1);             /* Call update function for dlog1 */
     dlog2.update(&dlog2);             /* Call update function for dlog2 */

/* This module does not have any user configurable s/w outputs and, therefore, does not
need any output parameter passing.  */

}
```

Background Information

This s/w module stores 400 realtime values of each of the selected input variables in the data RAM as illustrated in the following figures. The starting addresses of two RAM sections, where the data values are stored, are set to 8000h and 8400h.

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 unit in 24x/24xx event manager(EV). This also allows PWM period modulation.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: pwm_drv.asm

C-Callable Version File Names: F243PWM1.C, F243PWM2.ASM, F243PWM.H, F2407PWM1.C, F2407PWM2.C, F2407PWM3.ASM, F2407PWM4.ASM, F2407PWM.H, PWM.H

Item	ASM Only	C-Callable ASM	Comments
Code size	52 words	88 words † ‡ §	
Data RAM	6 words	0 words §	
Multiple instances	No	Yes	

† Multiple instances must point to distinct interfaces on the target device. Multiple instances pointing to the same PWM interface in hardware may produce undefined results. So the number of interfaces on the F241/3 is limited to one, while there can be upto two such interfaces on the LF2407.

‡ If, on the 2407, there are two interfaces concurrently linked in, then the code size will be 176 words + .cinit space + data memory space.

§ Each pre-initialized PWMGEN structure instance consumes 6 words in data memory and 8 words in the .cinit section.

Direct ASM Interface

Table 23. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	mfunc_cx (x=1,2,3)	Duty ratios for full compare unit 1, 2 and 3	Q15	8000−7FFF
	mfunc_p	PWM period modulation function	Q15	8000−7FFF
Outputs	PWMx (x=1,2,3,4,5,6)	Full compare PWM outputs from 24x/24xx device.	N/A	N/A
Init / Config	24x/24xx	Select appropriate 24x/24xx device from the x24x_app.h file.	N/A	N/A
	FPERIOD	PWM frequency select constant. Default value is set for 20kHz. Modify this constant for different PWM frequency.	Q0	Application dependent

Variable Declaration:
In the system file include the following statements:

```
.ref   FC_PWM_DRV, FC_PWM _DRV _INIT            ;function call
.ref   mfunc_c1, mfunc_c2, mfunc_c3, mfunc_p    ;inputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘pwm_drv’

Example:

```
ldp  #mfunc_c1              ;Set DP for module inputs
bldd #input_var1, mfunc_c1  ;Pass input variables
                            ;to module inputs
bldd #input_var2, mfunc_c2
bldd #input_var3, mfunc_c3
bldd #input_var4, mfunc_p
CALL FC_PWM_DRV
```

Note:

Since this is an output driver module it does not have any user configurable s/w outputs and, therefore, does not need any output parameter passing. This s/w module calculates the compare values, which are used in the full compare unit internal to 24x/24xx device. From the compare values the device generates the PWM outputs.

C/C-Callable ASM Interface

Object Definition

The structure of the PWMGEN Interface Object is defined by the following structure definition

```
typedef struct {
int period_max;    /* PWM Period in CPU clock cycles.  Q0-Input  */
         int mfunc_p;       /* Period scaler. Q15 - Input                */
         int mfunc_c1;      /* PWM 1&2 Duty cycle ratio. Q15, Input      */
         int mfunc_c2;      /* PWM 3&4 Duty cycle ratio. Q15, Input      */
         int mfunc_c3;      /* PWM 5&6 Duty cycle ratio. Q15, Input      */
         int (*init)();     /* Pointer to the init function              */
         int (*update)();   /* Pointer to the update function            */
         } PWMGEN ;
```

Table 24. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	mfunc_cx (x=1,2,3)	Duty ratios for full compare unit 1, 2 and 3	Q15	8000−7FFF
	mfunc_p	PWM period modulation function	Q15	8000−7FFF
Outputs	PWMx (x=1,2,3,4,5,6)	Full compare PWM outputs from 24x/24xx device.	N/A	N/A
Init / Config	24x/24xx	Select appropriate 24x/24xx device from the x24x_app.h file.	N/A	N/A
	period_max	PWM period setting. Modify this constant for different PWM frequency.	Q0	Application dependent

Special Constants and Datatypes

PWMGEN
The module definition itself is created as a data type. This makes it convenient to instance an interface to the PWM Generator module.

PWMGEN _DEFAULTS
Initializer for the PWMGEN Object. This provides the initial values to the terminal variables as well as method pointers.

PWMGEN_handle
Typedef'ed to PWMGEN *

F243_FC_PWM_GEN
Constant initializer for the F243 PWM Interface.

F2407_EV1_FC_PWM_GEN
Constant initializer for the F2407 PWM Interface, EV1.

F2407_EV2_FC_PWM_GEN
Constant initializer for the F2407 PWM Interface, EV2.

Methods

void init (PWMGEN *)
Initializes the PWM Gen unit hardware.

void update(PWMGEN *)
Updates the PWM Generation hardware with the data from the PWM Structure.

Module Usage

Instantiation:
The interface to the PWM Generation Unit is instanced thus:

```
PWMGEN  gen;
```

Initialization:
To instance a pre-initialized object

```
PWMGEN  gen =PWMGEN_DEFAULTS
```

Hardware Initialization:

```
gen.init(&gen);
```

Invoking the update function:

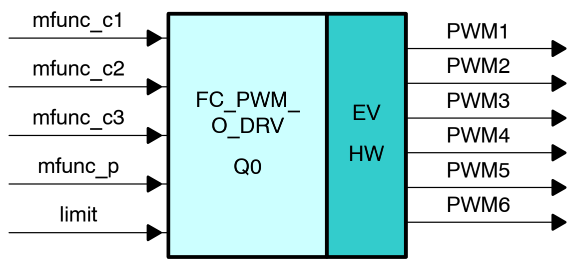
```
gen.update(&gen);
```

Example:
Lets instance one PWMGEN object and one SVGENMF object, (For details on SVGENMF see the SVGEN_MF.DOC.). The outputs of SVGENMF are output via the PWMGEN.

```
SVGENMF svgen= SVGEN_DEFAULTS;  /*Instance the space vector gen object */
PWMGEN  gen  = F243_FC_PWM_GEN; /*Instance the PWM interface object    */

main()
{
svgen.freq=1200;         /* Set properties for svgen */
gen.period_max=500;      /*Sets the prd reg for the Timer to 500 cycles*/
gen.init(&gen);          /* Call the hardware initialization function  */

}
void interrupt periodic_interrupt_isr()
{
sv1.calc(&sv1);          /* Call compute function for sv1 */

/* Lets output sv1.va,sv1.vb, and sv1.vc */

gen.mfunc_c1= svgen.va;  /*Connect the output of svgen to gen inputs*/
gen.mfunc_c2= svgen.vb;
gen.mfunc_c3= svgen.vc;

gen.update(&gen);        /* Call the hardware update function */
}
```

Description

The module implements over-modulation technique to increase DC bus voltage utilization for a voltage source inverter. The input *limit* sets the extent of over-modulation. For example, limit = 0 means no over-modulation and limit = (timer period)/2 means maximum over-modulation.



Availability

This module is available in the direct-mode assembly-only interface (Direct ASM).

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: pwmodrv.asm

Item	ASM Only	Comments
Code size	133 words	
Data RAM	11 words	
xDAIS module	No	
xDAIS component	No	IALG layer not implemented

Direct ASM Interface

Table 25. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	limit	Defines the level of over modulation. This is related to the PWM timer period.	Q0	0–timer_ period/2
	Mfunc_c1	Duty ratio for PWM1/PWM2	Q15	08000h– 7FFFh
	Mfunc_c2	Duty ratio for PWM3/PWM4	Q15	08000h– 7FFFh
	Mfunc_c3	Duty ratio for PWM5/PWM6	Q15	08000h– 7FFFh
	mfunc_p	PWM period modulation function	Q15	08000h– 7FFFh
H/W Outputs	PWMx (x=1,2,3,4,5,6)	Full compare PWM outputs from 24x/24xx device.	N/A	N/A
Init / Config	limit	Initial *limit* is set to 0 so that the system starts without any over-modulation. Specify *limit* for overmodulation.	Q0	0 – T1PER/2
	FPERIOD	PWM frequency select constant. Default value is set for 20kHz. Modify this constant for different PWM frequency.	Q0	Application dependent
	24x/24xx	Select appropriate 24x/24xx device from the x24x_app.h file.	N/A	N/A

Variable Declaration:
In the system file include the following statements:

```
.ref   FC_PWM_O_DRV
.ref   FC_PWM_O_DRV_INIT                          ;function call
.ref   Mfunc_c1, Mfunc_c2, Mfunc_c3, Mfunc_p     ;Inputs
```

Memory map:
All variables are mapped to an uninitialized named section “pwmodrv”

Example:

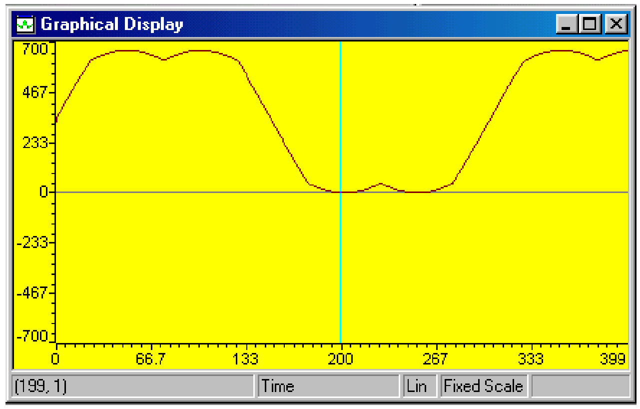
```
ldp  #mfunc_c1                  ;Set DP for module inputs
bldd #input_var1, mfunc_c1      ;Pass input variables to module inputs
bldd #input_var2, mfunc_c2
bldd #input_var3, mfunc_c3
bldd #input_var4, mfunc_p


CALL FC_PWM_O_DRV
```

Background Information

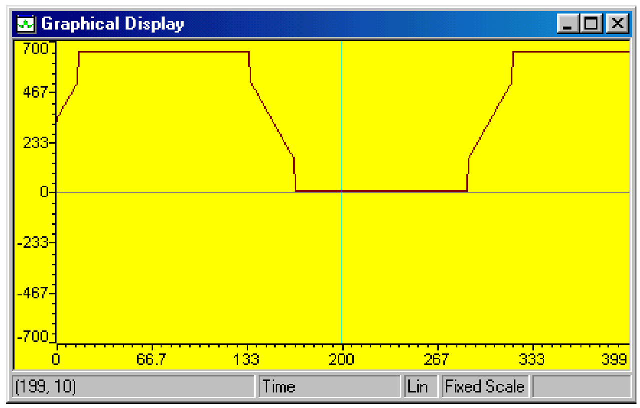
For high performance motor drive systems, full utilization of the dc bus voltage is an important factor to achieve maximum output torque under any operating conditions, and to extend the field weakening range of the motor. However, for a pulse-width modulated voltage source inverter (PWM−VSI), the maximum voltage is 78% of the six-step waveform value. Therefore, in general, a standard motor supplied from an inverter can not utilize the full DC bus voltage capability. To obtain higher DC bus voltage utilization, operating the inverter in over-modulation region is required.

This software module implements a simple but effective over-modulation scheme for PWM inverters. This module can be applied both for three phase drive (using Space Vector PWM or regular Sine PWM strategies) as well as single phase drive.

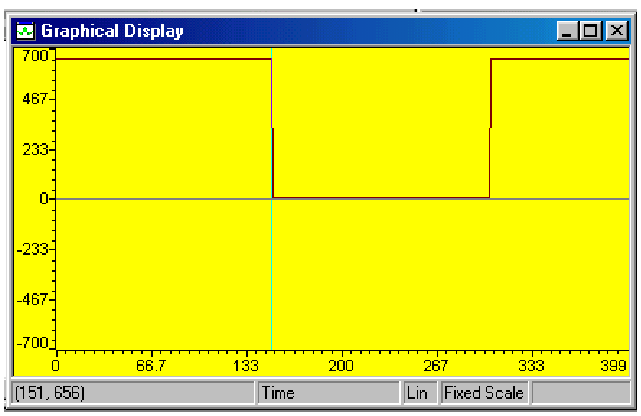
The level of over-modulation is controller by a variable called “limit”. Whenever, the ouptut waveform is within “limit”, the Compare values for PWM channels are saturated to the maximum value during the positive half of the waveform and to the minimum value during the negative half of the waveform. Figure 8 shows the effect of various values of “limit”.



(a)



(b)

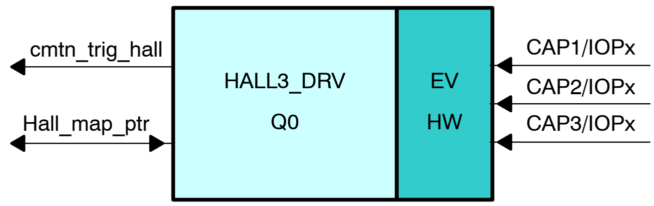


(c)

Figure 8. Implementation of Over-modulation Using the Software Module
(a) No over-modulation,
(b) Over-modulation with limit = T1PER/4,
(c) Maximum over-modulation (square wave) with limit = T1PER/2

Description

This module produces a commutation trigger for a 3-ph BLDC motor, based on hall signals received on capture pins 1, 2, and 3. Edges detected are validated or debounced, to eliminate false edges often occurring from motor oscillations. Hall signals can be connected in any order to CAPs1−3. The software attempts all (6) possible commutation states to initiate motor movement. Once the motor starts moving, commutation occurs on each debounced edge from received hall signals.



Availability

This module is available in the direct-mode assembly-only interface (Direct ASM).

Module Properties

Type: Target Dependent

Target Devices: x24x/x24xx

Assembly File Name: hall3_drv.asm

Item	ASM Only	Comments
Code size	170 words	
Data RAM	20 words	
xDAIS module	No	
xDAIS component	No	IALG layer not implemented

Direct ASM Interface

Table 26. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	CAP(1−3)/IOPx	Capture Inputs 1,2, and 3 (H/W)	N/A	N/A
	Hall_map_ptr	As an input, it is defined by MOD6_CNT.		
Outputs	cmtn_trig_hall	Commutation trigger for Mod6cnt input	Q0	0 or 1
	Hall_map_ptr	During hall map creation, this variable points to the current commutation state. After map creation, it points to the next commutation state.		
Init / Config	Select device	Select appropriate 24x/24xx device from the x24x_app.h file.	N/A	N/A

Variable Declaration:
In the system file include the following statements:

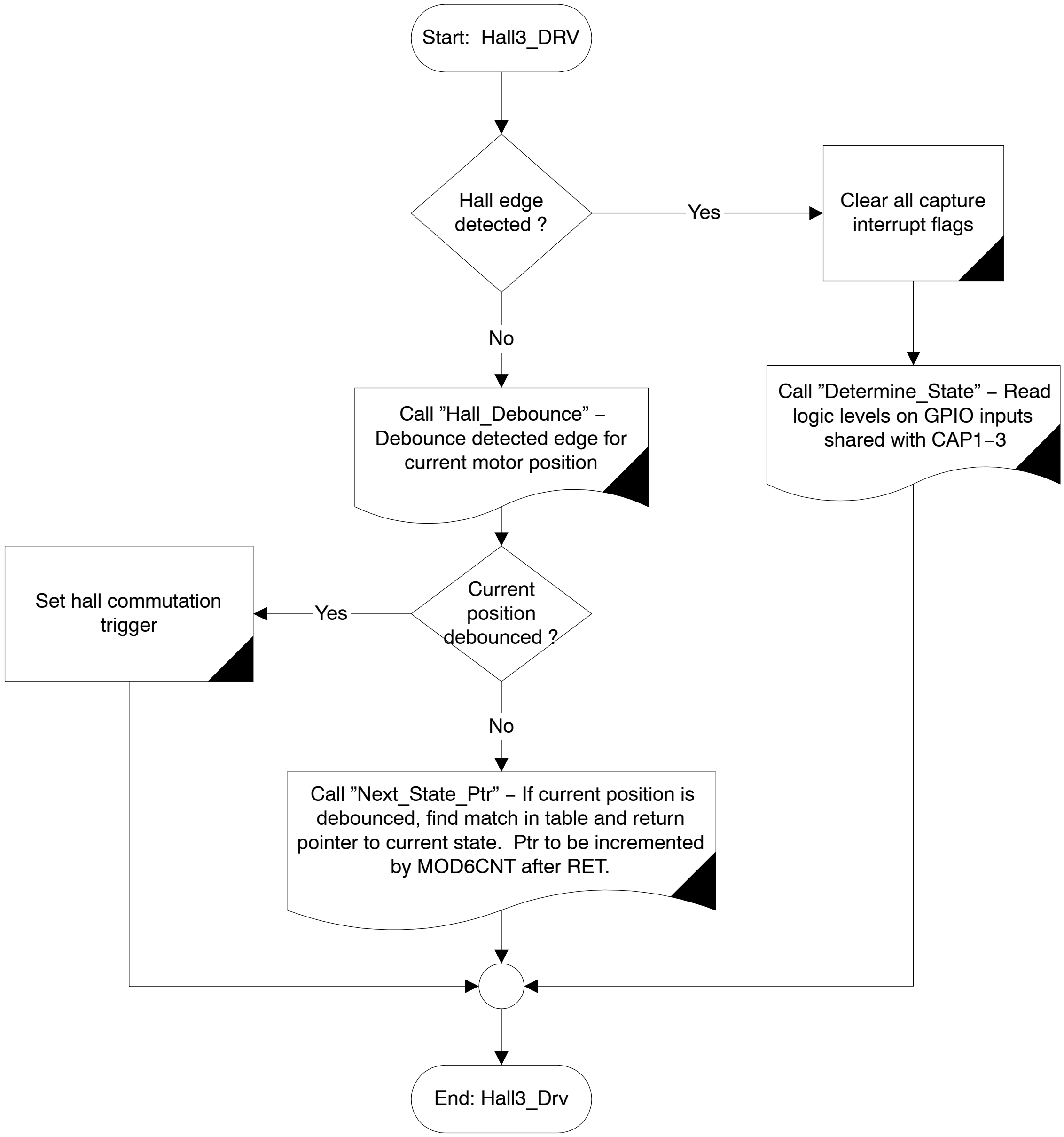
```
.ref   HALL3_DRV, HALL3_DRV_INIT      ;function call
.ref   cmtn_trig_hall, hall_map_ptr
```

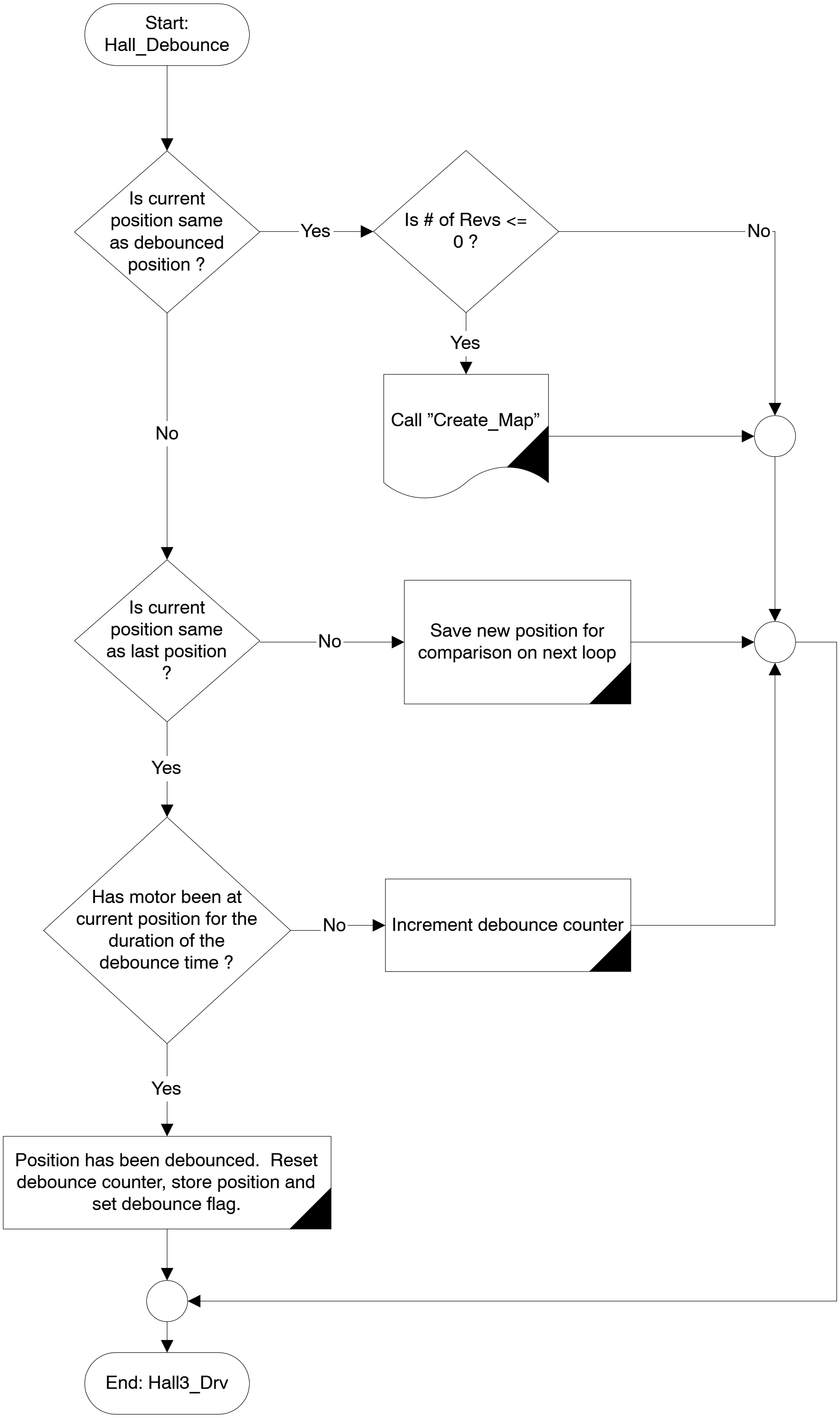
Memory map:
All variables are mapped to an uninitialized named section ‘HALL_VAR’.

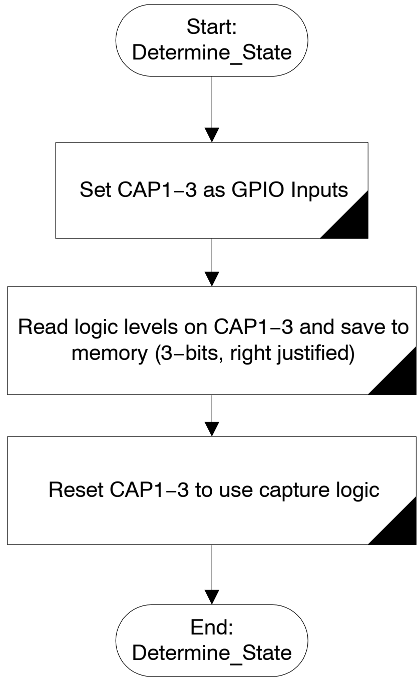
Example:

```
LDP  #hall_map_ptr
BLDD #input_var1, hall_map_ptr

CALL HALL3_DRV

LDP    #output_var1
BLDD #cmtn_trig_hall, output_var1
BLDD #hall_map_ptr, output_var2
```

Software Flowcharts

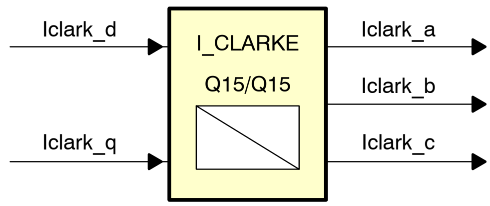



Start:
Hall_Debounce
Is current position same as debounced position ?
Yes
Is # of Revs <= 0 ?
No
Yes
Call ”Create_Map”
No
Is current position same as last position ?
No
Save new position for comparison on next loop
Yes
Has motor been at current position for the duration of the debounce time ?
No
Increment debounce counter
Yes
Position has been debounced. Reset debounce counter, store position and set debounce flag.
End: Hall3_Drv


Start:
Determine_State
Set CAP1−3 as GPIO Inputs
Read logic levels on CAP1−3 and save to memory (3−bits, right justified)
Reset CAP1−3 to use capture logic
End:
Determine_State

Description

Converts balanced two phase quadrature quantities into balanced three phase quantities.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Independent/Application Independent

Target Devices: x24x/x24xx

Direct ASM Version File Name: I_clarke.asm

C-Callable Version File Name: iclark.asm

Item	ASM Only	C-Callable ASM	Comments
Code size	21 words	32 words	
Data RAM	6 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented

† The inverse clark transform operates on structures allocated by the calling function.

Direct ASM Interface

Table 27. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	Iclark_d	Direct axis(d) component of the input two phase signal	Q15	8000−7FFF
	Iclark_q	Quadrature axis(q) component of the input two phase signal	Q15	8000−7FFF
Outputs	Iclark_a	Phase ‘a’ component of the transformed signal	Q15	8000−7FFF
	Iclark_b	Phase ‘b’ component of the transformed signal	Q15	8000−7FFF
	Iclark_c	Phase ‘c’ component of the transformed signal	Q15	8000−7FFF
Init / Config	none			

Variable Declaration:
In the system file include the following statements:

```
.ref I_CLARKE, I_CLARKE_INIT                           ;function call
.ref Iclark_d, Iclark_q, Iclark_a, Iclark_b, Iclark_c ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section ‘I_clarke’

Example:

```
ldp  #Iclark_d                ;Set DP for module input
bldd #input_var1, Iclark_d    ;Pass input variable to module input
bldd #input_var2, Iclark_q

CALL I_CLARKE

ldp  #output_var1             ;Set DP for output variable
bldd #Iclark_a, output_var1   ;Pass module output to output variable
bldd #Iclark_b, output_var2
bldd #Iclark_c, output_var3
```

C/C-Callable ASM Interface

This function is implemented as a function with two arguments, each a pointer to the input and output structures.

```
struct   { int d;
           int q;
         } iclark_in;

struct   { int a;
           int b;
           int c;
         } iclark_out;

  void iclark(&iclark_in,&iclark_out);
```

The inputs are read from the iclark_in structure and the outputs are placed in the iclark_out structure.

Table 28. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	d	Direct axis(d) component of the input two-phase signal.	Q15	8000−7FFF
	q	Quadrature axis(q) component of the input two-phase signal.	Q15	8000−7FFF
Outputs	a	Phase ‘a’ component of the transformed signal.	Q15	8000−7FFF
	b	Phase ‘b’ component of the transformed signal.	Q15	8000−7FFF
	c	Phase ‘c’ component of the transformed signal.	Q15	8000−7FFF
Init / Config	none			

Example:
In the following example, the variables intput_d, input_q are transformed to the output_a, output_b, and output_c

```
typedef struct { int a,b,c ; } triad;

triad threephase;
triad quadrature;

int input_d, input_q;
int output_a, output_b, output_c;

void some_func(void)
{
  quadrature.a=input_d;
  quadrature.b=input_q;

  iclark(&quadrature,&threephase);

  output_a=threephase.a;
  output_b=threephase.b;
  output_c=threephase.c;

}
```

Background Information

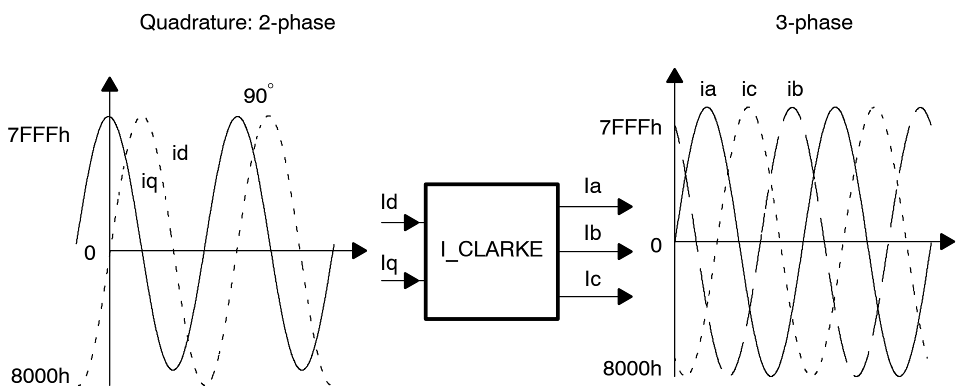
Implements the following equations:

$$\begin{cases} Ia = Id \\ Ib = \dfrac{-Id + Iq \times \sqrt{3}}{2} \\ Ic = \dfrac{-Id - Iq \times \sqrt{3}}{2} \end{cases}$$

Table 29. Variable Cross Ref Table

Variables in the Equations	Variables in the Code
Ia	Iclark_a
Ib	Iclark_b
Ic	Iclark_c
Id	Iclark_d
Iq	Iclark_q

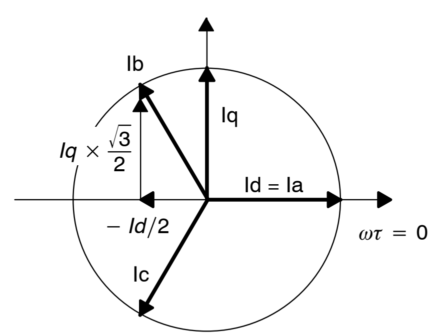
This transformation converts balanced two phase quadrature quantities into balanced three phase quantities as shown below:



The instantaneous input and the output quantities are defined by the following equations:

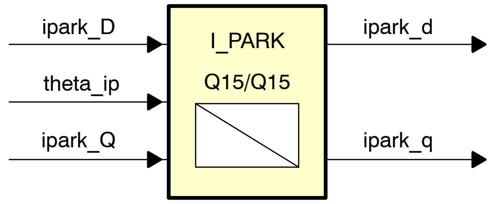
$$\begin{cases} id = I \times \sin(\omega t) \\ iq = I \times \sin(\omega t + \pi/2) \end{cases}$$

$$\begin{cases} ia = I \times \sin(\omega t) \\ ib = I \times \sin(\omega t + 2\pi/3) \\ ic = I \times \sin(\omega t - 2\pi/3) \end{cases}$$



Description

This transformation projects vectors in orthogonal rotating reference frame into two phase orthogonal stationary frame.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Independent/Application Independent

Target Devices: x24x/x24xx

Direct ASM Version File Name: I_park.asm

C-Callable Version File Name: ipark.asm

Item	ASM Only	C-Callable ASM	Comments
Code size	43 words	52 words	
Data RAM	12 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented

† The inverse park operates on structures allocated by the calling function.

Direct ASM Interface

Table 30. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	ipark_D	Direct axis(D) component of input in rotating reference frame.	Q15	8000−7FFF
	ipark_Q	Quadrature axis(Q) component of input in rotating reference frame	Q15	8000−7FFF
	theta_ip	Phase angle between stationary and rotating frame	Q15	0−7FFF (0−360 degree)
Outputs	ipark_d	Direct axis(d) component of transformed signal in stationary reference frame	Q15	8000−7FFF
	ipark_q	Quadrature axis(q) component of transformed signal in stationary reference frame	Q15	8000−7FFF
Init / **Config**	none			

Variable Declaration:
In the system file include the following statements:

```
.ref   I_PARK, I_PARK_INIT                            ;function call
.ref   ipark_D, ipark_Q, theta_ip, ipark_d, ipark_q   ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section ‘I_park’

Example:

```
ldp  #ipark_D                ;Set DP for module input
bldd #input_var1, ipark_D    ;Pass input variable to module input
bldd #input_var2, ipark_Q
bldd #input_var3, theta_ip

CALL I_PARK

ldp  #output_var1          ;Set DP for output variable
bldd #ipark_d, output_var1 ;Pass module o/p to output variable
bldd #ipark_q, output_var2
```

C/C-Callable ASM Interface

This function is implemented as a function with two arguments, each a pointer to the input and output structures.

```
struct   { int D;
           int Q;
           int theta;
         } ipark_in;

struct   { int d;
           int q;
         } ipark_out;

  void park(&ipark_in,&ipark_out);
```

The inputs are read from the park_in structure and the outputs are placed in the park_out structure.

Table 31. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	D	Direct axis(D) component of the input signal in rotating reference frame	Q15	8000−7FFF
	Q	Quadrature axis(Q) component of the input signal in rotating reference frame	Q15	8000−7FFF
	theta	Phase angle between stationary and rotating frame	Q15	0−7FFF (0−360 degree)
Outputs	d	Direct axis(d) component of transformed signal in stationary reference frame	Q15	8000−7FFF
	q	Quadrature axis(q) component of transformed signal in stationary reference frame	Q15	8000−7FFF
Init / Config	none			

Example:

In the following example, the variables rotating_d, rotating_q, are transformed to the stationery frame values based on theta_value.

```
typedef struct { int a,b,c ; } triad;

triad stationery_cmds;
triad rotating_cmds;

int stat_D,stat_Q;
int rotating_d,rotating_q,theta_value;

void some_func(void)
{
  rotating_cmds.a = rotating_d;
  rotating_cmds.b = rotating_q;

  park(&stationary_cmds,&rotating_cmds);
```

```
  stat_d = stationary_cmds.a;
  stationary_cmds.b = stat_q;
}
```

Background Information

Implements the following equations:

$$\begin{cases} Id = ID \times \cos\theta - IQ \times \sin\theta \\ Iq = ID \times \sin\theta + IQ \times \cos\theta \end{cases}$$

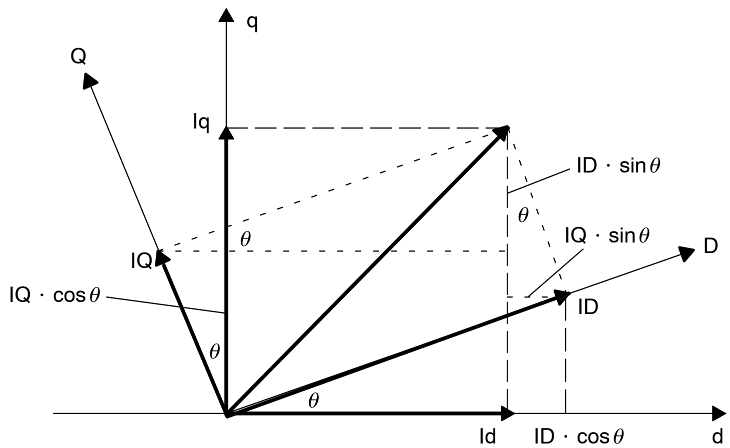


Table 32. Variable Cross Ref Table

Variables in the Equations	Variables in the Code
Id	ipark_d
Iq	ipark_q
θ	theta_ip
ID	ipark_D
IQ	ipark_Q

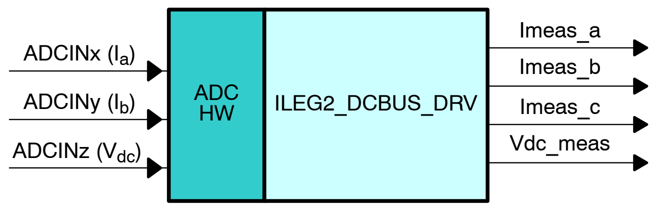
Description

This module allows 3-channel analog-to-digital conversion with programmable gains and offsets. The conversions are triggered on GP Timer 1 underflow. The converted results represent load currents and DC-bus voltage in the inverter when:

1) GP Timer 1 is the time base for symmetrical Pulse-Width Modulation (PWM);

2) Two of the analog inputs are the amplified voltage across resistors placed between the sources or emitters of low-side power devices and low-side DC rail; and

3) The third analog input is derived from the output of the voltage divider circuit connected across the DC bus.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Dependent/Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: i2vd_drv.asm

ASM Routines: ILEG2_DCBUS_DRV, ILEG2_DCBUS_DRV_INIT

C-callable ASM filenames: F07ILVD1.ASM, F07ILVD2.C, F07ILVD.h (for x24xx only)

Item	ASM Only	C-Callable ASM	Comments
Code size	87 words	103 words†	
Data RAM	13 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized ILEG2DCBUSMEAS structure instance consumes 13 words in the data memory and 15 words in the .cinit section.

Direct ASM Interface

Table 33. Module Terminal Variables/Functions

	Name	Description	Default	Format	Range	Scale
H/W Inputs	ADCINx, ADCINy, ADCINz	ADC pins in 24x/24xx device where x,y,z correspond to the channel numbers selected by Ch_sel	N/A	N/A	N/A	N/A
Outputs	Imeas_a	x^{th} channel digital representation for current I_a	N/A	Q15	−1.0 –> 0.999	Imax
	Imeas_b	y^{th} channel digital representation for current I_b	N/A	Q15	−1.0 –> 0.999	Imax
	Imeas_c	Computing current I_c	N/A	Q15	−1.0 –> 0.999	Imax
	Vdc_meas	z^{th} channel digital representation for DC-bus voltage V_{dc}	N/A	Q15	−1.0 –> 0.999	Vmax
Init / **Config**	Ch_sel	16-bit ADC channel select format can be seen as: Ch_sel = 0zyxh	0710h (243EVM), 0D32h (2407EVM)	Q0	x, y, z are between 0h –> Fh	N/A
	Imeas_a_ gain	Gain for x^{th} channel. Modify this if default gain is not used.	1FFFh (0.999)	Q13	−4.0 –> 3.999	N/A
	Imeas_b_ gain	Gain for y^{th} channel. Modify this if default gain is not used.	1FFFh (0.999)	Q13	−4.0 –> 3.999	N/A
	Vdc_meas_ gain	Gain for z^{th} channel. Modify this if default gain is not used.	1FFFh (0.999)	Q13	−4.0 –> 3.999	N/A
	Imeas_a_ offset	Offset for x^{th} channel. Modify this if default offset is not used.	0000h (0.000)	Q15	−1.0 –> 0.999	Imax
	Imeas_b_ offset	Offset for y^{th} channel. Modify this if default offset is not used.	0000h (0.000)	Q15	−1.0 –> 0.999	Imax
	Vdc_meas_ offset	Offset for z^{th} channel. Modify this if default offset is not used.	0000h (0.000)	Q15	−1.0 –> 0.999	Vmax

Routine names and calling limitation:
There are two routines involved:

- ILEG2_DCBUS_DRV, the main routine
- ILEG2_DCBUS_DRV_INIT, the initialization routine

The initialization routine must be called during program initialization. The ILEG2_DCBUS_DRV routine must be called in the control loop. The ILEG2_DCBUS_DRV must be called in GP Timer 1 underflow interrupt service routine.

Variable Declaration:
In the system file, including the following statements before calling the subroutines:

```
.ref   ILEG2_DCBUS_DRV, ILEG2_DCBUS_DRV_INIT          ;function call
.ref   Ch_sel, Imeas_a_gain, Imeas_b_gain, Vdc_meas_gain  ;Inputs
.ref   Imeas_a_offset, Imeas_b_offset, Vdc_meas_offset    ;Inputs
.ref   Imeas_a, Imeas_b, Imeas_c, Vdc_meas                ;Outputs
```

Memory map:
All variables are mapped to an uninitialized named section, i2vd_drv, which can be allocated to any one data page.

Example:
During system initialization specify the ILEG2_DCBUS_DRV parameters as follows:

```
LDP  #Ch_sel                ;Set DP for module inputs
SPLK #0D32h, Ch_sel         ;Select ADC channels. In this example
                            ;three channels selected are 13, 3 and 2.
SPLK #GAIN1, Imeas_a_gain   ;Specify gain value for each channel
SPLK #GAIN2, Imeas_b_gain
SPLK #GAIN3, Vdc_meas_gain
SPLK #OFFS1, Imeas_a_offset ;Specify offset value for each channel
SPLK #OFFS2, Imeas_b_offset
SPLK #OFFS3, Vdc_meas_offset
```

Then in the interrupt service routine call the module and read results as follows:

```
CALL ILEG2_DCBUS_DRV
LDP  #output_var1           ;Set DP for output variables
BLDD #Imeas_a, output_var1  ;Pass module outputs to output variables
BLDD #Imeas_b, output_var2  ;Pass module outputs to output variables
BLDD #Imeas_c, output_var3  ;Pass module outputs to output variables
BLDD #Vdc_meas, output_var4 ;Pass module outputs to output variables
```

C/C-Callable ASM Interface

Object Definition

The structure of the ILEG2DCBUSMEAS object is defined in the header file, F07ILVD.h, as seen in the following:

```
typedef struct  { int Imeas_a_gain;           /* Parameter: gain for Ia (Q13) */
                  int Imeas_a_offset;         /* Parameter: offset for Ia (Q15) */
                  int Imeas_a;                /* Output: measured Ia (Q15) */
                  int Imeas_b_gain;           /* Parameter: gain for Ib (Q13) */
                  int Imeas_b_offset;         /* Parameter: offset for Ib (Q15) */
                  int Imeas_b;                /* Output: measured Ib (Q15) */
                  int Vdc_meas_gain;          /* Parameter: gain for Vdc (Q13) */
                  int Vdc_meas_offset;        /* Parameter: offset for Vdc (Q15) */
                  int Vdc_meas;               /* Output: measured Vdc (Q15) */
                  int Imeas_c;                /* Output: computed Ic (Q15) */
                  int Ch_sel;                 /* Parameter: ADC channel selection */
                  int (*init)();              /* Pointer to the init function */
                  int (*read)();              /* Pointer to the read function */
                } ILEG2DCBUSMEAS;
```

Special Constants and Datatypes

ILEG2DCBUSMEAS
The module definition itself is created as a data type. This makes it convenient to instance ILEG2DCBUSMEAS object. To create multiple instances of the module simply declare variables of type ILEG2DCBUSMEAS.

ILEG2DCBUSMEAS_DEFAULTS
Initializer for the ILEG2DCBUSMEAS object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, F07ILVD.h.

Methods

void init(ILEG2DCBUSMEAS *);
void read(ILEG2DCBUSMEAS *);
This default definition of the object implements two methods – the initialization and the runtime compute function for Q15 conversion, and gain/offset calculation. This is implemented by means of a function pointer, and the initializer sets this to F2407_ileg2_dcbus_drv_init and F2407_ileg2_dcbus_drv_read functions. The argument to this function is the address of the ILEG2DCBUSMEAS object. Again, this statement is written in the header file, F07ILVD.h. The F2407_ileg2_dcbus_drv_init module is implemented in F07IlVD1.C and the F2407_ileg2_dcbus_drv_read module is implemented in F07IlVD2.ASM.

Module Usage

Instantiation:

The following example instances two such objects:

ILEG2DCBUSMEAS ilg2_vdc1, ilg2_vdc2;

Initialization:

To instance a pre-initialized object:

ILEG2DCBUSMEAS ilg2_vdc1 = ILEG2DCBUSMEAS_DEFAULTS;
ILEG2DCBUSMEAS ilg2_vdc2 = ILEG2DCBUSMEAS_DEFAULTS;

Invoking the compute function:

```
ilg2_vdc1.calc(&ilg2_vdc1);
ilg2_vdc2.calc(&ilg2_vdc2);
```

Example:

Lets instance two ILEG2DCBUSMEAS objects, otherwise identical, and run two independent ADC sequences. The following example is the c source code for the system file.

```
  /* instance the first object */
ILEG2DCBUSMEAS ilg2_vdc1 = ILEG2DCBUSMEAS_DEFAULTS;
  /* instance the second object */
ILEG2DCBUSMEAS ilg2_vdc2 = ILEG2DCBUSMEAS_DEFAULTS;

main()
{

  ilg2_vdc1.init(&ilg2_vdc1);              /* Call init function for ilg2_vdc1 */
  ilg2_vdc2.init(&ilg2_vdc2);              /* Call init function for ilg2_vdc2 */

}

void interrupt periodic_interrupt_isr()
{

  ilg2_vdc1.read(&ilg2_vdc1);              /* Call compute function for ilg2_vdc1 */
  ilg2_vdc2.read(&ilg2_vdc2);              /* Call compute function for ilg2_vdc2 */

  current_abc1.a = ilg2_vdc1.Imeas_a;      /* Access the outputs of ilg2_vdc1 */
  current_abc1.b = ilg2_vdc1.Imeas_b;      /* Access the outputs of ilg2_vdc1 */
  current_abc1.c = ilg2_vdc1.Imeas_c;      /* Access the outputs of ilg2_vdc1 */
volt1.DC_bus=ilg2_vdc1.Vdc_meas;           /* Access the outputs of ilg2_vdc1 */

  current_abc2.a = ilg2_vdc2.Imeas_a;      /* Access the outputs of ilg2_vdc2 */
  current_abc2.b = ilg2_vdc2.Imeas_b;      /* Access the outputs of ilg2_vdc2 */
  current_abc2.c = ilg2_vdc2.Imeas_c;      /* Access the outputs of ilg2_vdc2 */
  volt2.DC_bus=ilg2_vdc2.Vdc_meas;         /* Access the outputs of ilg2_vdc2 */

}
```

Background Information

The ADCIN pins accepts the analog input signals (I_a, I_b, and V_{dc}) in the following range:

- 0.0−5.0 volt; for x24x based DSP
- 0.0−3.3 volt; for x240x based DSP

with ground referenced to 0.0 volt.

Therefore, the current and voltage signals need to be conditioned properly before they are applied to the ADC pins.

From the three converted signals, four output variables of the module (Imeas_a, Imeas_b, Imeas_c, and Vdc_meas) are computed, as shown below:

Imeas_a = Imeas_a_gain*ADC_Ia_Q15 + Imeas_a_offset
Imeas_b = Imeas_b_gain*ADC_Ib_Q15 + Imeas_b_offset
Imeas_c = −(Imeas_a + Imeas_b)
Vdc_meas = Vdc_meas_gain*ADC_Vdc_Q15 + Vdc_meas_offset

Note that ADC_Ix_Q15 (x=a,b) and ADC_Vdc_Q15 are already converted to Q15 number.

Basically, the signals can be categorized into two main types: bipolar and unipolar signals. The AC currents (or AC voltages) are examples of bipolar signal and the DC-bus voltage is an example of unipolar signal.

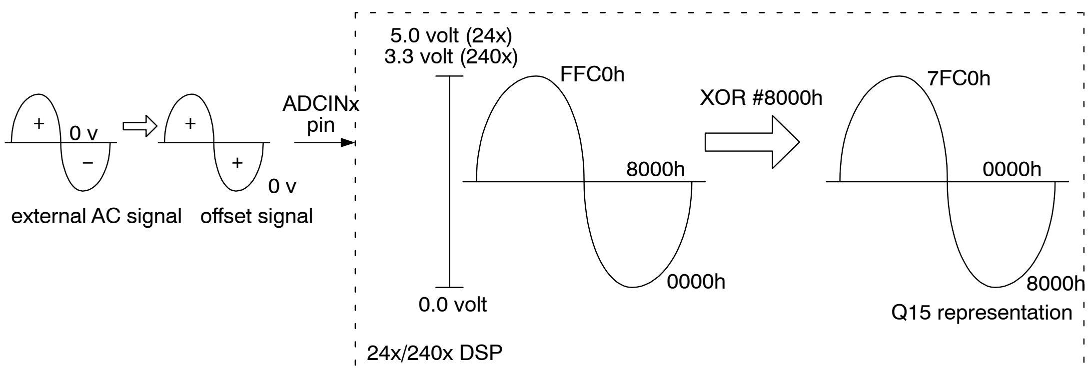


Figure 9. Q15-Number Conversion for Current Measurements (bipolar signal)

For DC-bus voltage (V_{dc}), the input signal is in the positive range, so its digitized variable has to be rescaled corresponding to the Q15 number. Figure 10 illustrates the Q15-number conversion for the DC-bus voltage measurement.

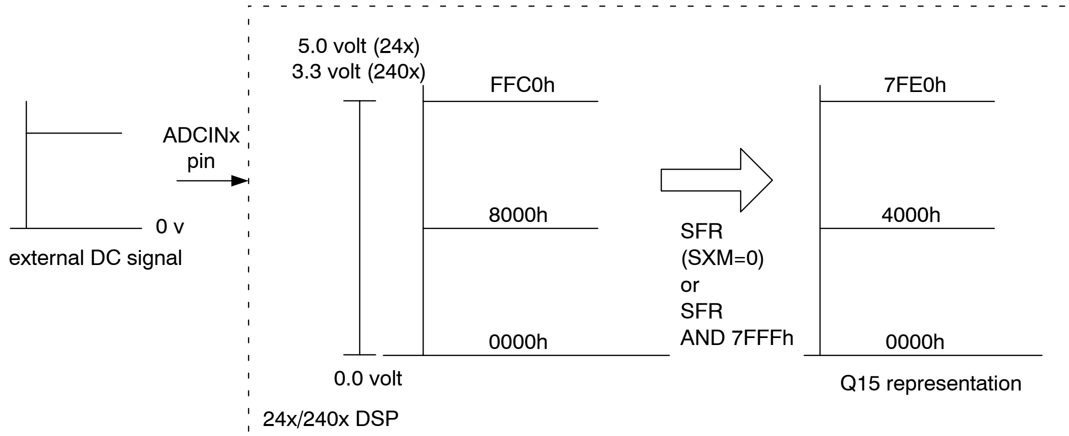


Figure 10. Q15-Number Conversion for DC-Bus Voltage Measurement (unipolar signal)

In both cases of Q15-number conversion, the number is distorted a little bit about the maximum value (e.g., 7FC0h for bipolar and 7FE0h for unipolar at the maximum value of 7FFFh).

ILEG2DRV	*Dual Inverter Leg Resistor Based Load Current Measurement Driver*

Description

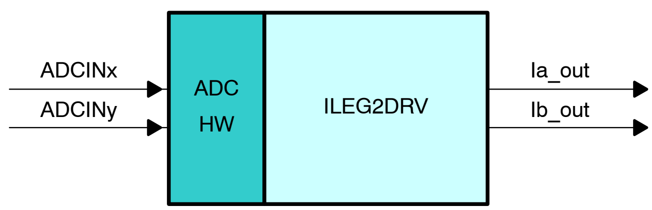
Ileg2drv is a driver module that converts two analog inputs into digital representations with programmable gains and offsets. The conversions are triggered on GP Timer 1 underflow. The converted results represent load currents of a three-phase voltage source inverter when:

1) Symmetrical Pulse-Width Modulation (PWM) is used to control the inverter with GP Timer 1 as PWM time base;

2) PWM outputs 1, 3 and 5 control the turn-on and off of the upper power devices;

3) PWM outputs 2, 4, and 6 control the turn-on and off of the lower power devices; and

4) The analog inputs are the amplified and filtered voltage outputs of resistors placed between the sources or emitters of low-side power devices and low-side DC rail.



Availability

This module is available in two interface formats:

5) The direct-mode assembly-only interface (Direct ASM)

6) The C-callable interface version

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: ileg2drv

Routines: ileg2drv, ileg2drv_init

Item	ASM Only	C-Callable ASM	Comments
Code size	62 words	TBD	
Data RAM	8 words	TBD	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented

Direct ASM Interface

Table 34. Module Terminal Variables/Functions

	Name	Description	Default	Format	Range	Scale
H/W Inputs	ADCINx, ADCINy	ADC pins in 24x/24xx device where x and y correspond to the channel numbers selected by A4_ch_sel	N/A	N/A	N/A	N/A
Outputs	Ia_out	1st channel digital representation	N/A	Q15	−1.0 –> 0.999	Imax
	Ib_out	2nd channel digital representation	N/A	Q15	−1.0 –> 0.999	Imax
Init / Config	I_ch_sel	Channel select variable. Init this in the form of *XY*h with *X* being the1st channel, and *Y* being the 2nd channel.	*XY*h: 10h for 24x, 40h for 240x	Q0	X,Y: 0 –> Fh	N/A
	Ia_gain	Gain for 1st channel. Modify this if default gain is not used.	1FFFh (1.0)	Q13	−4.0 –> 3.999	N/A
	Ib_gain	Gain for 2nd channel. Modify this if default gain is not used.	1FFFh (1.0)	Q13	−4.0 –> 3.999	N/A
	Ia_offset	Offset for 1st channel. Modify this if default offset is not used.	32 (0.001)	Q15	−1.0 –> 0.999	Imax
	Ib_offset	Offset for 2nd channel. Modify this if default offset is not used.	32 (0.001)	Q15	−1.0 –> 0.999	Imax

Routine names and calling limitation:
There are two routines involved:

ILEG2DRV, the main routine, and
ILEG2DRV_INIT, the initialization routine.

The initialization routine must be called during program (or incremental build) initialization. The ILEG2DRV must be called in GP Timer 1 underflow interrupt service routine.

Global reference declarations:
In the system file include the following statements before calling the subroutines:

```
.ref ILEG2DRV,ILEG2DRV_INIT                  ; function calls
.ref Ia_out,Ib_out                           ; Outputs
.ref Ia_gain,Ib_gain,Ia_offset,Ib_offset     ; Inputs
```

Memory map:
All variables are mapped to an uninitialized named section, *ileg2drv*, which can be allocated to any one data page.

Example:

```
CALL ILEG2DRV_INIT          ; Initialize ILEG2DRV
Splk #GAIN_CH1,Ia_gain      ; Initialize gain for 1st channel
…                           ; Use default values for other inputs

ldp  #Ia_gain               ; Set DP for module inputs
bldd #input_var1,Ia_gain    ; Pass input variables to module inputs
bldd #input_var2,Ib_gain    ;
…                           ; Use default values for other inputs

CALL ILEG2_DRV

ldp  #output_var1           ; Set DP for output variable
bldd #Ia_out,output_var1    ; Pass output to other variable
…                           ; Pass more outputs to other variables
                            ; if needed.
```

C/C-Callable ASM Interface

TBD

Background Information

Figure 11 is an illustration of using 24x or 240x to measure the load currents of a three-phase inverter driving a three-phase load. The currents to be measured are *Ia*, *Ib* and *Ic*. In most cases, the three-phase load has a floating neutral, which means only two load currents must be measured and the third is simply the negative of the sum of the two measured ones. Indeed this is true in most three-phase motor control applications.

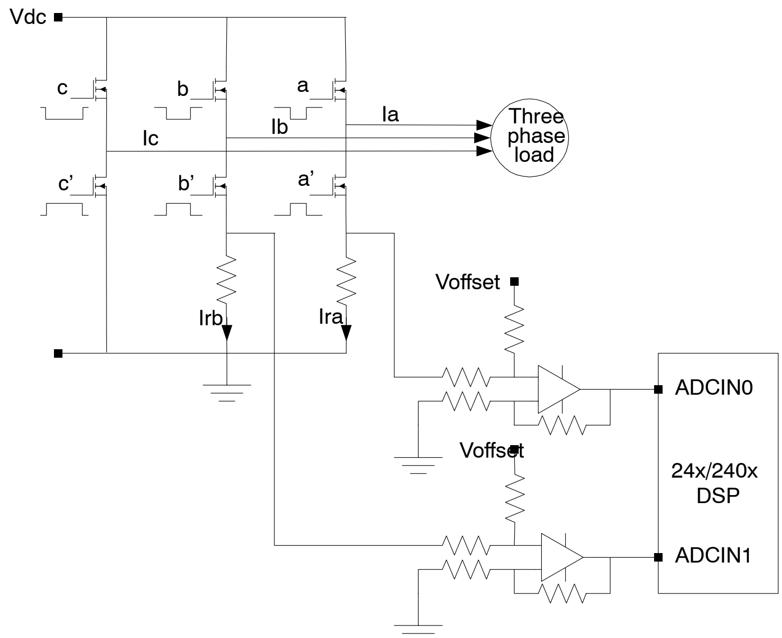


Figure 11. Inverter Load Current Measurement

As shown in Figure 11, two (low-resistance) resistors are connected in between the source (or emitter) of the low-side power devices and low-side DC rail. Note that the low-side DC rail is assumed to be the ground reference. The voltages across these two resistors are amplified and level shifted to generate an output range within *Vref_lo* and *Vref_hi* (typically 0 to 5V for 24x and 0 to 3.3V for 240x). They are then fed into the ADC inputs of the '24x or '240x device. The inputs are converted into digital representations once every PWM period. Since the resistors have known resistance, the converted results represent currents flowing through the resistors at the time the samples are taken. According to Table 35, the current flowing through a leg resistor represents the load current of the inverter leg whenever the high-side power device is off and the low-side power device is on. Therefore, to obtain measurement of the load current, the sample must be taken when the corresponding high-side power device is off.

Table 35. Leg Current vs Switching State

a	a'	Ira	b	b'	Ib
1	0	0	1	0	0
0	1	Ia	0	1	Irb

Figure 12 indicates how symmetric PWM is achieved with up-and-down counting mode of GP Timer 1 of ‘24x and ‘240x. It can be seen that the high-side power device is always off on GP Timer 1 underflow. Therefore, the Start of Conversion (SOC) is configured to be underflow of GP Timer 1.

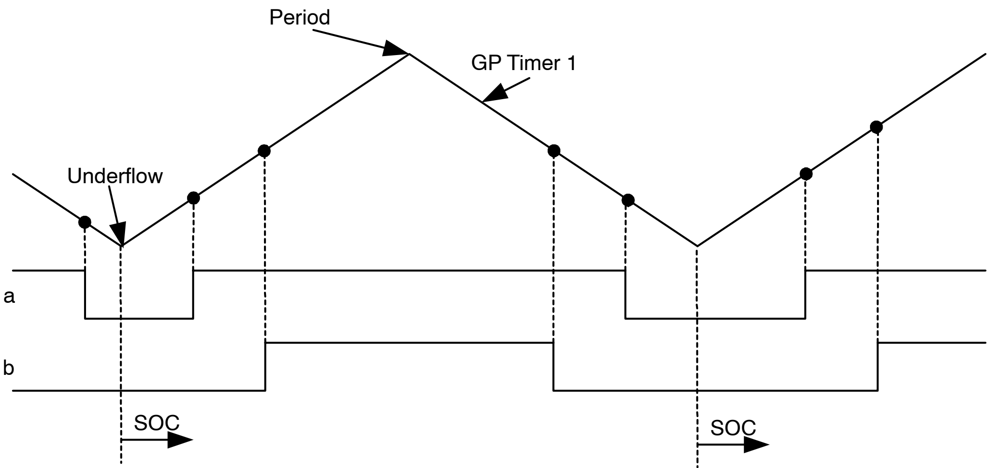


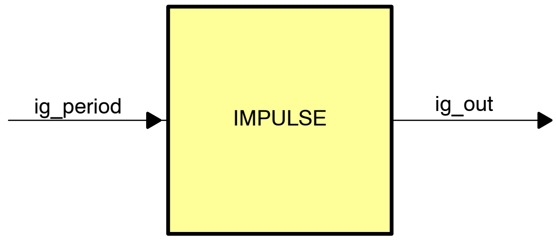
Figure 12. Symmetric PWM and Load Current Sampling

In addition to allowing selection of different ADC input channels, the module also allow different offsets and gains to be applied to the converted results. The offset and gain can be used to convert the outputs to a different Q format.

The default configuration assumes that the external Op-Amp circuit applies *Vref_hi* to the ADC when load current is at *Imax*, and *Vref_lo* when load current is at *–Imax*. Note, before the software offsets and gains, the converted result is 8000h (−1.0 as a Q15 number) when input voltage is *Vref_lo*, and 7FC0h (~0.998 as a Q15 number) when input voltage is *Vref_hi*. To make the result symmetric with respect to 0, an offset of 32 (~0.001 as a Q15 number) is used as the default for both channels.

Description

This module implements a periodic impulse function. The output variable *ig_out* is set to 7FFF for 1 sampling period. The period of the output signal *ig_out* is specified by the input *ig_period*.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Independent

Target Devices: x24x / x24xx

Assembly File Name: impulse.asm

C-Callable Version File Name: impulse.asm, impl.h

Item	ASM Only	C-Callable ASM	Comments
Code size	20 words	30 words†	
Data RAM	3 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized IMPULSE structure instance consumes 4 words in the dta memory and 6 words in the .cinit section.

Direct ASM Interface

Table 36. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	ig_period	Period of output impulses in number of sampling cycles	Q0	0–7FFFh
Output	ig_out	Impulse generator output	Q0	0 or 7FFFh
Init / Config	none			

Variable Declaration:
In the system file include the following statements:

```
.ref   IMPULSE, IMPULSE_INIT      ;function call
.ref   ig_period, ig_out          ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section 'impulse'

Example:

```
ldp  #ig_period               ;Set DP for module input
bldd #input_var1, ig_period   ;Pass input variable to module input

CALL IMPULSE

ldp  #out_var1                ;Set DP for output variable
bldd #ig_out, output_var1     ;Pass module output to output variable
```

C/C-Callable ASM Interface

Object Definition

The structure of the IMPULSE Object is defined by the following structure definition

```
/*-----------------------------------------------------------------------------
Define the structure of the IMPULSE
(Impulse generator)
-----------------------------------------------------------------------------*/

typedef struct  {
  int period;     /* Period of output impulses in number of sampling cycles */
  int out;        /* Impulse generator output                               */
  int skpcnt;
  int (*calc)();  /* Pointer to the Calculation function                    */
  }IMPULSE;
```

Table 37. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	period	Period of output impulses in number of sampling cycles	Q0	0−7FFFh
Output	out	Impulse generator output	Q0	0 or 7FFFh

Special Constants and Datatypes

IMPULSE
The module definition itself is created as a data type. This makes it convenient to instance a Impulse generator module.To create multiple instances of the module simply declare variables of type IMPULSE

IMPULSE_handle
Typedef'ed to IMPULSE *

IMPULSE_DEFAULTS
Initializer for the IMPULSE Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc (IMPULSE_handle)
The default definition of the object implements just one method – the runtime implementation of the Impulse generator. This is implemented by means of a function pointer, and the default initializer sets this to impulse_calc. The argument to this function is the address of the IMPULSE object.

Module Usage

Instantiation:
The following example instances two such objects:

```
IMPULSE p1,p2;
```

Initialization:
To instance a pre-initialized object

```
IMPULSE p1 = IMPULSE_DEFAULTS, p1 = IMPULSE_DEFAULTS;
```

Invoking the compute function:

```
p1.calc(&p1);
```

Example:

Lets instance two IMPULSE objects,otherwise identical ,but running with different values

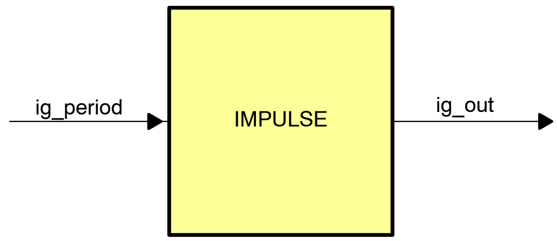
```
IMPULSE p1 = IMPULSE_DEFAULTS; /* Instance the first  object */
IMPULSE p2 = IMPULSE_DEFAULTS; /* Instance the second object */

main()
{
    p1.period =300;              /* Initialize  */
    p2.period =400;              /* Initialize  */

}
void interrupt periodic_interrupt_isr()
{
    (*p1.calc)(&p1);             /* Call compute function for p1 */
    (*p2.calc)(&p2);             /* Call compute function for p2 */

    x = p1. out; /* Access the output of p1 */

    q = p2. out; /* Access the output of p2 */

  /* Do something with the outputs */

}
```

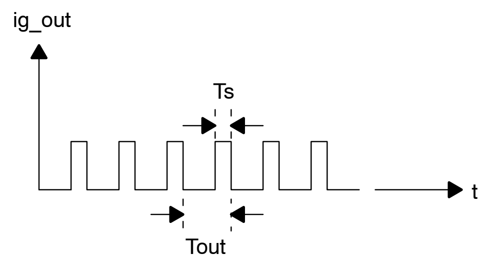
Background Information

Implements the following equation:

ig_out = 7FFF, for t = n . Tout, n = 1, 2, 3, …
= 0, otherwise

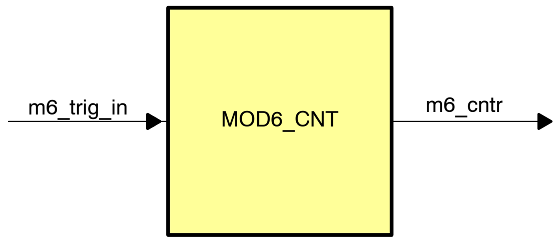
where,
Tout = Time period of output pulses = *ig_period* x Ts
Ts = Sampling time period





Description

This module implements a modulo 6 counter. It counts from state 0 through 5, then resets to 0 and repeats the process. The state of the output variable *m6_cntr* changes to the next state every time it receives a trigger input through the input variable *m6_trig_in*.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Independent

Target Devices: x24x / x24xx

Assembly File Name: mod6_cnt.asm

C-Callable Version File Name: mod6_cnt.asm, mod6.h

Item	ASM Only	C-Callable ASM	Comments
Code size	22 words	28 words†	
Data RAM	2 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized MOD6CNT structure instance consumes 3 words in the data memory and 5 words in the .cinit section.

Direct ASM Interface

Table 38. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	m6_trig_in	Modulo 6 counter trigger input	Q0	0 or 7FFFh
Output	m6_cntr	Modulo 6 counter output	Q0	0=<m6_cntr=<5
Init / Config	none			

Variable Declaration:
In the system file include the following statements:

```
.ref   MOD6_CNT, MOD6_CNT_INIT      ;function call
.ref   m6_trig_in, m6_cntr          ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section ‘mod6_cnt’

Example:

```
ldp  #m6_trig_in              ;Set DP for module input
bldd #input_var1, m6_trig_in  ;Pass input variable to module input

CALL MOD6_CNT

ldp  #output_var1             ;Set DP for output variable
bldd #m6_cntr, output_var1    ;Pass module output to output variable
```

C/C-Callable ASM Interface

Object Definition

The structure of the MOD6CNT Object is defined by the following structure definition

```
/*-----------------------------------------------------------------------------
Define the structure of the MOD6CNT
(Modulo6 counter)
-----------------------------------------------------------------------------*/

typedef struct  { int trig_in;      /* Modulo 6 counter trigger input  */
                  int cntr;         /* Modulo 6 counter output */
                  int  (*calc)();   /* pointer to the calculation function */
} MOD6CNT;
```

Table 39. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	trig_in	Modulo 6 counter trigger input	Q0	0 or 7FFFh
Output	cntr	Modulo 6 counter output	Q0	0=<cntr=<5

Special Constants and Datatypes

MOD6CNT
The module definition itself is created as a data type. This makes it convenient to instance a modulo6 counter module.To create multiple instances of the module simply declare variables of type MOD6CNT

MOD6CNT_handle
Typedef'ed to MOD6CNT *

MOD6CNT_DEFAULTS
Initializer for the MOD6CNT Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(MOD6CNT_handle)
The default definition of the object implements just one method – the runtime implementation of the modulo6 counter. This is implemented by means of a function pointer, and the default initializer sets this to mod6cnt_calc. The argument to this function is the address of the MOD6CNT object.

Module Usage

Instantiation:
The following example instances two such objects:

```
MOD6CNT p1,p2;
```

Initialization:
To instance a pre-initialized object

```
MOD6CNT p1 = MOD6CNT_DEFAULTS, p2 = MOD6CNT_DEFAULTS;
```

Invoking the compute function:

```
p1.calc(&p1);
```

Example:

Lets instance two MOD6CNT objects,otherwise identical ,but running with different values

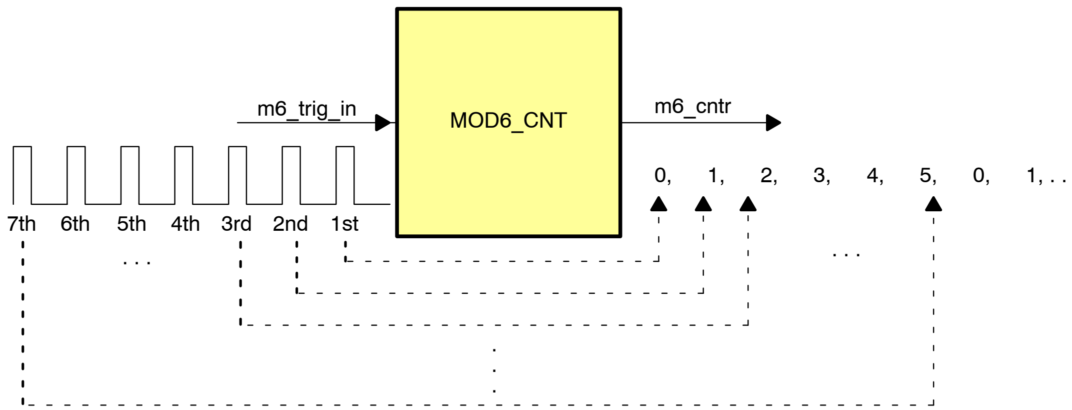
```
MOD6CNT p1 = MOD6CNT_DEFAULTS; /* Instance the first  object */
MOD6CNT p2 = MOD6CNT_DEFAULTS; /* Instance the second object */

main()
{
     p1.cntr = 3;                  /* Initialize  */
     p1.trig_in = 0x0200;

     p2.cntr = 4;                  /* Initialize  */
     p2.trig_in = 0x1500;

}
void interrupt periodic_interrupt_isr()
{
     (*p1.calc)(&p1);       /* Call compute function for p1 */
     (*p2.calc)(&p2);       /* Call compute function for p2 */

     x = p1.cntr; /* Access the output of p1 */

     q = p2.cntr; /* Access the output of p2 */

   /* Do something with the outputs */

}
```

Background Information

Implements the following equation:

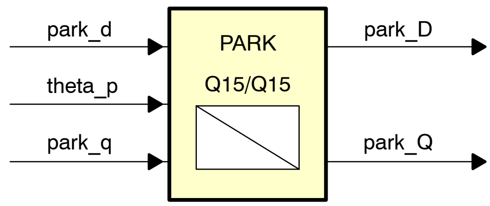
m6_cntr = 0, when 1st trigger pulse occur (*m6_trig_in* is set to 7FFF for the 1st time)
= 1, when 2nd trigger pulse occur (*m6_trig_in* is set to 7FFF for the 2nd time)
= 2, when 3rd trigger pulse occur (*m6_trig_in* is set to 7FFF for the 3rd time)
= 3, when 4th trigger pulse occur (*m6_trig_in* is set to 7FFF for the 4th time)
= 4, when 5th trigger pulse occur (*m6_trig_in* is set to 7FFF for the 5th time)
= 5, when 6th trigger pulse occur (*m6_trig_in* is set to 7FFF for the 6th time)

and repeats the output states for the subsequent pulses.



Description

This transformation converts vectors in balanced 2-phase orthogonal stationary system into orthogonal rotating reference frame.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Independent/Application Independent

Target Devices: x24x/x24xx

Direct ASM Version File Name: park.asm

C-Callable Version File Name: park.asm

Item	ASM Only	C-Callable ASM	Comments
Code size	36 words	52 words	
Data RAM	12 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented

† The park transform operates on structures allocated by the calling function.

Direct ASM Interface

Table 40. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	park_d	Direct axis(d) component of the input signal in stationary reference frame	Q15	8000−7FFF
	park_q	Quadrature axis(q) component of the input signal in stationary reference frame	Q15	8000−7FFF
	theta_p	Phase angle between stationary and rotating frame	Q15	0−7FFF (0−360 degree)
Outputs	park_D	Direct axis(D) component of transformed signal in rotating reference frame	Q15	8000−7FFF
	park_Q	Quadrature axis(Q) component of transformed signal in rotating reference frame	Q15	8000−7FFF
Init / Config	none			

Variable Declaration:
In the system file include the following statements:

```
.ref   PARK, PARK_INIT                                ;function call

.ref   theta_p, park_d, park_q, park_D, park_Q    ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section 'park'

Example:

```
ldp  #park_d               ;Set DP for module input
bldd #input_var1, park_d   ;Pass input variable to module input
bldd #input_var2, park_q
bldd #input_var3, theta_p

CALL PARK

ldp  #output_var1          ;Set DP for output variable
bldd #park_D, output_var1  ;Pass module output to output variable
bldd #park_Q, output_var2
```

C/C-Callable ASM Interface

This function is implemented as a function with two arguments, each a pointer to the input and output structures.

```
struct   { int d;
           int q;
           int theta;
         } park_in;

struct   { int D;
           int Q;
         } park_out;

  void park(&park_in,&park_out);
```

The inputs are read from the park_in structure and the outputs are placed in the park_out structure.

Table 41. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	d	Direct axis(d) component of the input signal in stationary reference frame	Q15	8000−7FFF
	q	Quadrature axis(q) component of the input signal in stationary reference frame	Q15	8000−7FFF
	theta	Phase angle between stationary and rotating frame	Q15	0−7FFF (0−360 degree)
Outputs	D	Direct axis(D) component of transformed signal in rotating reference frame	Q15	8000−7FFF
	Q	Quadrature axis(Q) component of transformed signal in rotating reference frame	Q15	8000−7FFF
Init / Config	none			

Example:
In the following example, the variables stat_d, stat_q, are transformed to the rotating frame values based on theta_value.

```
typedef struct { int a,b,c ; } triad;

triad stationery_cmds;
triad rotating_cmds;

int some_other_var1, some_other_var2;
int stat_d,stat_q,theta_value;

void some_func(void)
{
  stationary_cmds.a=stat_d;
  stationary_cmds.b=stat_q;
  stationary_cmds.c=theta_value;

  park(&stationary_cmds,&rotating_cmds);
```

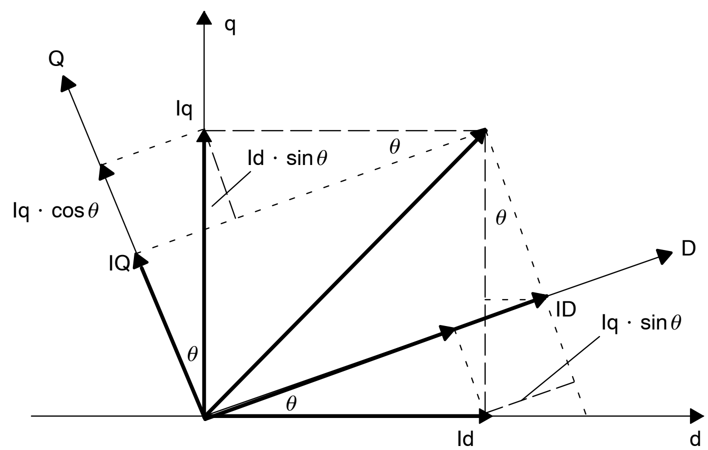
```
  some_other_var1=rotating_cmds.a;
  some_other_var2=rotating_cmds.b;
}
```

Background Information

Implements the following equations:

$$\begin{cases} ID = Id \times \cos\theta + Iq \times \sin\theta \\ IQ = -Id \times \sin\theta + Iq \times \cos\theta \end{cases}$$

This transformation converts vectors in 2-phase orthogonal stationary system into the rotating reference frame as shown in figure below:



The instantaneous input quantities are defined by the following equations:

$$\begin{cases} Id = I \times \sin(\omega t) \\ Iq = I \times \sin(\omega t + \pi/2) \end{cases}$$

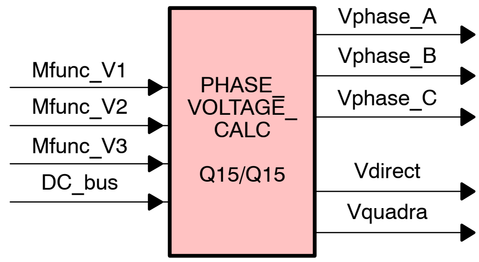
Table 42. Variable Cross Ref Table

Variables in the Equations	Variables in the Code
Id	park_d
Iq	park_q
θ	theta_p
ID	park_D
IQ	park_Q

PHASE_VOLTAGE_ CALC	*Three phase voltages and two stationary dq-axis voltages calculation based on DC-bus voltage and three upper switching functions*

Description

This software module calculates three phase voltages applied to the 3-ph motor (i.e., induction or synchronous motor) using the conventional voltage-source inverter. Three phase voltages can be reconstructed from the DC-bus voltage and three switching functions of the upper power switching devices of the inverter. In addition, this software module also includes the clarke transformation that converts three phase voltages into two stationary dq-axis voltages.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: volt_cal.asm

ASM Routines: PHASE_VOLTAGE_CALC, PHASE_VOLTAGE_CALC_INIT

C-callable ASM filenames: volt_cal.asm, volt_cal.h

Item	ASM Only	C-Callable ASM	Comments
Code size	68 words	76 words†	
Data RAM	12 words	0 words†	
xDAIS module	No	No	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre−initialized PHASEVOLTAGE structure instance consumes 10 words in the dat memory and 12 words in the .cinit section.

Direct ASM Interface

Table 43. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	Mfunc_V1	Switching function of upper switching device 1	Q15	–1 –> 0.999
	Mfunc_V2	Switching function of upper switching device 2	Q15	–1 –> 0.999
	Mfunc_V3	Switching function of upper switching device 3	Q15	–1 –> 0.999
	DC_Bus	DC-bus voltage	Q15	–1 –> 0.999
Outputs	Vphase_A	Line-neutral phase voltage A	Q15	–1 –> 0.999
	Vphase_B	Line-neutral phase voltage B	Q15	–1 –> 0.999
	Vphase_C	Line-neutral phase voltage C	Q15	–1 –> 0.999
	Vdirect	Stationary d-axis phase voltage	Q15	–1 –> 0.999
	Vquadra	Stationary q-axis phase voltage	Q15	–1 –> 0.999
Init / Config	out_of_phase_	Out-of-phase correction of three inputs of switching functions. It must be changed in the s/w module.	N/A	0 or 1

Routine names and calling limitation:
There are two routines involved:

PHASE_VOLTAGE_CALC, the main routine; and
PHASE_VOLTAGE_CALC_INIT, the initialization routine.

The initialization routine must be called during program initialization. The PHASE_VOLTAGE_CALC routine must be called in the control loop.

Variable Declaration:
In the system file, including the following statements before calling the subroutines:

```
.ref   PHASE_VOLTAGE_CALC                ; Function calls
.ref   PHASE_VOLTAGE_CALC_INIT           ; Function calls
.ref   Vphase_A, Vphase_B, Vphase_C      ; Outputs
.ref   Vdirect, Vquadra                  ; Outputs
.ref   Mfunc_V1, Mfunc_V2                ; Inputs
.ref   Mfunc_V3, DC_bus                  ; Inputs
```

Memory map:
All variables are mapped to an uninitialized named section, volt_cal, which can be allocated to any one data page.

Example:

In the interrupt service routine call the module and read results as follows:

```
LDP  #DC_bus                 ; Set DP for module inputs
BLDD #input_var1,Mfunc_V1    ; Pass input variables to module inputs
BLDD #input_var2,Mfunc_V2    ; Pass input variables to module inputs
BLDD #input_var3,Mfunc_V3    ; Pass input variables to module inputs
BLDD #input_var4,DC_bus      ; Pass input variables to module inputs

CALL PHASE_VOLTAGE_CALC

LDP  #output_var1              ; Set DP for module output
BLDD #Vphase_A,output_var1     ; Pass output to other variables
BLDD #Vphase_B,output_var2     ; Pass output to other variables
BLDD #Vphase_C,output_var3     ; Pass output to other variables
BLDD #Vdirect,output_var4      ; Pass output to other variables
BLDD #Vquadra,output_var5      ; Pass output to other variables
```

C/C-Callable ASM Interface

Object Definition

The structure of the PHASEVOLTAGE object is defined in the header file, volt_cal.h, as seen in the following:

```
typedef struct  { int  DC_bus;      /* Input: DC−bus voltage (Q15) */
                  int  Mfunc_V1;    /* Input: Modulation voltage phase A (Q15) */
                  int  Mfunc_V2;    /* Input: Modulation voltage phase B (Q15) */
                  int  Mfunc_V3;    /* Input: Modulation voltage phase C (Q15) */
                  int  Vphase_A;    /* Output: Phase voltage phase A (Q15) */
                  int  Vphase_B;    /* Output: Phase voltage phase B (Q15) */
                  int  Vphase_C;    /* Output: Phase voltage phase C (Q15) */
                  int  Vdirect;     /* Output: Stationary d−axis phase voltage (Q15) */
                  int  Vquadra;     /* Output: Stationary q−axis phase voltage (Q15) */
                  int  (*calc)();   /* Pointer to calculation function */
                } PHASEVOLTAGE;
```

Special Constants and Datatypes

PHASEVOLTAGE
The module definition itself is created as a data type. This makes it convenient to instance a PHASEVOLTAGE object. To create multiple instances of the module simply declare variables of type PHASEVOLTAGE.

PHASEVOLTAGE_DEFAULTS
Initializer for the PHASEVOLTAGE object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, volt_cal.h.

Methods

void calc(PHASEVOLTAGE *);
This default definition of the object implements just one method – the runtime compute function for reconstruction of three phase voltages including clarke transformation. This is implemented by means of a function pointer, and the default initializer sets this to phase_voltage_calc function. The argument to this function is the address of the PHASEVOLTAGE object. Again, this statement is written in the header file, volt_cal.h.

Module Usage

Instantiation:
The following example instances two such objects:

```
PHASEVOLTAGE volt1, volt2;
```

Initialization:
To instance a pre-initialized object:

```
PHASEVOLTAGE volt1 = PHASEVOLTAGE _DEFAULTS;
PHASEVOLTAGE volt2 = PHASEVOLTAGE _DEFAULTS;
```

Invoking the compute function

```
volt1.calc(&volt1);
volt2.calc(&volt2);
```

Example:

Lets instance two PHASEVOLTAGE objects, otherwise identical, and run two systems for phase voltage reconstruction. The following example is the c source code for the system file.

```
  /* instance the first object */
PHASEVOLTAGE volt1= PHASEVOLTAGE_DEFAULTS;
  /* instance the second object */
PHASEVOLTAGE volt2= PHASEVOLTAGE _DEFAULTS;

main()
{

    volt1.DC_bus=ilg2_vdc1.Vdc_meas;      /* Pass inputs to volt1 */
    volt1.Mfunc_V1=vhz1.svgen.va;         /* Pass inputs to volt1 */
    volt1.Mfunc_V2=vhz1.svgen.vb;         /* Pass inputs to volt1 */
    volt1.Mfunc_V3=vhz1.svgen.vc;         /* Pass inputs to volt1 */

    volt2.DC_bus=ilg2_vdc2.Vdc_meas;      /* Pass inputs to volt2 */
    volt2.Mfunc_V1=vhz2.svgen.va;         /* Pass inputs to volt2 */
    volt2.Mfunc_V2=vhz2.svgen.vb;         /* Pass inputs to volt2 */
    volt2.Mfunc_V3=vhz2.svgen.vc;         /* Pass inputs to volt2 */

}

void interrupt periodic_interrupt_isr()
{

    volt1.calc(&volt1);                   /* Call compute function for volt1 */
    volt2.calc(&volt2);                   /* Call compute function for volt2 */

    Va_1=volt1.Vphase_A;                  /* Access the outputs of volt1 */
    Vb_1=volt1.Vphase_B;
    Vc_1=volt1.Vphase_C;
    Vd_1=volt1.Vdirect;
    Vq_1=volt1.Vquadra;

    Va_2=volt2.Vphase_A;                  /* Access the outputs of volt2 */
    Vb_2=volt2.Vphase_B;
    Vc_2=volt2.Vphase_C;
    Vd_2=volt2.Vdirect;
    Vq_2=volt2.Vquadra;

}
```

Background Information

The phase voltage of a general 3-ph motor (V_{an}, V_{bn}, and V_{cn}) can be calculated from the DC-bus voltage (V_{dc}) and three upper switching functions of inverter (S_1, S_2, and S_3). The 3-ph windings of motor are connected either Δ or Y without a neutral return path (or 3-ph, 3-wire system). The overall system is shown in Figure 13.

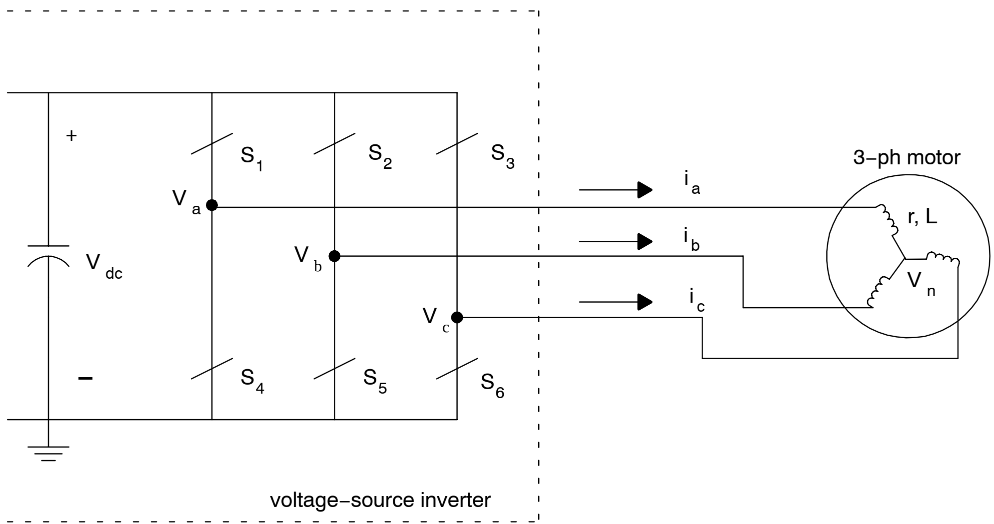


Figure 13. Voltage-Source Inverter With a 3-ph Electric Motor

Each phase of the motor is simply modeled as a series impedance of resistance and inductance (r, L) and back emf (e_a, e_b, e_c). Thus, three phase voltages can be computed as

$$V_{an} = V_a - V_n = i_a r + L\frac{di_a}{dt} + e_a \tag{1}$$

$$V_{bn} = V_b - V_n = i_b r + L\frac{di_b}{dt} + e_b \tag{2}$$

$$V_{cn} = V_c - V_n = i_c r + L\frac{di_c}{dt} + e_c \tag{3}$$

Summing these three phase voltages, yields

$$V_a + V_b + V_c - 3V_n = (i_a + i_b + i_c)r + L\frac{d(i_a + i_b + i_c)}{dt} + e_a + e_b + e_c \tag{4}$$

For a 3-phase system with no neutral path and balanced back emfs, $i_a + i_b + i_c = 0$, and $e_a + e_b + e_c = 0$. Therefore, equation (4) becomes,

$$V_{an} + V_{bn} + V_{cn} = 0 \tag{5}$$

Furthermore, the neutral voltage can be simply derived from (4)–(5) as

$$V_n = \frac{1}{3}(V_a + V_b + V_c) \tag{6}$$

Now three phase voltages can be calculated as

$$V_{an} = V_a - \frac{1}{3}(V_a + V_b + V_c) = \frac{2}{3}V_a - \frac{1}{3}V_b - \frac{1}{3}V_c \tag{7}$$

$$V_{bn} = V_b - \frac{1}{3}(V_a + V_b + V_c) = \frac{2}{3}V_b - \frac{1}{3}V_a - \frac{1}{3}V_c \tag{8}$$

$$V_{cn} = V_c - \frac{1}{3}(V_a + V_b + V_c) = \frac{2}{3}V_c - \frac{1}{3}V_a - \frac{1}{3}V_b \tag{9}$$

Three voltages V_a, V_b, V_c are related to the DC-bus voltage (V_{dc}) and three upper switching functions (S_1, S_2, S_3) as:

$$V_a = S_1 V_{dc} \tag{10}$$

$$V_b = S_2 V_{dc} \tag{11}$$

$$V_c = S_3 V_{dc} \tag{12}$$

where S_1, S_2, S_3 = either 0 or 1, and

$$S_4 = 1-S_1,\ S_5 = 1-S_2, \text{ and } S_6 = 1-S_3. \tag{13}$$

As a result, three phase voltages in (7)–(9) can also be expressed in terms of DC-bus voltage and three upper switching functions as:

$$V_{an} = V_{dc}\left(\frac{2}{3}S_1 - \frac{1}{3}S_2 - \frac{1}{3}S_3\right) \tag{14}$$

$$V_{bn} = V_{dc}\left(\frac{2}{3}S_2 - \frac{1}{3}S_1 - \frac{1}{3}S_3\right) \tag{15}$$

$$V_{cn} = V_{dc}\left(\frac{2}{3}S_3 - \frac{1}{3}S_1 - \frac{1}{3}S_2\right) \tag{16}$$

It is emphasized that the S_1, S_2, and S_3 are defined as the upper switching functions. If the lower switching functions are available instead, then the out-of-phase correction of switching functions is required in order to get the upper switching functions as easily computed from equation (13).

Next the clarke transformation is used to convert the three phase voltages (V_{an}, V_{bn}, and V_{cn}) to the stationary dq-axis phase voltages (V_{ds}^s, and V_{qs}^s). Because of the balanced system (5), V_{cn} is not used in clarke transformation.

$$V_{ds}^s = V_{an} \tag{17}$$

$$V_{qs}^s = \frac{1}{\sqrt{3}}(V_{an} + 2V_{bn}) \tag{18}$$

Figure 14 depicts the abc-axis and stationary dq-axis components for the stator voltages of motor.

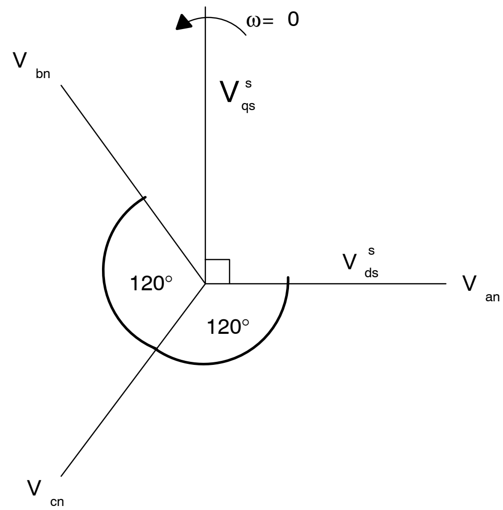


Figure 14. The abc-Axis and Stationary dq-Axis Components of the Stator Phase Voltages

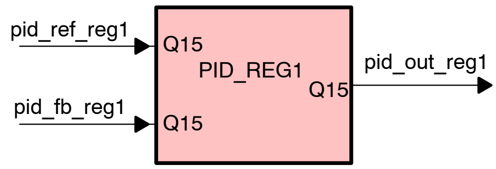
Table 44 shows the correspondence of notation between variables used here and variables used in the program (i.e., volt_cal.asm). The software module requires that both input and output variables are in per unit values (i.e., they are defined in Q15).

Table 44. Correspondence of Notations

	Equation Variables	Program Variables
Inputs	S_1	Mfunc_V1
	S_2	Mfunc_V2
	S_3	Mfunc_V3
	V_{dc}	DC_bus
Outputs	V_{an}	Vphase_A
	V_{bn}	Vphase_B
	V_{cn}	Vphase_C
	V_{ds}^{s}	Vdirect
	V_{qs}^{s}	Vquadra

Description

This module implements a digital PID controller without anti-windup correction. It can also be used as a PI or PD controller. In this implementation, the differential equation is transformed to a difference equation by means of the backward approximation.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: pid_reg1.asm

ASM Routines: PID_REG1, PID_REG1_INIT

C-callable ASM filenames: pid_reg1.asm, pid_reg1.h

Item	ASM Only	C-Callable ASM	Comments
Code size	94 words	99 words†	
Data RAM	21 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized PIDREG1 structure instance consumes 12 words in the data memory and 14 words in the .cinit section.

Direct ASM Interface

Table 45. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	pid_ref_reg1	Reference signal for PID regulator	Q15	–1 –> 0.999
	pid_fb_reg1	Feedback signal for PID regulator	Q15	–1 –> 0.999
Output	pid_out_reg1	PID regulator output	Q15	–1 –> 0.999
Init / Config	Kp_reg1†	Proportional gain coefficient	Q15	System dependent
	Ki_low_reg1†	Integral coefficient (low16 bit)	Q31 (L)	System dependent
	Ki_high_reg1†	Integral coefficient (high 16 bit)	Q31 (H)	System dependent
	Kd_reg1†	Derivative coefficient	Q15	System dependent
	pid_out_min†	Minimum PID regulator output	Q15	System dependent
	pid_out_max†	Maximum PID regulator output	Q15	System dependent

† From the system file initialize these PI regulator coefficients.

Routine names and calling limitation:
There are two routines involved:

PID_REG1, the main routine; and
PID_REG1_INIT, the initialization routine.

The initialization routine must be called during program initialization. The PID_REG1 routine must be called in the control loop.

Variable Declaration:
In the system file, including the following statements before calling the subroutines:

```
.ref   PID_REG1, PID_REG1_INIT          ;function call
.ref   pid_ref_reg1, pid_fb_reg1        ;Inputs
.ref   pid_out_reg1                     ;Output
```

Memory map:
All variables are mapped to an uninitialized named section, pid_reg1, which can be allocated to any one data page.

Example:
During system initialization specify the PID parameters as follows:

```
LDP  #Kp_reg1              ;Set DP for module parameters
SPLK #Kp_REG1_,Kp_reg1
SPLK #Ki_LO_REG1_,Ki_low_reg1
SPLK #Ki_HI_REG1_,Ki_high_reg1
SPLK #Kd_REG1_,Kd_reg1
SPLK #PID_OUT_MAX_,pid_out_max
SPLK #PID_OUT_MIN_,pid_out_min
```

Then in the interrupt service routine call the module and read results as follows:

```
LDP  # pid_fb_reg1                  ;Set DP for module inputs
BLDD #input_var1, pid_fb_reg1       ;Pass input variables to module inputs
BLDD #input_var2, pid_ref_reg1      ;Pass input variables to module inputs

CALL   PID_REG1

LDP  #output_var1                   ;Set DP for output variable
BLDD #pid_out_reg1, output_var1 ;   Pass module output to output variable
```

C/C-Callable ASM Interface

Object Definition

The structure of the PIDREG1object is defined in the header file, pid_reg1.h, as seen in the following:

```
typedef struct  { int  pid_ref_reg1;     /* Input: Reference input (Q15) */
                  int  pid_fb_reg1;      /* Input: Feedback input (Q15) */
                  int  Kp_reg1;          /* Parameter: Proportional gain (Q15) */
                  int  Ki_high_reg1;     /* Parameter: Integral gain (Q31) */
                  int  Ki_low_reg1;      /* Parameter: Integral gain (Q31) */
                  int  Kd_reg1;          /* Parameter: Derivative gain (Q15) */
                  int  pid_out_max;      /* Parameter: Maximum PID output (Q15) */
                  int  pid_out_min;      /* Parameter: Minimum PID output (Q15) */
                  int  pid_e1_reg1;      /* History: Previous error at time = k-1 (Q15) */
                  int  pid_e2_reg1;      /* History: Previous error at time = k-2 (Q15) */
                  int  pid_out_reg1;     /* Output: PID output (Q15) */
                  int  (*calc)();        /* Pointer to calculation function */
                } PIDREG1;
```

Special Constants and Datatypes

PIDREG1
The module definition itself is created as a data type. This makes it convenient to instance a PIDREG1 object. To create multiple instances of the module simply declare variables of type PIDREG1.

PIDREG1_DEFAULTS
Initializer for the PIDREG1 object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, pid_reg1.h.

Methods

void calc(PIDREG1 *);
This default definition of the object implements just one method – the runtime compute function for PID controller. This is implemented by means of a function pointer, and the default initializer sets this to pid_reg1_calc function. The argument to this function is the address of the PIDREG1 object. Again, this statement is written in the header file, pid_reg1.h.

Module Usage

Instantiation:
The following example instances two such objects:

```
PIDREG1  pid1, pid2;
```

Initialization:
To instance a pre-initialized object:

```
PIDREG1 pid1 = PIDREG1_DEFAULTS;
PIDREG1 pid2 = PIDREG1_DEFAULTS;
```

Invoking the compute function:

```
pid1.calc(&pid1);
pid2.calc(&pid2);
```

Example:

Lets instance two PIDREG1 objects, otherwise identical, and run two feedback systems. The following example is the c source code for the system file.

```
PIDREG1 pid1 = PIDREG1_DEFAULTS;    /* instance the first object */
PIDREG1 pid2 = PIDREG1_DEFAULTS;    /* instance the second object */

main()
{

    pid1.pid_ref_reg1=0x4000;           /* Pass inputs to pid1 */
    pid1.pid_fb_reg1=mras1.wr_hat_mras; /* Pass inputs to pid1 */
    pid2.pid_ref_reg1=0x7000;           /* Pass inputs to pid2 */
    pid2.pid_fb_reg1=mras2.wr_hat_mras; /* Pass inputs to pid2 */

}

void interrupt periodic_interrupt_isr()
{

    pid1.calc(&pid1);         /* Call compute function for pid1 */
    pid2.calc(&pid2);         /* Call compute function for pid2 */

    u1= pid1.pid_out_reg1;    /* Access the outputs of pid1 */
    u2= pid2.pid_out_reg1;    /* Access the outputs of pid2 */

}
```

Background Information

The block diagram of a conventional PID controller without anti-windup correction is shown in Figure 15.

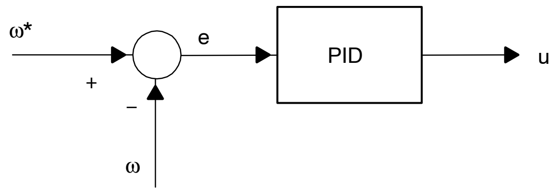


Figure 15. PID Controller Block Diagram

The differential equation for PID controller is described in the following equation.

$$u(t) = K_P e(t) + K_I \int_0^t e(\varsigma)d\varsigma + K_D \frac{de(t)}{dt} \tag{1}$$

where

$u(t)$ is the output of PID controller

$e(t)$ is the error between the reference and feedback variables (i.e., $e = \omega^* - \omega$)

ω^* is the reference variable

ω is the feedback variable

K_P is the proportional gain of PID controller

K_I is the integral gain of PID controller

K_D is the derivative gain of PID controller

Applying the Laplace transform to equation (1) with zero initial condition (i.e., e(0)=0), yields,

$$U(s) = \left[K_P + \frac{K_I}{s} + K_D s \right] E(s) \tag{2}$$

Using backward approximation, the differential equation can be transformed to the difference equation by substituting $s \Rightarrow \frac{1 - z^{-1}}{T}$ [1], where T is the sampling period (sec):

$$U(z) = \left[K_P + \frac{K_I T}{1 - z^{-1}} + \frac{K_D}{T}(1 - z^{-1}) \right] E(z) \tag{3}$$

Rearranging equation (3), yields

$$U(z)(1 - z^{-1}) = \left[K_P(1 - z^{-1}) + K_I T + \frac{K_D}{T}(1 - 2z^{-1} + z^{-2}) \right] E(z) \tag{4}$$

Equation (4) can be rewritten in discrete time-domain as,

$$u(k) - u(k-1) = K_P\big(e(k) - e(k-1)\big) + K_I T e(k) + \frac{K_D}{T}\big(e(k) - 2e(k-1) + e(k-2)\big) \tag{5}$$

Rearranging equation (5), we have,

$$u(k) = u(k-1) + \left(K_P + K_I T + \frac{K_D}{T} \right) e(k) - \left(K_P + 2\frac{K_D}{T} \right) e(k-1) + \frac{K_D}{T} e(k-2) \tag{6}$$

Denoting $K_0 = K_P + K_I T + \frac{K_D}{T}$, $K_1 = K_P + 2\frac{K_D}{T}$, and $K_2 = \frac{K_D}{T}$, the final equation is,

$$u(k) = u(k-1) + K_0 e(k) - K_1 e(k-1) + K_2 e(k-2) \tag{7}$$

where $K_0 > K_2$ and $K_1 > K_2$ are all typically positive numbers. Also, K_0, K_1, and K_2 can be independently set for different values of K_P, K_I, and K_D. In other words, any value of K_P, K_I, and K_D can be selected by setting K_0, K_1, and K_2 independently.

Equation (7) can be used to derive the PI or PD controller, as shown below:

PI Controller

According to equation (6), once K_D becomes zero, then $K_2 = 0$ and $K_0 > K_1$ where $K_0 = f(K_P,K_I)$ (i.e., a function of K_P and K_I) and $K_1 = f(K_P)$ (i.e., a function of K_P only)

PD Controller

According to equation (6), once K_I becomes zero, then $K_1 > K_0 > K_2$ and $K_1 = K_0 + K_2$ where $K_0 = f(K_P,K_D)$ (i.e., a function of K_P and K_D) and $K_2 = f(K_D)$ (i.e., a function of K_D only)

Notice that this PID controller is applicable for unsaturated output u(k) because it has no anti-windup correction (to get rid of the integral action when the output saturates).

In summary, Table 46 summarizes the setting K_P, K_I, and K_D for different types of controller and the corresponding output equation u(k). The corresponding K_0, K_1, and K_2 are also shown for different controllers, as shown in Table 46:

Table 46. Setting KP, KI, and KD and the Corresponding Output Equation u(k)

	Setting K_P, K_I, and K_D	Output equation u(k)	Comment
PI	$K_P \neq 0$, $K_I \neq 0$, $K_D = 0$	$u(k) = u(k-1) + K_0 e(k) - K_1 e(k-1)$	$K_2 = 0$, $K_0 > K_1$
PD	$K_P \neq 0$, $K_I = 0$, $K_D \neq 0$	$u(k) = u(k-1) + K_0 e(k) - K_1 e(k-1) + K_2 e(k-2)$ or $u(k) = K_0 e(k) - K_2 e(k-1)$	$K_1 = K_0 + K_2$ and $K_0 > K_2$
PID	$K_P \neq 0$, $K_I \neq 0$, $K_D \neq 0$	$u(k) = u(k-1) + K_0 e(k) - K_1 e(k-1) + K_2 e(k-2)$	$K_0 > K_2$ and $K_1 > K_2$

Table 47 shows the correspondence of notation between variables used here and variables used in the program (i.e., pid_reg1.asm). The software module requires that both input and output variables are in per unit values (i.e., they are defined in Q15).

Table 47. Correspondence of Notations

	Equation Variables	Program Variables
Inputs	$\omega^*(k)$	pid_ref_reg1
	$\omega(k)$	pid_fb_reg1
Output	$u(k)$	pid_out_reg1
Others	$u(k-1)$	pid_out1_reg1
	$e(k)$	pid_e0_reg1
	$e(k-1)$	pid_e1_reg1
	$e(k-2)$	pid_e2_reg1
	K_P	Kp_reg1
	$K_I T$	Ki_low_reg1, Ki_high_reg1
	K_D/T	Kd_reg1
	K_0	K0_low_reg1, K0_high_reg1
	K_1	K1_reg1
	K_2	Kd_reg1

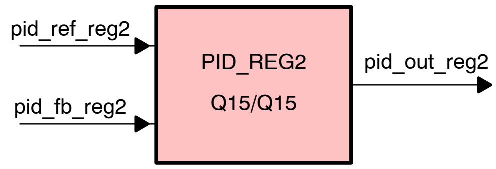
References:

1) G.F. Franklin, D.J. Powell, and M.L. Workman, *Digital Control of Dynamic Systems*, Addison-Wesley, 1997.

PID_REG2 — *Proportional and Integral Regulator 2*

Description

This module implements a PI regulator with integral windup correction



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: pid_reg2.asm

C-Callable Version File Name: pid_reg2.asm, pid2.h

Item	ASM Only	C-Callable ASM	Comments
Code size	50 words	74 words†	
Data RAM	12 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized PID2 structure instance consumes 13 words in the data memory and 15 words in the .cinit section.

Direct ASM Interface

Table 48. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	pid_ref_reg2	Reference signal for PI regulator.	Q15	8000−7FFF
	pid_fb_reg2	Feedback signal for PI regulator.	Q15	8000−7FFF
Output	pid_out_reg2	PI regulator output	Q15	pid_min_reg2 – pid_max_reg2
Init / Config	K0_reg2†	Proportional gain coefficient	Q9	System dependent
	K1_reg2†	Integral coefficient	Q13	System dependent
	Kc_reg2†	Integral windup correction coefficient	Q13	System dependent
	pid_min_reg2†	Minimum PI regulator output	Q15	System dependent
	pid_max_reg2†	Maximum PI regulator output	Q15	System dependent

† From the system file initialize these PI regulator coefficients.

Variable Declaration:
In the system file include the following statements:

```
.ref PID_REG2, PID_REG2_INIT        ;Function call
.ref pid_fb_reg2, pid_ref_reg2      ;Inputs
.ref pid_out_reg2,                  ;Output
.ref pid_max_reg2, pid_min_reg2     ;Parameters
.ref K0_reg2, K1_reg2, Kc_reg2      ;Parameters
```

Memory map:
All variables are mapped to an uninitialized named section ‘pid_reg2’

Example:

```
ldp  # pid_fb_reg2              ;Set DP for module inputs
bldd #input_var1, pid_fb_reg2   ;Pass input variables to module inputs
bldd #input_var2, pid_ref_reg2
CALL PID_REG2

ldp  #output_var1               ;Set DP for output variable
bldd #pid_out_reg2, output_var1 ;Pass module output to output variable
```

C/C-Callable ASM Interface

Object Definition

The structure of the PID2 Object is defined by the following structure definition

```
/*-----------------------------------------------------------------------------
Define the structure of the PID2
(pid regulator2)
-----------------------------------------------------------------------------*/

typedef struct {

     int fb_reg2;  /* Feedback signal for PI regulator Q15 Input  */
     int ref_reg2; /* Reference signal for PI regulator Q15 Input */
     int k0_reg2;  /* PI parameter - proportional gain Q9 */
     int k1_reg2;  /* PI parameter - integral time * sample time Q13 */
     int kc_reg2;  /* PI parameter - sampling time / integral time Q13 */
     int un_reg2;  /* Integral component of PI Q15 */
     int en0_reg2; /* reference signal - feedback signal Q15 */
     int upi_reg2; /* actual PI output without taking into account saturation Q15 */
                   /* i.e. if output is not saturated out_reg2 = upi_reg2 */
     int epi_reg2; /* out_reg2 - upi_reg2 Q15 */
     int max_reg2; /* PI parameter - upper cut off saturation limit of PI regulator output Q15*/
     int min_reg2; /* PI parameter - lower cut off saturation limit of PI regulator output Q15*/
     int out_reg2; /* final PI regulator output Q15 */
     int (*calc)();/* Pointer to the calculation function */
} PID2;
```

Table 49. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	ref_reg2	Reference signal for PI regulator.	Q15	8000−7FFFh
	fb_reg2	Feedback signal for PI regulator.	Q15	8000−7FFFh
Output	out_reg2	PI regulator output	Q15	min_reg2 – max_reg2

Special Constants and Datatypes

PID2

The module definition itself is created as a data type. This makes it convenient to instance a pid regulator 2 module.To create multiple instances of the module simply declare variables of type PID2

PID2_handle

Typedef’ed to PID2 *

PID2_DEFAULTS
Initializer for the PID2 Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(PID2_handle)
The default definition of the object implements just one method – the runtime implementation of the pid regulator 2. This is implemented by means of a function pointer, and the default initializer sets this to pid2_calc. The argument to this function is the address of the PID2 object.

Module Usage

Instantiation:
The following example instances one such objects:

```
PID2 p1,p2
```

Initialization:
To instance a pre-initialized object

```
PID2  p1 = PID2_DEFAULTS, p2 = PID2_DEFAULTS;
```

Invoking the compute function:

```
p1.calc(&p1);
```

Example:

Lets instance two PID2 objects,otherwise identical ,but running with different freq values.

```
PID2  p1 = PID2_DEFAULTS; /* Instance the first object */
PID2  p2 = PID2_DEFAULTS; /* Instance the second object */

main()
{
    p1.k0_reg2 = 5;
    p1.k1_reg2 = 6;
    p1.kc_reg2 = 7;
    p1.min_reg2 = 10;
    p1.max_reg2 = 20;
    p1.un_reg2 = 20;

    p2.k0_reg2 = 17;
    p2.k1_reg2 = 13;
    p2.kc_reg2 = 14;
    p2.min_reg2 = 20;
    p2.max_reg2 = 40;
    p2.un_reg2 = 20;
}
void interrupt periodic_interrupt_isr()
{
     (*p1.calc)(&p1);    /* Call compute function for p1 */
     (*p2.calc)(&p2);    /* Call compute function for p2 */

     x = p1.out_reg2;    /* Access the output */

     q = p2.out_reg2;    /* Access the output */

  /* Do something with the outputs */

}
```

Background Information

An analog PI controller can be transformed to an equivalent digital form as shown below, before being implemented by 24x/24xx:

$$G(s) = K_P \times \frac{1+T_i s}{T_i s} = K_P + \frac{K_I}{s} = \frac{U(s)}{E(s)}$$

Where,

$$K_I = \frac{K_P}{T_i}$$

K_P and T_i are the gain and integral time respectively.

In discrete form the controller above can be expressed as,

$$U(n) = K_P E(n) + K_I T_S \sum_{i=0}^{n} E(i)$$

where T_S is the sampling time. This is implemented with output saturation and integral component correction using the following three equations:

$$U(n) = K0 * E(n) + I(n)$$
$$I(n) = I(n-1) + K1 * E(n) + Kc * Epi$$
$$Epi = Us - U(n)$$

where,

$$U(n) \geq U_{max} \Rightarrow Us = U_{max}$$
$$U(n) \leq U_{min} \Rightarrow Us = U_{min}$$

otherwise,

$$Us = U(n)$$

The coefficients are defined as,

$$K0 = K_P,$$

$$K1 = K_I T_S = \frac{K_P T_S}{T_i},$$

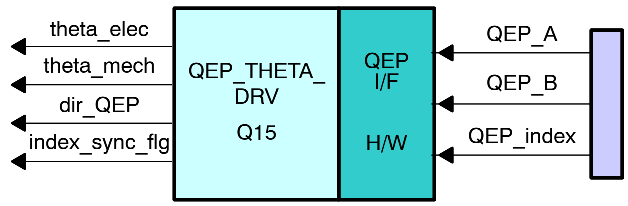
$$Kc = \frac{K1}{K0} = \frac{T_S}{T_i}$$

Table 50. Variable Cross Ref Table

Variables in the Equations	Variables in the Code
U(n)	pid_out_reg2
I(n)	Un_reg2
E(n)	En0_reg2
Epi	epi_reg2
U_{max}	pid_max_reg2
U_{min}	pid_min_reg2
K0	K0_reg2
K1	K1_reg2
Kc	Kc_reg2

Description

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



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM File Name: qep_drv.asm

C–Callable Version File Names: F243QEP1.C, F243QEP2.ASM, F243QEP.H, F2407QEP1.C, F2407QEP2.ASM, F2407QEP.HQEP.H

Item	ASM Only	C-Callable ASM	Comments
Code size	53 words	108 words†	
Data RAM	9 words	0 words†	
Multiple instances	No	See note	

† Each pre-initialized QEP structure instance consumes 13 words in the data memory and 15 words in the .cinit section.

Note: Multiple instances must point to distinct interfaces on the target device. Multiple instances pointing to the same QEP interface in hardware may produce undefined results. So the number of interfaces on the F241/3 is limited to one, while there can be upto two such interfaces on the LF2407.

Direct ASM Interface

Table 51. Module Terminal Variables/Functions

	Name	Description	Format	Range
H/W Inputs	QEP_A	Quadrature pulse A input to 24x/24xx from the position encoder	N/A	N/A
	QEP_B	Quadrature pulse B input to 24x/24xx from the position encoder	N/A	N/A
	QEP_index	Zero index pulse input to 24x/24xx from the position encoder	N/A	N/A
Outputs	theta_elec	Per unit (pu) electrical displacement of the rotor.	Q15	0–7FFF (0–360 degree)
	theta_mech	Per unit (pu) mechanical displacement of the rotor	Q15	0–7FFF (0–360 degree)
	dir_QEP	Rotor direction of rotation signal	Q0	0 or F
	index_sync_flg	Flag variable for synchronizing rotor displacement calculation with zero index pulse.	Q0	0 or F
	QEP_cnt_idx	T2CNT value prior to resetting it at the occurrence of the index pulse.	Q0	N/A
Init / Config	24x/24xx†	Select appropriate 24x/24xx device in the x24x_app.h file.		
	polepairs†	Number of pole pairs in the motor	Q0	N/A
	cal_angle†	Timer 2 counter (T2CNT) value when the rotor mechanical displacement is 0.	Q0	N/A
	mech_scale†	Scaling factor for converting T2CNT values to per unit mechanical displacement.	Q26	N/A

† From the system file, initialize these parameters with the desired values if the default values are not used. These are initialized to some default values in the init routine (QEP_THETA_DRV_INIT).

Variable Declaration:
In the system file include the following statements:

```
.ref   QEP_THETA_DRV, QEP_THETA_DRV _INIT        ;function call
.ref   QEP_INDEX_ISR_DRV                         ;ISR call
.ref   polepairs, cal_angle, mech_scale          ;inputs
.ref   theta_elec, theta_mech, dir_QEP           ;outputs
.ref   index_sync_flg, QEP_cnt_idx               ;outputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘qep_drv’

Example:

```
CALL   QEP_THETA_DRV

ldp    #output_var1              ;Set DP for output variable
bldd   #theta_elec, output_var1  ;Pass module outputs to output variables
bldd   #theta_mech, output_var2
bldd   #dir_QEP, output_var3
```

Note:

This module does not need any input parameter passing in the interrupt routine. It receives it's inputs from the hardware(H/W) internal to 24x/24xx. The signals from the shaft position encoder are first applied to the appropriate QEP pins of 24x/24xx device. Then the QEP interface(QEP I/F) H/W inside 24x/24xx generates three intermediate signals which are finally used as inputs to this module.

C/C-Callable ASM Interface

Object Definition

The structure of the EVMDAC object is defined by the following structure definition

```
/*-----------------------------------------------------------------------
Define the structure of the QEP (Quadrature Encoder) Driver Object
-----------------------------------------------------------------------*/
typedef struct {
       int theta_elec;     /* Motor Electrical Angle, Q15, Output   */
       int theta_mech;     /* Motor Mechanical Angle  Q15, Output   */
       int QepDir;         /* Motor rotation direction Q0, Output   */
       int dwn_cnt_offset; /* Encoder offset 65533- #lines Q0,Input */
       int theta_raw;      /* Raw angle  Q0, Internal, Output       */
       int mech_scaler;    /* Scaler for conv'n to Q15 Q15,Parameter*/
       int pole_pairs;     /* # of poles/2 for the motor, Q0 Input  */
       int rev_counter;    /* # of index events, Q0, Output+History */
       int pulse_count;    /* Pulses on encoder at index- Output-Q0 */
       int index_flag ;    /* Index sync status Q0 output+History   */
       int (*calc)();      /* Pointer to the calc funtion           */
       int (*init)();      /* Pointer to the init funcion           */
       int (*indexevent)(); /* Pointer to index event handler       */
       }  QEP ;
```

Table 52. Module Terminal Variables/Functions

	Name	Description	Format	Range
H/W Inputs	QEP_A	Quadrature pulse A input to 24x/24xx from the position encoder	N/A	N/A
	QEP_B	Quadrature pulse B input to 24x/24xx from the position encoder	N/A	N/A
	QEP_index	Zero index pulse input to 24x/24xx from the position encoder	N/A	N/A
Outputs	theta_elec	Per unit (pu) electrical displacement of the rotor.	Q15	0−7FFF (0−360 degree)
	theta_mech	Per unit (pu) mechanical displacement of the rotor	Q15	0−7FFF (0−360 degree)
	QEP_dir	Rotor direction of rotation signal	Q0	0 or F
	index_flg	Flag variable for synchronizing rotor displacement calculation with zero index pulse.	Q0	0 or F
	QEP_cnt_idx	T2CNT value prior to resetting it at the occurrence of the index pulse.	Q0	N/A
Init / Config	24x/24xx†	Select appropriate 24x/24xx device in the x24x_app.h file.		
	pole_pairs†	Number of pole pairs in the motor	Q0	N/A
	cal_angle†	Timer 2 counter (T2CNT) value when the rotor mechanical displacement is 0.	Q0	N/A

	Name	Description	Format	Range
	mech_scale†	Scaling factor for converting T2CNT values to per unit mechanical displacement.	Q26	N/A
	rev_counter	Number of index events handled.	Q0	−32768 to 32767

† From the system file, initialize these parameters with the desired values if the default values are not used. These are initialized to some default values in the init routine (QEP_THETA_DRV_INIT).

Special Constants and Datatypes

QEP
Module definition data type.

QEP_DEFAULTS
Initializer for the QEP Object. This provides the initial values to the variables as well as method pointers.

Module Usage

Instantiation:
The interface to the QEP is instanced thus:

```
QEP qep1;
```

Initialization:
To instance a pre−initialized interface:

```
QEP qep1=QEP_DEFAULTS;
```

To initialize the QEP measurement hardware (timer/counter etc) call the init function:

```
qep1.init(&qep1);
```

Invoking the angle calculation function:

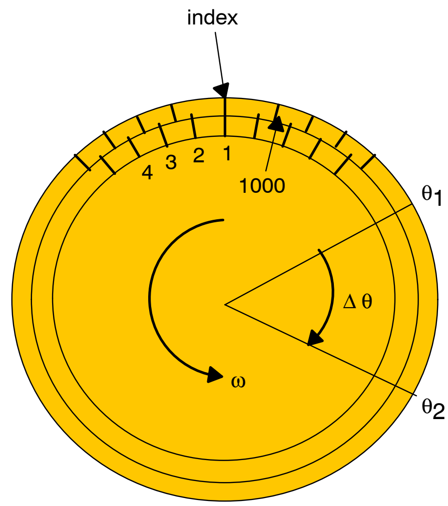
```
qep.calc(&qep1);
```

Invoking the index event handler:
The index event handler resets the QEP counter, and synchronizes the software / hardware counters to the index pulse. Also it sets the QEP.index_flag 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.

```
void interrupt_linked_to_the_index()
{
  qep1.index_event(&qep1);
}
```

Background Information



Example:
1000 QEP pulses = 4000 counter "ticks," per 360°

Figure 16. Speed Sensor Disk

Figure 16 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 17 as QEP_A, QEP_B and QEP_index.

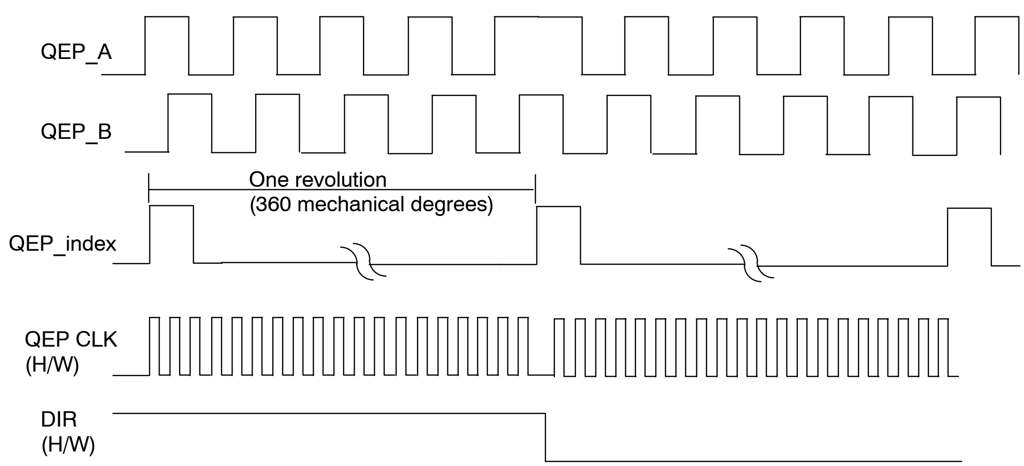


Figure 17. Quadrature Encoder Pulses, Decoded Timer Clock and Direction Signal

These signals are applied to 24x/24xx CAP/QEP interface circuit to determine the motor speed, position and direction of rotation. QEP_A and QEP_B signals are applied to the QEP1 and QEP2 pins of 24x/24xx device respectively. QEP_index signal is applied to the CAP3 pin. The QEP interface circuit in 24x/24xx, when enabled (CAPCONx[13,14]), count these QEP pulses and generates two signals internal to the device. These two signals are shown in Figure 17 as QEP_CLK and DIR. QEP_CLK signal is used as the clock input to GP Timer2. DIR signal controls the GP Timer2 counting direction.

Now the number of pulses generated by the speed sensor is proportional to the angular displacement of the motor shaft. In Figure 16, a complete 360° rotation of motor shaft generates 1000 pulses of each of the signals QEP_A and QEP_B. The QEP circuit in 24x/24xx 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.

The counting direction of GP Timer2 is reflected by the status bit, BIT14, in GPTCON register. Therefore, in the s/w, BIT14 of GPTCON is checked to determine the direction of rotation of the motor.

The capture module (CAP3) is configured to generate an interrupt on every rising edge of the QEP_index signal. In the corresponding CAP3 interrupt routine the function QEP_INDEX_ISR_DRV is called. This function resets the timer counter T2CNT and sets the index synchronization flag *index_sync_flg* to 000F. 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 *theta_raw.* This value indicates the clock pulse count at that instant of time. Therefore, *theta_raw* is a measure of the rotor mechanical displacement in terms of the number of clock pulses. From this value of *theta_raw*, the corresponding per unit mechanical displacement of the rotor, *theta_mech*, is calculated as follows:

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

$$
\begin{aligned}
& mech\_scale \times 4000 = 360^{\circ}\ mechanical = 1\ per\ unit(pu)\ mechanical\ displacement \\
& \Rightarrow mech\_scale = (1/4000)\ pu\ mech\ displacement\ /\ count \\
& = 16777\ pu\ mech\ displacement\ /\ count\ (in\ Q26)
\end{aligned}
$$

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

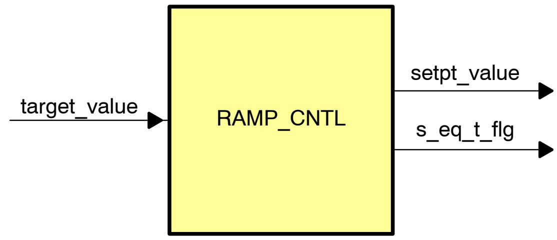
$$theta\_mech = mech\_scale \times theta\_raw$$

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

$$theta\_elec = polepairs \times theta\_mech$$

Description

This module implements a ramp up and ramp down function. The output flag variable s_eq_t_flg is set to 7FFFh when the output variable setpt_value equals the input variable target_value.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: rmp_cntl.asm

C-Callable ASM File Names: rmp_cntl.asm, rmp_cntl.h

Item	ASM Only	C-Callable ASM	Comments
Code size	47 words	72 words†	
Data RAM	7 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized RMPCNTL structure instance consumes 8 words in the data memory and 10 words in the .cinit section.

Direct ASM Interface

Table 53. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	target_value	Desired value of the ramp	Q0	rmp_lo_limit – rmp_hi_limit
Outputs	setpt_value	Ramp output value	Q0	rmp_lo_limit – rmp_hi_limit
	s_eq_t_flg	Ramp output status flag	Q0	0 or 7FFF
Init / Config	rmp_dly_max†	Ramp step delay in number of sampling cycles	Q0	0−7FFF
	rmp_hi_limit†	Maximum value of ramp	Q0	0−7FFF
	rmp_lo_limit†	Minimum value of ramp	Q0	0−7FFF

† From the system file, initialize these variables as required by the application. From the Real-Time Code Composer window, specify *target_value* to vary the output signal *setpt_value*.

Variable Declaration:
In the system file include the following statements:

```
.ref   RAMP_CNTL, RAMP_CNTL_INIT      ; function call
.ref   target_value                   ; Inputs
.ref   rmp_dly_max, rmp_lo_limit      ; Input Parameters
.ref   rmp_hi_limit                   ; Input Parameter
.ref   setpt_value, s_eq_t_flg        ; Outputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘rmp_cntl’

Example:

```
ldp  #target_value               ;Set DP for module input
bldd #input_var1, target_value ;Pass input variable to module input

CALL RAMP_CNTL

ldp    #output_var1            ;Set DP for output variable
bldd #setpt_value, output_var1 ;Pass module output to output variable
```

C/C-Callable ASM Interface

Object Definition

The structure of the RMPCNTL object is defined in the header file, rmp_cntl.h, as seen in the following:

```
typedef struct  { int  target_value;    /* Input: Target input (Q15) */
                  int  rmp_dly_max;     /* Parameter: Maximum delay rate */
                  int  rmp_lo_limit;    /* Parameter: Minimum limit (Q15) */
                  int  rmp_hi_limit;    /* Parameter: Maximum limit (Q15) */
                  int  rmp_delay_cntl;  /* Variable: Incremental delay  */
                  int  setpt_value;     /* Output: Target output (Q15) */
                  int  s_eq_t_flg;      /* Output: Flag output */
                  int  (*calc)();       /* Pointer to calculation function */
                } RMPCNTL;
```

Special Constants and Datatypes

RMPCNTL
The module definition itself is created as a data type. This makes it convenient to instance a RMPCNTL object. To create multiple instances of the module simply declare variables of type RMPCNTL.

RMPCNTL_DEFAULTS
Initializer for the RMPCNTL object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, rmp_cntl.h.

Methods

void calc(RMPCNTL *);
This default definition of the object implements just one method – the runtime compute function for ramp control. This is implemented by means of a function pointer, and the default initializer sets this to rmp_cntl_calc function. The argument to these functions is the address of the RMPCNTL object. Again, this statement is written in the header file, rmp_cntl.h.

Module Usage

Instantiation:
The following example instances two such objects

```
RMPCNTL  rmpc1, rmpc2;
```

Initialization:
To instance a pre-initialized object:

```
RMPCNTL rmpc1 = RMPCNTL_DEFAULTS;
RMPCNTL rmpc2 = RMPCNTL_DEFAULTS;
```

Invoking the compute function:

```
rmpc1.calc(&rmpc1);
rmpc2.calc(&rmpc2);
```

Example:

Lets instance two RMPCNTL objects, otherwise identical, and run two ramp controlling variables. The following example is the c source code for the system file.

```
RMPCNTL rmpc1= RMPCNTL_DEFAULTS;    /* instance the first object */
RMPCNTL rmpc2 = RMPCNTL_DEFAULTS;   /* instance the second object */

main()
{

     rmpc1.target_value = input1;        /* Pass inputs to rmpc1 */
     rmpc2.target_value = input2;        /* Pass inputs to rmpc2 */

}

void interrupt periodic_interrupt_isr()
{

     rmpc1.calc(&rmpc1);        /* Call compute function for rmpc1 */
     rmpc2.calc(&rmpc2);        /* Call compute function for rmpc2 */

output1 = rmpc1.setpt_value;   /* Access the outputs of rmpc1 */
output2 = rmpc2.setpt_value;   /* Access the outputs of rmpc2 */

}
```

Background Information

Implements the following equations:

Case 1: When *target_value* > *setpt_value*

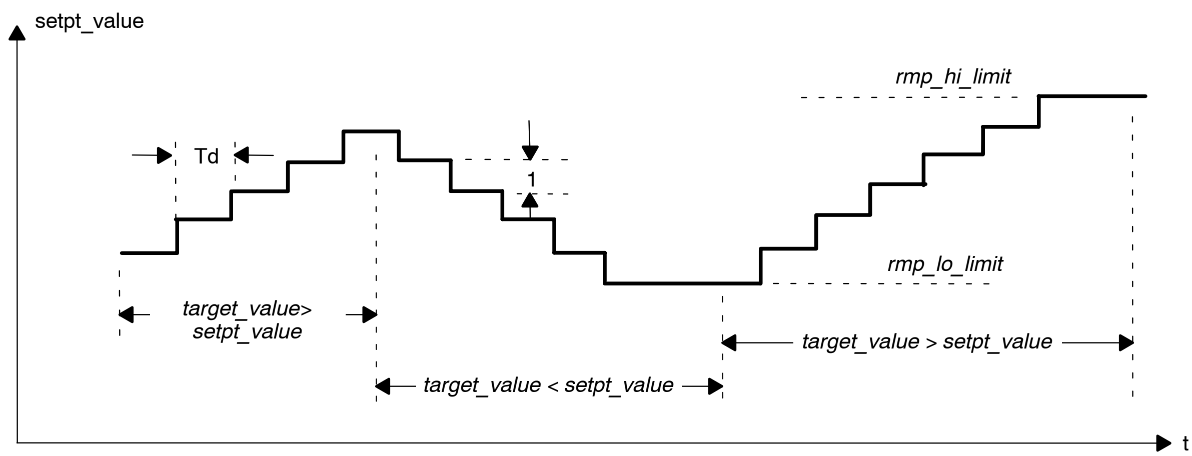
$$setpt\_value = \begin{cases} setpt\_value + 1, & \text{for } t = n \cdot Td,\ n = 1, 2, 3\ldots \text{ and } (setpt\_value + 1) < rmp\_hi\_limit \\ rmp\_hi\_limit, & \text{for } (setpt\_value + 1) > rmp\_hi\_limit \end{cases}$$

where,
Td = *rmp_dly_max* . Ts
Ts = Sampling time period

Case 2: When *target_value* < *setpt_value*

$$setpt\_value = \begin{cases} setpt\_value - 1, & \text{for } t = n \cdot Td,\ n = 1, 2, 3\ldots.. \text{ and } (setpt\_value - 1) > rmp\_lo\_limit \\ rmp\_lo\_limit, & \text{for } (setpt\_value - 1) < rmp\_lo\_limit \end{cases}$$

where,
Td = *rmp_dly_max* . Ts
Ts = Sampling time period



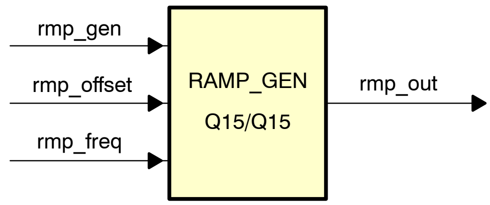
Example:

setpt_value = 0 (initial value), target_value = 1000 (user specified), rmp_dly_max = 500 (user specified), sampling loop time period Ts = 0.000025 Sec.

This means that the time delay for each ramp step is Td = 500x0.000025 = 0.0125 Sec. Therefore, the total ramp time will be Tramp = 1000x0.0125 Sec = 12.5 Sec

Description

This module generates ramp output of adjustable gain, frequency and dc offset.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: rampgen.asm

C-Callable Version File Names: rampgen.asm, rampgen.h

Item	ASM Only	C-Callable ASM	Comments
Code size	28 words	27 words text + cinit mem†	
Data RAM	8 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized RAMPGEN structure consumes 7 words in the data memory and 9 words in the .cinit section.

Direct ASM Interface

Table 54. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	rmp_gain	Normalized slope of the ramp signal.	Q15	0−7FFF
	rmp_offset	Normalized DC offset in the ramp signal.	Q15	0−7FFF
	rmp_freq	Normalized frequency of the ramp signal.	Q15	0−7FFF
Outputs	rmp_out	Normalized Ramp output	Q15	0−7FFF
Init / Config	step_angle_max	Initialize the maximum ramp frequency by specifying this maximum step value. The default value is set to 1000 to generate a maximum frequency of 305.2Hz using a 20kHz sampling loop.	Q0	User specified

Variable Declaration:
In the system file include the following statements:

```
.ref   RAMP_GEN, RAMP_GEN_INIT          ;function call
.ref   rmp_gain, rmp_offset, rmp_freq   ;inputs
.ref   step_angle_max                   ;input
.ref   rmp_out                          ;output
```

Memory map:
All variables are mapped to an uninitialized named section ‘rampgen’

Example:

```
ldp  #rmp_gain               ;Set DP for module inputs
bldd #input_var1, rmp_gain   ;Pass input variables to module inputs
bldd #input_var2, rmp_freq
bldd #input_var3, rmp_offset

CALL RAMP_GEN

ldp  #output_var1            ;Set DP for output variable
bldd #rmp_out, output_var1   ;Pass module output to output variable
```

C/C-Callable ASM Interface

Object Definition

The structure of the RAMPGEN object is defined by the following structure definition

```
/*-------------------------------------------------------------------
Define the structure of the RAMPGEN Ramp Function Generator
-------------------------------------------------------------------*/
typedef struct {
                int freq;     /* Frequency setting Q15 Input      */
                int freq_max; /* Frequency setting Q0  Input      */
                int alpha;    /* Internal var history             */
                int gain;     /* Waveform amplitude Q15 Input     */
                int offset;   /* Offset setting Q15 Input         */
                int out;      /* Ramp outputQ15 Output            */
                int (*calc)(); /* Pointer to calculation function */
               } RAMPGEN;
```

Table 55. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	gain	Normalized slope of the ramp signal.	Q15	0−7FFF
	offset	Normalized DC offset in the ramp signal.	Q15	0−7FFF
	freq	Normalized frequency of the ramp signal.	Q15	0−7FFF
Outputs	out	Normalized Ramp output	Q15	0−7FFF
Init / Config	freq_max	Initialize the maximum ramp frequency by specifying this maximum step value. The default value is set to 1000 to generate a maximum frequency of 305.2Hz using a 20kHz sampling loop.	Q0	User specified

Special Constants and Datatypes

RAMPGEN
Data type for instancing Rampgen module(s).

RAMPGEN_handle
Typedefed to RAMPGEN *.

RAMPGEN_DEFAULTS
Default values for RAMPGEN objects.

Methods

void calc(RAMPGEN_handle);

Invoke this function to compute the next point on the RAMP. The RAMPGEN properties must be initialized properly before calling the compute function. Also it is VERY important that the method pointer in the RAMPGEN object be initialized to a valid RAMPGEN compute function, to avoid execution into garbage and system crashes.

Module Usage

Instantiation:

The following example instances two such objects:

```
RAMPGEN rmp1,rmp2;
```

Initialization:

The above creates 'empty' object. To create pre-initialized objects, the following form can be used:

```
RAMPGEN rmp1= RAMPGEN_DEFAULTS;
RAMPGEN rmp2= RAMPGEN_DEFAULTS;
```

Invoking the compute function:

```
rmp1.calc(&rmp1);
```

Computes the next point of the ramp.

Example:

```
    RAMPGEN ramp1=RAMP_DEFAULTS;

main()
{
    ramp1.freq=0x2000;
}

void periodic_interrupt_isr()
{
    int output;
    ramp1.calc(&ramp1); /* Call the ramp calculation function */
    output=ramp1.out;   /* Access output of ramp              */

      /* Do something with the output  */

. . .
. . .
. . .
. . .

}
```

Background Information

In this implementation the frequency of the ramp output is controlled by a precision frequency generation algorithm which relies on the modulo nature (i.e. wrap-around) of finite length variables in 24x/24xx. One such variable, called *alpha_rg* (a data memory location in 24x/24xx) in this implementation, is used as a modulo-16 variable to control the time period (1/frequency) of the ramp signal. Adding a fixed step value (*step_angle_rg*) to this variable causes the value in *alpha_rg* to cycle at a constant rate through 0 to FFFFh. At the end limit the value in *alpha_rg* simply wraps around and continues at the next modulo value given by the step size. The rate of cycling through 0 to FFFFh is very easily and accurately controlled by the value of the step size.

For a given step size, the frequency of the ramp output (in Hz) is given by:

$$f = \frac{step\_angle\_rg \times f_s}{2^m}$$

where

f_s = sampling loop frequency in Hz

m = # bits in the auto wrapper variable alpha_rg.

From the above equation it is clear that a step_angle_rg value of 1 gives a frequency of 0.3052Hz when m=16 and f_S=20kHz. This defines the frequency setting resolution of the ramp output.

Now if the maximum step size is *step_angle_max* and the corresponding maximum frequency is f_{max}, then from the above equation we have,

$$f_{\max} = \frac{step\_angle\_max \times f_s}{2^m}$$

From the last two equations we have,

$$\frac{f}{f_{\max}} = \frac{step\_angle\_rg}{step\_angle\_max}$$
$$\Rightarrow step\_angle\_rg = rmp\_freq \times step\_angle\_max$$

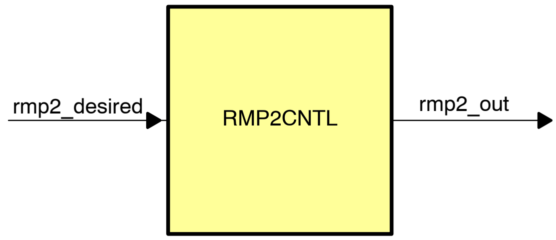
This last equation is implemented in the code to control the frequency of the ramp output. Here, the normalized ramp output frequency, rmp_freq, is given by,

$$rmp\_freq = \frac{f}{f_{\max}}$$

In the code the variable *step_angle_max* is initialized to 1000. This means the maximum ramp frequency is f_{max}=305.17 Hz, when m=16 and f_s=20kHz.

Description

This module implements a ramp up and ramp down function. The output variable *rmp2_out* follows the desired ramp value *rmp2_desired*.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: rmp2cntl.asm

C-Callable Version File Name: rmp2cntl.asm, rmp2.h

Item	ASM Only	C-Callable ASM	Comments
Code size	48 words	53 words†	
Data RAM	6 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized RMP2 structure instance consumes 7 words in the data memory and 9 words in the .cinit section.

Direct ASM Interface

Table 56. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	rmp2_desired	Desired output value of ramp 2	Q0	0–7FFF
Output	rmp2_out	Ramp 2 output	Q0	rmp2_min – rmp2_max
Init / Config	rmp2_dly†	Ramp 2 step delay in number of sampling cycles	Q0	0–7FFF
	rmp2_max†	Maximum value of ramp 2	Q0	0–7FFF
	rmp2_min†	Minimum value of ramp 2	Q0	0–7FFF

† From the system file, initialize these variables as required by the application. From the Real-Time Code Composer watch window, specify *rmp2_desired* to vary the output *signal rmp2_out*.

Variable Declaration:
In the system file include the following statements:

```
.ref   RMP2CNTL, RMP2CNTL_INIT           ;function call
.ref    rmp2_dly, rmp2_desired           ;input
.ref    rmp2_max, rmp2_min, rmp2_out     ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section ‘rmp2cntl’

Example:

```
ldp  #rmp2_desired              ;Set DP for module input
bldd #input_var1, rmp2_desired  ;Pass input variable to module input

CALL RMP2CNTL

ldp  #output_var1             ;Set DP for output variable
bldd #rmp2_out, output_var1   ;Pass module output to output variable
```

C/C-Callable ASM Interface

Object Definition

The structure of the RMP2 Object is defined by the following structure definition

```
/*-----------------------------------------------------------------------------
Define the structure of the RMP2
(Ramp2 control module)
-----------------------------------------------------------------------------*/

typedef struct {
int max   /*  Maximum  value of Ramp2  */
int min;   /*  Minimum  value of Ramp2                         */
int dly;   /* Ramp 2 step delay in number of sampling cycles       */
int delay_cntr; /* Counter for ramp 2 step delay  */
int desired;     /*  Desired value of ramp2    */
int out;         /*  Ramp2 output   */
int (*calc)();   /*  Pointer to the calculation function  */
} RMP2;
```

Table 57. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	delay_cntr	Counter for ramp 2 step delay	Q0	0–7FFF
	dly	Ramp 2 step delay in number of sampling cycles	Q0	0–7FFF
	desired	Desired value of ramp 2	Q0	0–7FFF
	max	Maximum value of ramp 2	Q0	0–7FFF
	min	Minimum value of ramp 2	Q0	0–7FFF
Output	out	Ramp 2 output	Q0	min–max

Special Constants and Datatypes

RMP2
The module definition itself is created as a data type. This makes it convenient to instance a ramp2 control module.To create multiple instances of the module simply declare variables of type RMP2

RMP2_handle
Typedef'ed to RMP2 *

RMP2_DEFAULTS;
Initializer for the RMP2 Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc (RMP2_handle)
The default definition of the object implements just one method – the runtime computation of the ramp2 control function. This is implemented by means of a function pointer, and the default initializer sets this to rmp2_calc. The argument to this function is the address of the RMP2 object.

Module Usage

Instantiation:
The following example instances two such objects:

```
RMP2 p1,p2
```

Initialization:
To instance a pre-initialized object

```
RMP2 p1=RMP2_DEFAULTS, p2=RMP2_DEFAULTS;
```

Invoking the compute function:

```
p1.calc(&p1);
```

Example:
Lets instance one RMP2 object

```
RMP2  p1 = RMP2_DEFAULTS;    /* Instance the first object*/
RMP2  p2 = RMP2_DEFAULTS;    /* Instance the second object*/

main()
{
    p1.desired = 8;   /* initialize */
    p1.min=50;
    p1.out = 30;
    p1.dly = 1;

    p2.desired = 6;   /* initialize */
    p2.min=60;
    p2.out = 40;
    p2.dly = 2;
}

void interrupt periodic_interrupt_isr()
{

    (*p1.calc)(&p1);    /* Call compute function for p1 */

    (*p2.calc)(&p2);    /* Call compute function for p2 */

    x = p1.out; /* Access the output */

    q = p2.out; /* Access the output */

  /* Do something with the outputs */

}
```

Background Information

Implements the following equations:

Case 1: When *rmp2_desired* > *rmp2_out*.

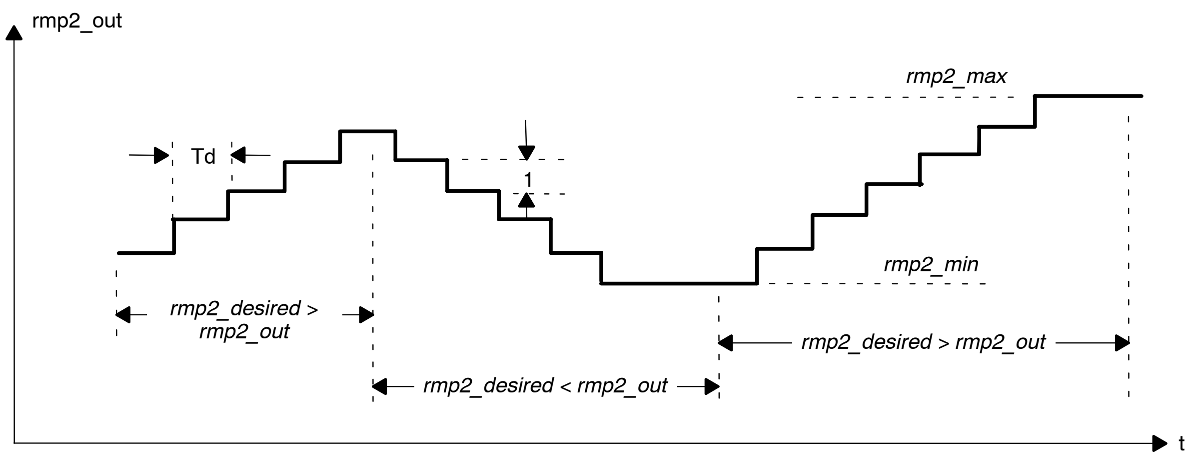
rmp2_out = *rmp2_out* + 1, for t = n ▪ Td, n = 1, 2, 3….. and (*rmp2_out* + 1)< *rmp2_max*
= *rmp2_max*, for (*rmp2_out* + 1)>rmp2_max

where,
Td = *rmp2_dly* ▪ Ts
Ts = Sampling time period

Case 2: When *rmp2_desired* < *rmp2_out*.

rmp2_out = *rmp2_out* − 1, for t = n ▪ Td, n = 1, 2, 3….. and (*rmp2_out* − 1)> *rmp2_min*
= *rmp2_min*, for (*rmp2_out* − 1)<*rmp2_min*

where,
Td = *rmp2_dly* ▪ Ts
Ts = Sampling time period



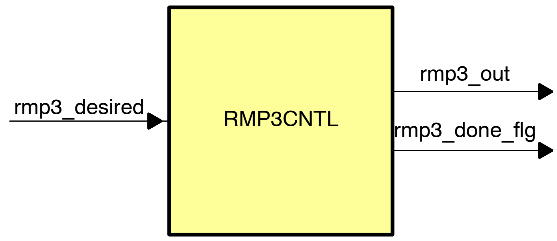
Example:

rmp2_out=0(initial value), rmp2_desired=1000(user specified), rmp2_dly=500(user specified), sampling loop time period Ts=0.000025 Sec.

This means that the time delay for each ramp step is Td=500x0.000025=0.0125 Sec. Therefore, the total ramp time will be Tramp=1000x0.0125 Sec=12.5 Sec

Description

This module implements a ramp down function. The output flag variable *rmp3_done_flg* is set to 7FFFh when the output variable *rmp3_out* equals the input variable *rmp3_desired*.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: rmp3cntl.asm, rmp3.h

C-Callable Version File Name: rmp3cntl.asm

Item	ASM Only	C-Callable ASM	Comments
Code size	33 words	45 words†	
Data RAM	6 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre−initialized RMP3 structure instance consumes 7 words in the data memory and 9 words in the .cinit section.

Direct ASM Interface

Table 58. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	rmp3_desired	Desired value of ramp 3	Q0	0–7FFF
Outputs	rmp3_out	Ramp 3 output	Q0	rmp3_min–7FFF
	rmp3_done_flg	Flag output for indicating ramp 3 status	Q0	0 or 7FFF
Init / Config	rmp3_min†	Minimum value of ramp 3	Q0	0–7FFF
	rmp3_dly†	Ramp 3 step delay in number of sampling cycles	Q0	0–7FFF
	rmp3_desired†	Desired value of ramp 3	Q0	0–7FFF
	rmp3_out†	Ramp 3 output	Q0	0–7FFF

† From the system file, initialize these variables as required by the application.

Variable Declaration:
In the system file include the following statements:

```
.ref    RMP3CNTL, RMP3CNTL_INIT             ;function call
.ref    rmp3_dly, rmp3_desired              ;input
.ref    rmp3_min, rmp3_done_flg, rmp3_out  ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section 'rmp3cntl'

Example:

```
ldp  #rmp3_desired                ;Set DP for module input
bldd #input_var1, rmp3_desired    ;Pass input variables to module inputs
bldd #input_var2, rmp3_dly

CALL RMP3CNTL

ldp  #output_var1                 ;Set DP for output variable

bldd #rmp3_out, output_var1  ;    Pass module outputs to output variables
bldd #rmp3_done_flg, output_var2
```

C/C-Callable ASM Interface

Object Definition

The structure of the RMP3 Object is defined by the following structure definition

```
/*-----------------------------------------------------------------------------
Define the structure of the RMP3
(Ramp3 control module)
-----------------------------------------------------------------------------*/

typedef struct{

  int desired;              /* Desired value of ramp3 */
  int dly;                  /* ramp3 step delay */
  int dly_cntr;             /* counter for ramp3 step delay  */
  int min;                  /* minimun value of ramp3 */
  int out;                  /* ramp3 output */
  int done_flg;             /* flag output for indicating ramp3 status */
  int (*calc)();            /* pointer to calculation function */
}RMP3;
```

Table 59. Module Terminal Variables/Functions

	Name	Description	Format	Range
Input	dly_cntr	Counter for ramp 3 step delay	Q0	0−7FFFh
	dly	Ramp 3 step delay in number of sampling cycles	Q0	0−7FFFh
	desired	Desired value of ramp 3	Q0	0−7FFFh
	min	Minimum value of ramp 3	Q0	0−7FFFh
Outputs	out	Ramp 3 output	Q0	min−7FFFh
	done_flg	Flag output for indicating ramp 3 status	Q0	0 or 7FFFh

Special Constants and Datatypes

RMP3
The module definition itself is created as a data type. This makes it convenient to instance a ramp3 control module.To create multiple instances of the module simply declare variables of type RMP3

RMP3_handle
Typedef'ed to RMP3 *

RMP3_DEFAULTS;
Initializer for the RMP3 Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc (RMP3_handle)
The default definition of the object implements just one method – the runtime implementation of the ramp3 control. This is implemented by means of a function pointer, and the default initializer sets this to rmp3_calc. The argument to this function is the address of the RMP3 object.

Module Usage

Instantiation:
The following example instances one such objects:

```
RMP3 p1,p2
```

Initialization:
To instance a pre-initialized object

```
RMP3 p1=RMP3_DEFAULTS, p2=RMP3_DEFAULTS;
```

Invoking the compute function:

```
p1.calc(&p1);
```

Example:
Lets instance two RMP3 objects,otherwise identical ,but running with different values

```
RMP3 p1 =RMP3_DEFAULTS;           /* initialization */
RMP3 p2 =RMP3_DEFAULTS;           /* initialization */

main()
{
    p1.desired = 3;
    p1.min     = 12;
    p1.out     = 15;
    p1.dly     = 3;

    p2.desired = 7;
    p2.min     = 30;
    p2.out     = 10;
    p2.dly     = 12;
}
void interrupt periodic_interrupt_isr()
{
    (*p1.calc)(&p1);        /* Call compute function for p1 */
    (*p2.calc)(&p2);        /* Call compute function for p2 */

     x=p1.out;              /* Access the output */
    y=p1.done_flg;          /* Access the output */

     p=p2.out;              /* Access the output */
    q=p2.done_flg;          /* Access the output */

/* Do something with the outputs */

}
```

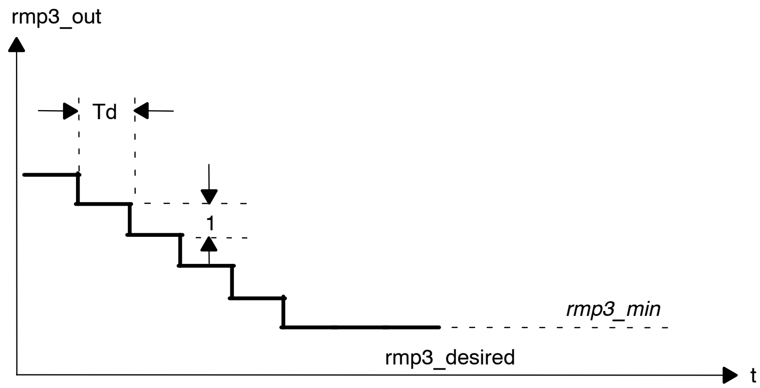
Background Information

Implements the following equations:

rmp3_out = *rmp3_out* − 1, for t = n ⋅ Td, n = 1, 2, 3….. and (*rmp3_out* − 1)> *rmp3_min*
= *rmp3_min*, for (*rmp3_out* − 1)<*rmp3_min*

rmp3_done_flg = 7FFF, when *rmp3_out* = *rmp3_desired* or *rmp3_min*

where,
Td = *rmp3_dly* ⋅ Ts
Ts = Sampling time period



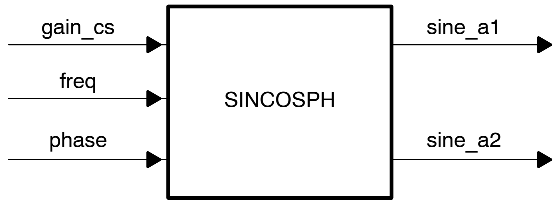
Example:

Rmp3_out=500(user specified initial value), rmp3_desired=20(user specified), Rmp3_dly=100(user specified), sampling loop time period Ts=0.000025 Sec.

This means that the time delay for each ramp step is Td=100x0.000025=0.0025 Sec. Therefore, the total ramp down time will be Tramp=(500−20)x0.0025 Sec=1.2 Sec

Description

The software module “SINCOSPH” generates two sine waves with variable magnitude (gain_cs), frequency (freq), and phase difference (phase). The two sine waves are “sine_a1” and “sine_a2”. The maximum magnitude of these waves set by the variable “gain_cs”. The frequency of the waves is set by “freq” and the phase difference is set by the variable “phase”.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: sincosph.asm

ASM Routines: SINCOSPH, SINCOSPH _INIT

C-Callable ASM File Names: sincosph.asm, sincosph.h

Item	ASM Only	C-Callable ASM	Comments
Code size	53 words	58 words†	
Data RAM	13 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized SINCOSPH structure instance consumes 7 words in the data memory and 9 words in the .cinit section.

Direct ASM Interface

Table 60. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	gain_cs	This sets the magnitude of the sine waves	Q15	0−7FFFh
	freq	The frequency of the sine waves (frequency is same for both waves)	Q15	0−7FFFh
	phase	This sets the phase difference between the waves.	Q00	0−180
Outputs	sine_a1	The first sine wave	Q15	8000h < sine_a1 < 7FFFh
	sine_a2	The second wave. The phase difference between first and second wave will be equal to “phase” angle	Q15	8000h < sine_a2 < 7FFFh
Init / Config	none			

Variable Declaration:
In the system file include the following statements:

```
.ref   SINCOSPH, SINCOSPH_INIT        ;function call
.ref   gain_cs, freq, phase           ;Inputs
.ref   sine_a1, sine_a2,              ;Outputs
```

Memory map:
All variables are mapped to an uninitialized named section ‘sincos’

Example:

```
LDP  #gain_cs            ; Set data page pointer (DP) for module
BLDD #v_out,    gain_cs  ; Passing input variables to module inputs.
                         ; Here “v_out” is the magnitude of the waves
BLDD #vhz_freq,   freq   ; “vhz_freq” is the frequency of the waves
BLDD   #phase_in, phase  ; “phase_in” is the phase angle.

CALL   SINCOSPH

LDP    #output_variable        ; Set DP for output variable
BLDD   #sine_a1, output_var1   ; Pass the value of first sine wave
BLDD   #sine_a2, output_var2   ; Pass the value of second sine wave
```

C/C-Callable ASM Interface

Object Definition

The structure of the SINCOSPH object is defined in the header file, sincosph.h, as seen in the following:

```
typedef struct  { int  phase_cs;      /* Input: Phase shift in degree (Q0) */
                  int  freq_cs;       /* Input: Frequency (Q15) */
                  int  gain_cs;       /* Input: Magnitude (Q15) */
                  int  sg2_freq_max;  /* Parameter: Maximum step angle (Q0) */
                  int  ALPHA_a1;      /* Variable: Incremental angle (Q0) */
                  int  sine_a1;       /* Output: Sinusoidal output 1 (Q15) */
                  int  sine_a2;       /* Output: Sinusoidal output 2 (Q15) */
                  int  (*calc)();     /* Pointer to calculation function */
                } SINCOSPH;
```

Special Constants and Datatypes

SINCOSPH
The module definition itself is created as a data type. This makes it convenient to instance a SINCOSPH object. To create multiple instances of the module simply declare variables of type SINCOSPH.

SINCOSPH_DEFAULTS
Initializer for the SINCOSPH object. This provides the initial values to the terminal variables, internal variables, as well as method pointers. This is initialized in the header file, sincosph.h.

Methods

void calc(SINCOSPH *);
This default definition of the object implements just one method – the runtime compute function for sine-cosine generation. This is implemented by means of a function pointer, and the default initializer sets this to sincosph_calc function. The argument to these functions is the address of the SINCOSPH object. Again, this statement is written in the header file, sincosph.h.

Module Usage

Instantiation:
The following example instances two such objects

```
SINCOSPH sc1, sc2;
```

Initialization:
To instance a pre-initialized object:

```
SINCOSPH sc1 = SINCOSPH_DEFAULTS;
SINCOSPH sc2 = SINCOSPH_DEFAULTS;
```

Invoking the compute function:

```
sc1.calc(&sc1);
sc2.calc(&sc2);
```

Example:

Lets instance two SINCOSPH objects, otherwise identical, and run two sine-cosine generators. The following example is the c source code for the system file.

```
SINCOSPH sc1= SINCOSPH_DEFAULTS;    /* instance the first object */
SINCOSPH sc2 = SINCOSPH_DEFAULTS;   /* instance the second object */

main()
{

    sc1.phase_cs = phase_in1;         /* Pass inputs to sc1 */
    sc1.freq_cs = freq_in1;           /* Pass inputs to sc1 */
    sc1.gain_cs = gain_in1;           /* Pass inputs to sc1 */
    sc2.phase_cs = phase_in2;         /* Pass inputs to sc2 */
    sc2.freq_cs = freq_in2;           /* Pass inputs to sc2 */
    sc2.gain_cs = gain_in2;           /* Pass inputs to sc2 */

}

void interrupt periodic_interrupt_isr()
{

    sc1.calc(&sc1);               /* Call compute function for sc1 */
    sc2.calc(&sc2);               /* Call compute function for sc2 */

sine1 = sc1.sine_a1;              /* Access the outputs of sc1 */
sine2 = sc1.sine_a2;              /* Access the outputs of sc1 */

sine3 = sc2.sine_a1;              /* Access the outputs of sc2 */
sine4 = sc2.sine_a2;              /* Access the outputs of sc2 */

}
```

Background Information

The generation of the sine wave is performed using a look up table. To be able to control the frequency of sine waves, a method based on the modulo mathematical operation is used. For more information, see *Digital Signal Processing applications with the TMS320 Familt: Theory, Algorithms, and Implementations, Volume 1*, (Literature Number SPRA012A).

A 16 bit software counter is used to determine the location of the next value of the sine waves. A step value is added to the counter every time a new value from the sine table is to be loaded. By changing the value of the step, one can accurately control the frequency of the sine wave.

Although a 16 bit counter is used, the upper byte determines the location of the next sine value to be used; thus, by changing how quickly values overflow from the lower byte (i.e. manipulating the step value), the frequency of the sine wave can be changed. The modulo mathematical operation is used when there is overflow in the accumulator from the lower word to the upper word. When an overflow occurs, only the remainder (lower word) is stored.

For example, the counter is set to 0000h and the step value is set to 40h. Every time a value is to be looked up in the table, the value 40h is added to the counter; however, since the upper byte is used as the pointer on the look up table, the first, second, and third values will point to the same location. In the fourth step, which results in an overflow into the upper byte, the value that is loaded will change. Since the upper byte is used as the pointer, the lookup table has 256 values, which is equivalent to the number of possibilities for an 8-bit number – 0 to 255. Additionally, since the upper word of the accumulator is disregarded, the pointer for the sine lookup table does not need to be reset.

Step	**Accumulator**	**Counter**	**Pointer**	**Step Value = 40h**
0	0000 0000h	0000h	00h	1st value of sine table
1	0000 0040h	0040h	00h	
2	0000 0080h	0080h	00h	
3	0000 00C0h	00C0h	00h	
4	0000 0100h	0100h	01h	2nd value of sine table
.	.	.	.	
.	.	.	.	
.	.	.	.	
n	0000 FFC0h	FFC0h	FFh	256th value of sine table
n+1	0001 0000h	0000h	00h	1st value of sine table
n+2	0000 0040h	0040h	00h	

The step size controls the frequency that is output; as a result, the larger the step, the quicker the overflow into the upper byte, and the faster the pointer traverses through the sine lookup table.

Step	Counter	Pointer	Step Value = C0h
0	0000h	00h	1st value of sine table
1	00C0h	00h	
2	0180h	01h	2nd value of sine table
3	0240h	02h	3rd value of sine table
4	0300h	03h	4th value of sine table

Although the step size indicates how quickly the pointer moves through the look up table, the step size does not provide much information about the approximate frequency that the sine wave will be modulating the PWM signal. To determine the frequency of the sine wave, one needs to determine how often the value in the compare register will be modified.

The frequency that the sine wave will be modulated at can be calculated from the following formula

$$f\,(step) = \frac{step}{T_s \times 2^n}$$

Where,

f(step) = desired frequency
T_S = the time period between each update (in this case, the PWM period)
n = the number of bits in the counter register
step = the step size used

The frequency that the PWM signal will be modulated is proportional to the step size and inversely proportional to the size of the counter register and the period at which the routine is accessed. Thus, to increase the resolution that one can increment or decrement the frequency of the PWM modulation, one needs to have a larger counting register or access the routine at a slower frequency by increasing the period.

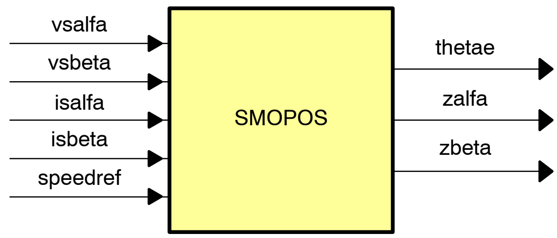
The second sine wave is generated using the same method. However, for the second wave a phase is also added with the counter before reading the value from the sine table.

SMOPOS

Permanent Magnet Synchronous Motor Angular Position Estimation Based on Sliding-Mode Observer

Description

This software module implements a rotor position estimation algorithm for Permanent-Magnet Synchronous Motor (PMSM) based on Sliding-Mode Observer (SMO).



Availability

This module is available in the direct-mode assembly-only interface (Direct ASM).

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Assembly File Name: smopos.asm

Routines: smopos, smopos_init

Parameter Calculation Spreadsheet: smopos.xls

Item	ASM Only	Comments
Code size	135 words	
Data RAM	25 words	
xDAIS module	No	
xDAIS component	No	IALG layer not implemented

Direct ASM Interface

Table 61. Module Terminal Variables/Functions

	Name	Description	Format	Range	Scale
Inputs	isalfa	α-axis phase current	Q15	−1.0 –> 0.999	Imax†
	isbeta	β-axis phase current	Q15	−1.0 –> 0.999	Imax
	vsalfa	α-axis phase voltage command	Q15	−1.0 –> 0.999	Vmax†
	vsbeta	β-axis phase voltage command	Q15	−1.0 –> 0.999	Vmax
	speedref	Reference speed	Q15	−1.0 –> 0.999	Spdmax †
Outputs	thetae	Estimated electric angular position	Q15	0 –> 0.999 (0−360 degree)	2*pi
	zalfa	α-axis sliding control	Q15	−1.0 –> 0.999	Vmax
	zbeta	β-axis sliding control	Q15	−1.0 –> 0.999	Vmax
Program Parameters	fsmopos_	F term of motor model	Q15	−1.0 –> 0.999	N/A
	gsmopos_	G term of motor model	Q15	−1.0 –> 0.999	N/A
	Kslide_	Bang-bang control gain	Q15	−1.0 –> 0.999	N/A

† The motor current and voltage are normalized with respect to Imax and Vmax, respectively. Here, $Vmax = Vbus/\sqrt{3}$ with Vbus being the Bus voltage. Note, selection of Imax affects the gain of current sampling circuit. Spdmax is what the motor speed is normalized against.

Routine names and calling limitation:

There are two routines involved:

smopos, the main routine; and
smopos_init, the initialization routine.

The initialization routine must be called during program (or incremental build) initialization. The smopos routine must be called in current control loop.

Global reference declarations:

In the system file include the following statements before calling the subroutines:

```
.ref smopos,smopos_init                   ; Function calls
.ref thetae,zalfa,zbeta                   ; Outputs
.ref vsalfa,vsbeta,isalfa,isbeta,spdref   ; Inputs
```

Memory map:

All variables are mapped to an uninitialized named section, *smopos*, which can be allocated to any one (128 words) data page.

Example:

```
CALL smopos_init         ; Initialize smopos

ldp  #vsalfa                 ; Set DP for module inputs
bldd #input_var1,vsalfa      ; Pass input variables to module inputs
bldd #input_var2,vsbeta      ;
bldd #input_var3,isalfa      ;
bldd #input_var4,isbeta      ;
bldd #input_var5,spdref      ;

CALL smopos

ldp  #output_var1            ; Set DP for output variable
bldd #thetae,output_var1     ; Pass output to other variable
…                            ; Pass more outputs to other variables
                             ; if needed.
```

Background Information

Figure 18 is an illustration of a permanent-magnet synchronous motor control system based on field orientation principle. The basic concept of field orientation is based on knowing the position of rotor flux and positioning the stator current vector at orthogonal angle to the rotor flux for optimal torque output. The implementation shown in Figure 18 derives the position of rotor flux from encoder feedback. However, the encoder increases system cost and complexity.

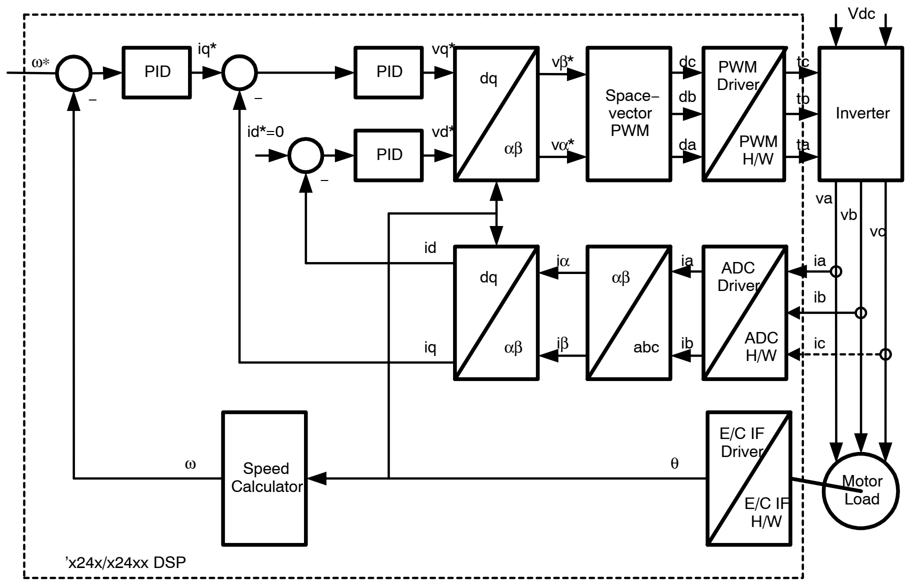


Figure 18. Field Oriented Control of PMSM

Therefore for cost sensitive applications, it is ideal if the rotor flux position information can be derived from measurement of voltages and currents. Figure 19 shows the block diagram of a sensorless PMSM control system where rotor flux position is derived from measurement of motor currents and knowledge of motor voltage commands.

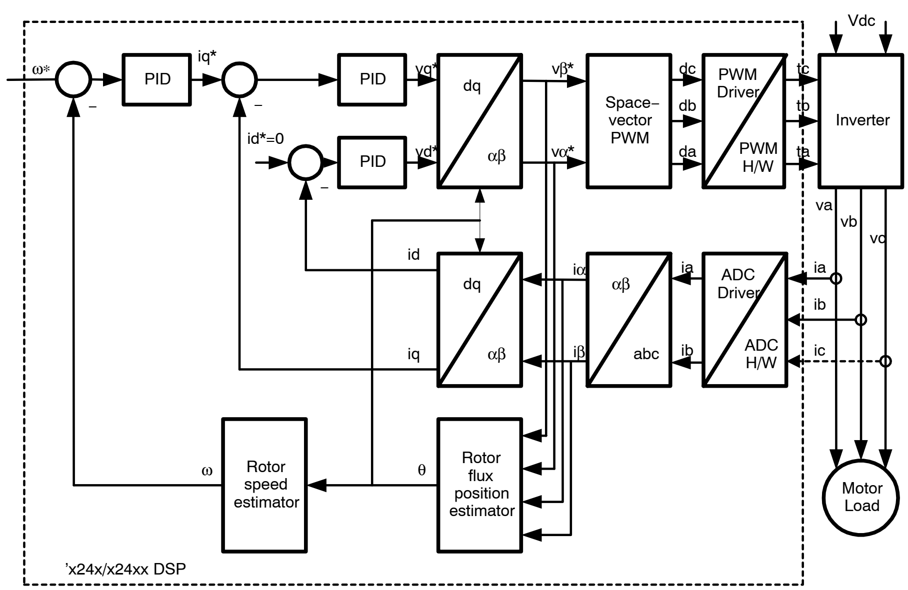


Figure 19. Sensorless Field-Oriented Control of PMSM

This software module implements a rotor flux position estimator based on a sliding mode current observer. As shown in Figure 20, the inputs to the estimator are motor phase currents and voltages expressed in α-β coordinate frame.

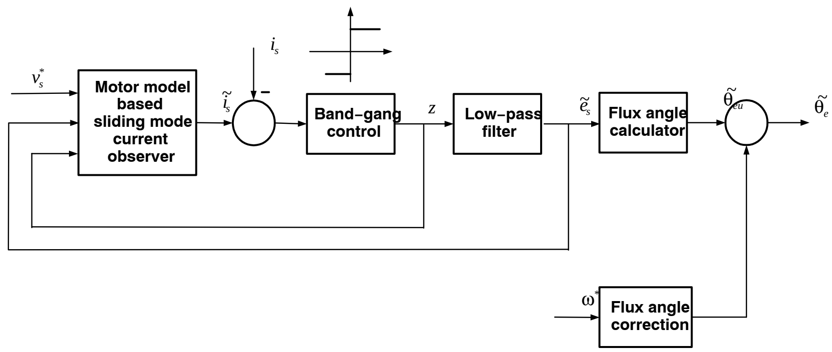


Figure 20. Sliding Mode Observer-Based Rotor Flux Position Estimator

Figure 21 is an illustration of the coordinate frames and voltage and current vectors of PMSM, with *a*, *b* and *c* being the phase axes, α and β being a fixed Cartesian coordinate frame aligned with phase *a*, and *d* and *q* being a rotating Cartesian coordinate frame aligned with rotor flux. v_s, i_s and e_s are the motor phase voltage, current and back emf vectors (each with two coordinate entries). All vectors are expressed in α-β coordinate frame for the purpose of this discussion. The α-β frame expressions are obtained by applying Clarke transformation to their corresponding three phase representations.

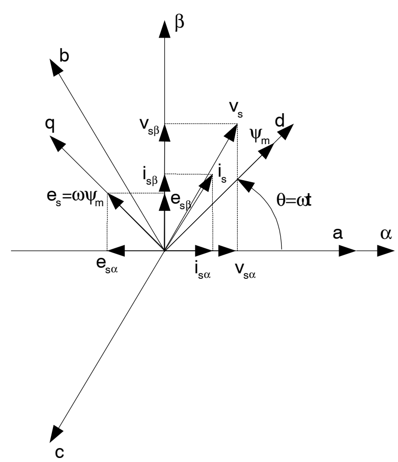


Figure 21. PMSM Coordinate Frames and Vectors

Equation 1 is the mathematical model of PMSM in α-β coordinate frame.

$$\frac{d}{dt} i_s = A i_s + B(v_s - e_s) \tag{1}$$

The matrices *A* and *B* are defined as $A = -\frac{R}{L} I_2$ and $B = \frac{1}{L} I_2$ with $L = \frac{3}{2} L_m$, where L_m and *R* are the magnetizing inductance and resistance of stator phase winding and I_2 is a 2 by 2 identity matrix.

1) **Sliding Mode Current Observer**

The sliding mode current observer consists of a model based current observer and a bang-bang control generator driven by error between estimated motor currents and actual motor currents. The mathematical equations for the observer and control generator are given by Equations 2 and 3.

$$\frac{d}{dt}\tilde{i}_s = A\tilde{i}_s + B(v_s^* - \tilde{e}_s - z) \qquad (2)$$

$$z = k\,\mathrm{sign}(\tilde{i}_s - i_s) \qquad (3)$$

The goal of the bang-bang control *z* is to drive current estimation error to zero. It is achieved by proper selection of *k* and correct formation of estimated back emf, $\tilde{e}_s$. Note that the symbol ~ indicates that a variable is estimated. The symbol * indicates that a variable is a command.

The discrete form of Equations 2 and 3 are given by Equations 4 and 5.

$$\tilde{i}_s(n+1) = F\tilde{i}_s(n) + G(v_s^*(n) - \tilde{e}_s(n) - z(n)) \qquad (4)$$

$$z(n) = k\,\mathrm{sign}(\tilde{i}_s(n) - i_s(n)) \qquad (5)$$

The matrices *F* and *G* are given by $F = e^{-\frac{R}{L}T_s} I_2$ and $G = \frac{1}{R}(1 - e^{-\frac{R}{L}T_s}) I_2$ where T_s is the sampling period.

2) **Estimated Back EMF**

Estimated back emf is obtained by filtering the bang-bang control, *z*, with a first order low-pass filter described by Equation 6.

$$\frac{d}{dt}\tilde{e}_s = -\omega_0\tilde{e}_s + \omega_0 z \qquad (6)$$

The parameter ω_0 is defined as $\omega_0 = 2\pi f_0$, where f_0 represents the cutoff frequency of the filter. The discrete form of Equation 6 is given by Equation 7.

$$\tilde{e}_s(n+1) = \tilde{e}_s(n) + 2\pi f_0 (z(n) - \tilde{e}_s(n)) \qquad (7)$$

3) **Rotor Flux Position Calculation**

Estimated rotor flux angle is obtained based on Equation 8 for back emf.

$$e_s = \frac{3}{2}k_e\omega\begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix} \qquad (8)$$

Therefore given the estimated back emf, estimated rotor position can be calculated based on Equation 9.

$$\tilde{\theta}_{eu} = \arctan(-\tilde{e}_{s\alpha}, \tilde{e}_{s\beta}) \qquad (9)$$

4) **Rotor Flux Position Correction**

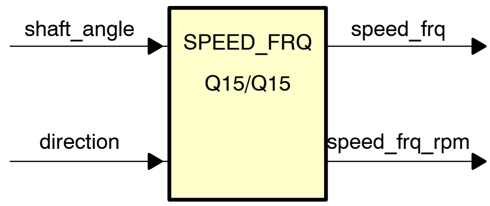
The low-pass filter used to obtain back emf introduces a phase delay. This delay is directly linked to the phase response of the low-pass filter and is often characterized by the cutoff frequency of the filter. The lower the cutoff frequency is, the bigger the phase delay for a fixed frequency. Based on the phase response of the low-pass filter, a lookup take for phase delay can be constructed. The command frequency is used as the index to lookup the table at run time to obtain the phase delay. This phase delay is then added to the calculated rotor flux angle to compensate for the delay introduced by the filter.

The following table describes the correspondence between variables and or parameters in the program and those used in the above mathematical equations and representations. Note that this software module assumes that both the input and output variables are per unit, i.e. they are both normalized with respect to their preselected maximums. The file *smopos.xls* that is used to calculate the program parameters has taken this into account.

Equation Variables		Program Variables
v_s*	$v_{s\alpha}$*	vsalfa
	$v_{s\beta}$*	vsbeta
i_s	$i_{s\alpha}$	isalfa
	$i_{s\beta}$	isbeta
$\tilde{i}_s$	$\tilde{i}_{s\alpha}$	isalfae
	$\tilde{i}_{s\beta}$	isbetae
$\tilde{i}_s$	$\tilde{e}_{s\alpha}$	esalfa (high word), esalfalo (low word)
	$\tilde{e}_{s\beta}$	esbeta (high word), esbetalo (low word)
	ω *	speedref
	$\tilde{\theta}_{eu}$	thetaeu
	$\tilde{\theta}_e$	thetae
z	z_α	zalfa
	z_β	zbeta
	$e^{-\frac{R}{L}T_s}$	fsmopos
	$\frac{1}{R}\left(1 - e^{-\frac{R}{L}T_s}\right)$	gsmopos
	k	kslide
	$2\pi f_0$	kslf

Description

This module calculates motor speed based on measurement of frequency of the signal generated by a speed sensor. The frequency of the speed sensor signal is the number of pulses generated per second, which is again proportional to the angular displacement of the sensor disk and hence that of the rotor. Therefore, this module gets the input as rotor shaft displacements (*shaft_angle*) for a known time interval and then uses this information to calculate the motor speed.



Availability

This module is available in direct-mode assembly-only interface (Direct ASM).

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: speed_fr.asm

Item	ASM Only	Comments
Code size	49 words	
Data RAM	9 words	
xDAIS ready	No	
xDAIS component	No	
Multiple instances	No	

Direct ASM Interface

Table 62. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	shaft_angle	Rotor displacement in pu mechanical degrees	Q15	0−7FFF (0−360 degree)
	direction	Rotor direction of rotation signal	Q0	0 or F
Outputs	speed_frq	Per unit motor speed	Q15	0−7FFF
	speed_frq_rpm	Motor speed in revolution/min	Q0	Application dependant
Init / Config	SPEED_LP _MAX	Time interval in number of sampling cycles for calculating the per unit mechanical displacement. The default value is set to 100. When this is used in a 10 kHz sampling loop, the time interval becomes 100x0.0001=0.01 sec. This means 1 pu mechanical displacement takes 0.01 sec, which sets the maximum measurable speed to 100 rps, or 6000 rpm. Set this parameter appropriately, according to the maximum speed of the motor.	Q0	User specified
	rpm_scaler	Maximum motor speed in rpm. Default value is set to 6000. Set this parameter appropriately, according to the maximum speed of the motor.	Q0	User specified

Variable Declaration:
In the system file include the following statements:

```
.ref   SPEED_FRQ, SPEED_FRQ_INIT                  ;function call
.ref   shaft_angle,direction,speed_frq,speed_frq_rpm  ;inputs/outputs
.ref   SPEED_LP_MAX, rpm_scaler
```

Memory map:
All variables are mapped to an uninitialized named section ‘speed_fr.’

Example:

```
ldp  #shaft_angle             ;Set DP for module inputs
bldd #input_var1, shaft_angle ;Pass input variables to module inputs
bldd #input_var2, direction

CALL SPEED_FRQ

ldp  #output_var1            ;Set DP for output variable
bldd #speed_frq, output_var1 ;Pass module outputs to output variables
bldd #speed_frq_rpm, output_var2
```

Background Information

This module calculates motor speed based on measurement of frequency of the signal generated by a speed sensor. The frequency of the speed sensor signal is the number of pulses generated per second, which is again proportional to the angular displacement of the sensor disk and hence that of the rotor. Therefore, this module gets it's input as rotor per unit shaft displacements (*shaft_angle*) for a known time interval and then use this information to calculate the motor speed.

Figure 22 shows a typical speed sensor disk mounted on a motor shaft f. When the motor rotates, the sensor generates quadrature pulses (QEP). The number of pulses generated is proportional to the angular displacement of the motor shaft. In Figure 22, a complete 360° rotation of motor shaft generates 1000 QEP pulses. 24x/24xx devices have an internal QEP interface circuit that can count these pulses. This QEP circuit counts both edges of the two QEP pulses. Therefore, the frequency of the counter clock in the QEP circuit is four times that of each input sequence. This means, for 1000 QEP pulses, the maximum counter value 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. This means that the shaft_angle input to this module, which represents the per unit mechanical displacement of the motor shaft at a certain instant of time, also represents the per unit counter value at the same instant of time. Figure 23 shows the instantaneous counter values for both forward and reverse direction of rotation.

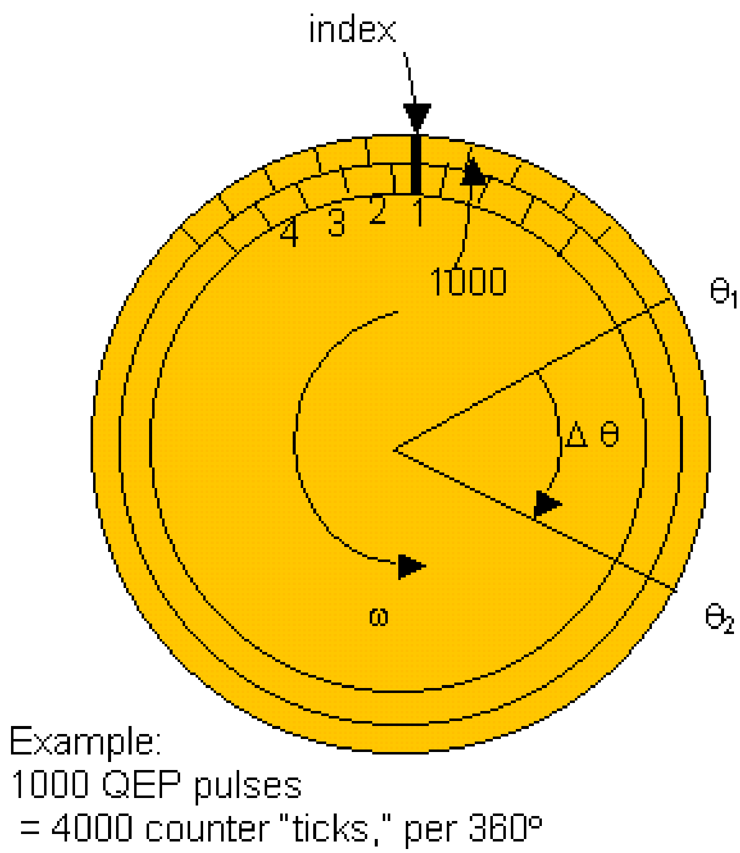


Figure 22. Speed sensor disk

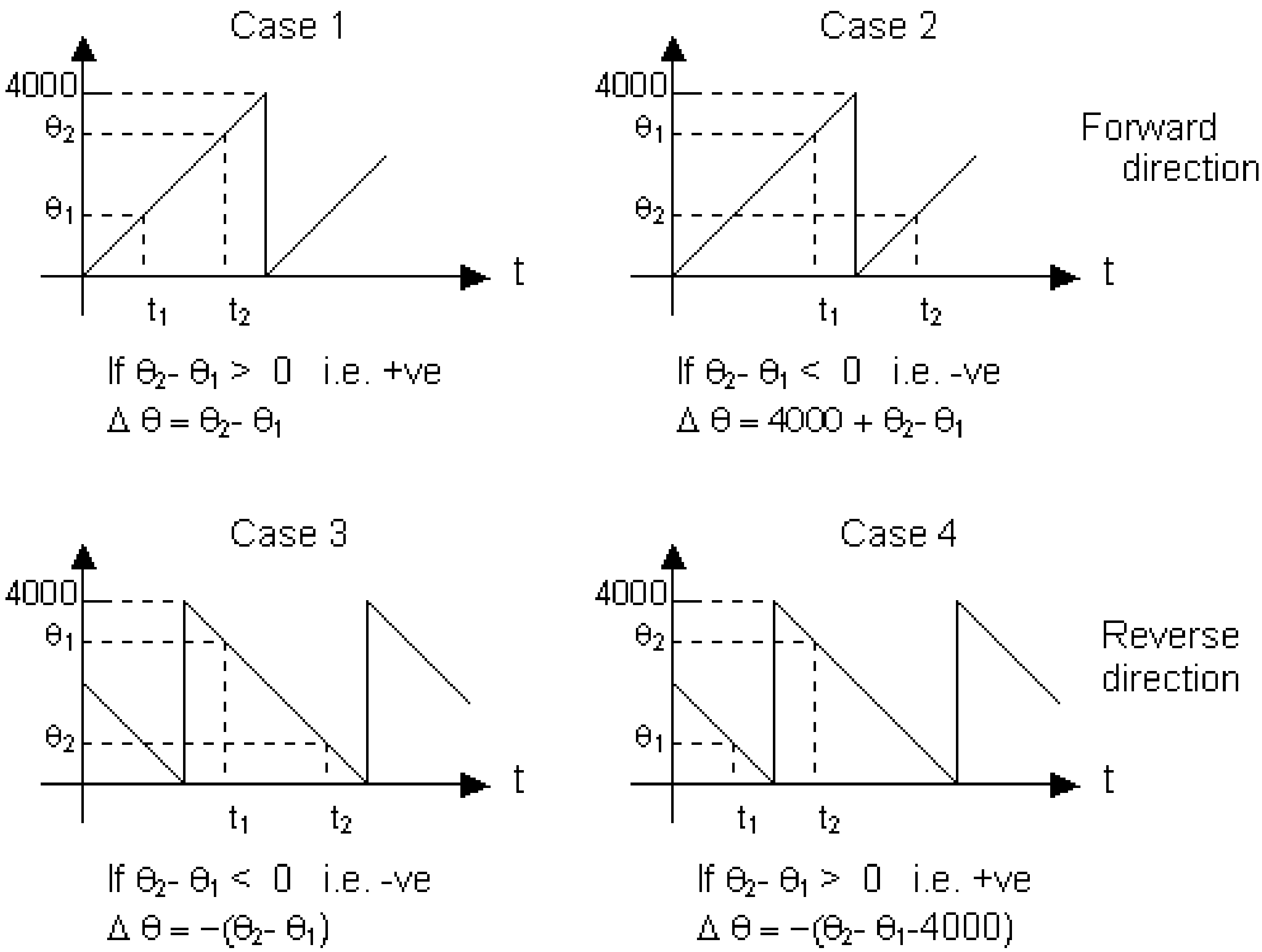


Figure 23. QEP counter values for forward and reverse direction

In each case in Figure 23, two per unit counter values are compared and the difference is calculated as indicated in the figure. This difference represents the per unit mechanical displacement of the rotor shaft. In the woftware, this difference is calculated for a time interval of 0.01 second. This again implies that the rotor makes a maximum of 100 revolutions in one second. This sets the maximum motor speed of 100 rps or 6000 rpm that can be measured when the time interval is set to 0.01 second. Now,

$$\Delta\theta = pu\ \ mechanical\ \ displacement$$
$$\Rightarrow\ speed\_frq\ = \ pu\ \ mech\ \ displacement\ = \Delta\theta$$

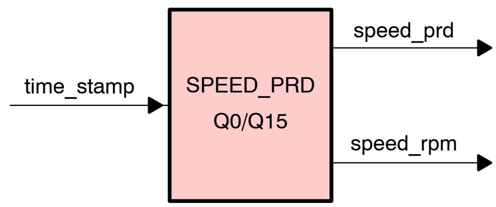
Then, the speed in rpm is derived as:

$$speed\ in\ revolution / \mathrm{min} = \mathrm{max}\ rpm\ speed \times pu\ speed$$
$$\Rightarrow speed\_frq\_rpm\ =\ (6000 \times speed\_frq)\ rpm$$

Variables in the equations	Variables in the code
θ_1	s_angle_old
θ_2	s_angle_cur
$\Delta\theta$	delta_angle

Description

This module calculates the motor speed based on a signal's period measurement. Such a signal, for which the period is measured, can be the periodic output pulses from a motor speed sensor.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version

Module Properties

Type: Target Dependent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: speed_pr.asm

C-Callable Version File Names: speed_pr.asm, speed_pr.h

Item	ASM Only	C-Callable ASM	Comments
Code size	55 words	64 words†	
Data RAM	13 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented

† Each pre-initialized SPEED_MEAS structure instance consumes 9 words in the data memory and 11 words in the .cinit section.

Direct ASM Interface

Table 63. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	time_stamp	Captured base timer counter value corresponding to the periodic edges of the sensed signal.	Q0	0−FFFF
Outputs	speed_prd	Normalized motor speed	Q15	0−7FFF
	speed_rpm	Motor speed in revolution per minute	Q0	0−rpm_max
Init / Config	rpm_max	Speed of normalization. The value chosen should be equal to or greater than the maximum motor speed.	Q0	Specified by user
	speed_scaler	Scaling constant. Use the Excel file to calculate this.	Q0	System dependent
	shift	Number of left shift less 1 required for max accuracy of 32bit/16bit division used for speed calculation. Use the Excel file to calculate this. When speed_scaler is calculated as 1, shift will be −1. In that case do not apply any left shift on the result of the 32bit/16bit division.	Q0	System dependent

Variable Declaration: In the system file include the following statements:

```
.ref   SPEED_PRD, SPEED_PRD _INIT                     ;function call
.ref    time_stamp, rpm_max, speed_scaler, shift     ;input
.ref    speed_prd, speed_rpm                         ;output
```

Memory map: All variables are mapped to an uninitialized named section ‘speedprd’

Example:

```
ldp  # time_stamp                 ;Set DP for module input
bldd #input_var1, time_stamp      ;Pass input to module input
CALL SPEED_PRD
ldp  #output_var1                 ;Set DP for output variables
bldd #speed_prd, output_var1      ;Pass module outputs to output
                                  ;variables
bldd #speed_rpm, output_var2
```

C/C-Callable ASM Interface

Object Definition

The structure of the SPEED_MEAS object is defined by the following structure definition

```
typedef struct {
  int time_stamp_new;    /*Variable: New 'Timestamp' corresponding to a capture event*/
  int time_stamp_old;    /*Variable: Old 'Timestamp' corresponding to a capture event*/
  int time_stamp;          /*Input: Current 'Timestamp' corresponding to a capture event*/
  int shift;                 /*Parameter: For maximum accuracy of 32bit/16bit division*/
  int speed_scaler;        /*Parameter: Scaler converting 1/N cycles to a Q15 speed*/
  int speed_prd;           /*Output: speed in per-unit
  int rpm_max;             /*Parameter: Scaler converting Q15 speed to rpm (Q0) speed*/
  int speed_rpm;         /*Output: speed in r.p.m.
  int (*calc) ();          /*Pointer to the calculation function*/
  } SPEED_MEAS;            /*Data type created*/
```

Table 64. Module Terminal Variables

	Name	Description	Format	Range
Inputs	time_stamp	Captured base timer counter value corresponding to the periodic edges of the sensed signal.	Q0	0−FFFF
Outputs	speed_prd	Normalized motor speed	Q15	0−7FFF
	speed_rpm	Motor speed in revolution per minute	Q0	0−rpm_max
Init / Config	rpm_max	Speed of normalization. The value chosen should be equal to or greater than the maximum motor speed.	Q0	Specified by user
	speed_scaler	Scaling constant. Use the Excel file to calculate this.	Q0	System dependent
	shift	Number of left shift less 1 required for max accuracy of 32bit/16bit division used for speed calculation. Use the Excel file to calculate this. When speed_scaler is calculated as 1, shift will be −1. In that case, do not apply any left shift on the result of the 32 bit/16 bit division.	Q0	System dependent

Special Constants and Datatypes

SPEED_MEAS

The module definition itself is created as a data type. This makes it convenient to instance a Space Vector Generation module. To create multiple instances of the module simply declare variables of type SVGENMF.

SPEED_PR_MEAS_DEFAULTS
Initializer for the SVGENMF Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(SPEED_MEAS *)
Pointer to the speed calculation function.

Module Usage

Instantiation:

```
SPEED_MEAS  shaftSpeed;
```

Initialization:
To instance a pre-initialized object

```
SPEED_MEAS  shaftSpeed=SPEED_PR_MEAS_DEFAULTS;
```

Invoking the compute function:

```
shaftSpeed.calc(&shaftSpeed);
```

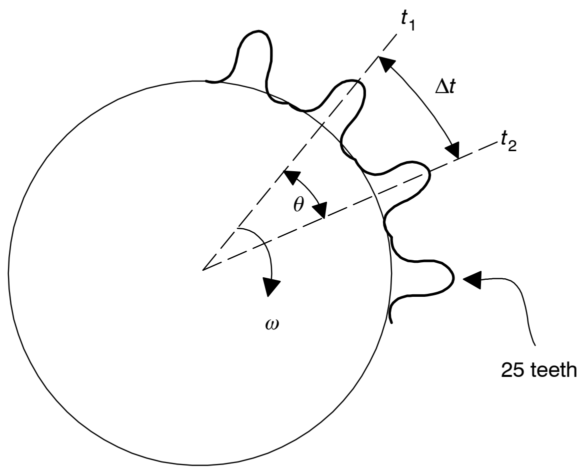
Example:

```
/*------------------------------------------------------------------
Pre initialized declaration for the speed measurement object.
------------------------------------------------------------------*/
        SPEED_MEAS shaftSpeed=SPEED_PR_MEAS_DEFAULTS;

/*------------------------------------------------------------------
Declaration for the capture driver. For more details see the CAP_DRV
document.
------------------------------------------------------------------*/
        CAPTURE cap=CAPTURE_DEFAULTS;
main()
{

/*------------------------------------------------------------------
    Initialize the capture interface
------------------------------------------------------------------*/
        cap.init(&cap);

}

void periodic_interrupt_isr()
{
/*-----------------------------------------------------------------
Call the capture driver read function. Note, that this func returns
the status, as the return value, NOT the time_stamp. The time_stamp
is returned directly into the CAPTURE object structure.
------------------------------------------------------------------*/
  if((cap.read(&cap))==0)  /* Call the capture read function */
  {
 shaftSpeed.time_stamp=cap.time_stamp; /* Read out new time stamp */
 shaftSpeed.calc(&shaftSpeed);        /* Call the speed calulator */
  }

}
```

Background Information

A low cost shaft sprocket with n teeth and a Hall effect gear tooth sensor is used to measure the motor speed. Figure 24 shows the physical details associated with the sprocket. The Hall effect sensor outputs a square wave pulse every time a tooth rotates within its proximity. The resultant pulse rate is n pulses per revolution. The Hall effect sensor output is fed directly to the 24x/24xx Capture input pin. The capture unit will capture (the value of it's base timer counter) on either the rising or the falling edges(whichever is specified) of the Hall effect sensor output. The captured value is passed to this s/w module through the variable called *time_stamp*.

In this module, every time a new input *time_stamp* becomes available it is compared with the previous *time_stamp*. Thus, the tooth-to-tooth period (t_2–t_1) value is calculated. In order to reduce jitter or period fluctuation, an average of the most recent n period measurements can be performed each time a new pulse is detected.



$$\Delta t = t_2 - t_1 \quad \text{sec}$$

$$\theta = \frac{360}{25} = 14.4°$$

$$\omega = \frac{2\pi}{\sum_{n=1}^{25} \Delta t_n} \quad \text{rad/sec}$$

Figure 24. Speed Measurement With a Sprocket

From the two consecutive *time_stamp* values the difference between the captured values are calculated as,

Δ = time_stamp(new) – time_stamp(old)

Then the time period in sec is given by,

$$\Delta t = t_2 - t_1 = K_p \times T_{CLK} \times \Delta$$

where,

K_P = Prescaler value for the Capture unit time base

T_{CLK} = CPU clock period in sec

From Figure 24, the angle θ in radian is given by,

$$\theta = \frac{2\pi}{n}$$

where,

n = number of teeth in the sprocket, i.e., the number of pulses per revolution

Then the speed ω in radian/sec and the normalized speed ω_N are calculated as,

$$\omega = \frac{\theta}{\Delta t} = \frac{2\pi}{n\Delta t} = \frac{2\pi}{n \times K_p \times T_{CLK} \times \Delta}$$

$$\Rightarrow \omega_N = \frac{\omega}{\omega_{\max}} = \frac{\omega}{2\pi\left(\frac{1}{n \times K_P \times T_{CLK}}\right)} = \frac{1}{\Delta}$$

Where, ω_{max} is the maximum value of ω which occurs when Δ=1. Therefore,

$$\omega_{\max} = \frac{2\pi}{nK_P T_{CLK}}$$

For, n=25, K_P=32 and T_{CLK}=50x10^{-9} sec (20MHz CPU clock), the normalized speed ω_N is given by,

$$\omega_N = \frac{\omega}{2\pi(25000)} = \frac{1}{\Delta}$$

The system parameters chosen above allows maximum speed measurement of 1500,000 rpm. Now, in any practical implementation the maximum motor speed will be significantly lower than this maximum measurable speed. So, for example, if the motor used has a maximum operating speed of 23000 rpm, then the calculated speed can be expressed as a normalized value with a base value of normalization of at least 23000 RPM. If we choose this base value of normalization as 23438 rpm, then the corresponding base value of normalization, in rad/sec, is,

$$\omega_{max1} = \frac{23438 \times 2\pi}{60} \approx 2\pi(390)$$

Therefore, the scaled normalized speed is calculated as,

$$\omega_{N1} = \frac{\omega}{2\pi(390)} \approx \frac{64}{\Delta} = 64 \times \omega_N = speed\_scaler \times \omega_N$$

This shows that in this case the scaling factor is 64.

The speed, in rpm, is calculated as,

$$N_1 = 23438 \times \omega_{N1} = 23438 \times \frac{64}{\Delta} = rpm\_max \times \omega_{N1}$$

The capture unit in 24x/24xx allows accurate time measurement (in multiples of clock cycles and defined by a prescaler selection) between events. In this case the events are selected to be the rising edge of the incoming pulse train. What we are interested in is the delta time between events and hence for this implementation Timer 1 is allowed to free run with a prescale of 32 (1.6uS resolution for 20MHz CPU clock) and the delta time Δ, in scaled clock counts, is calculated as shown in Figure 25.

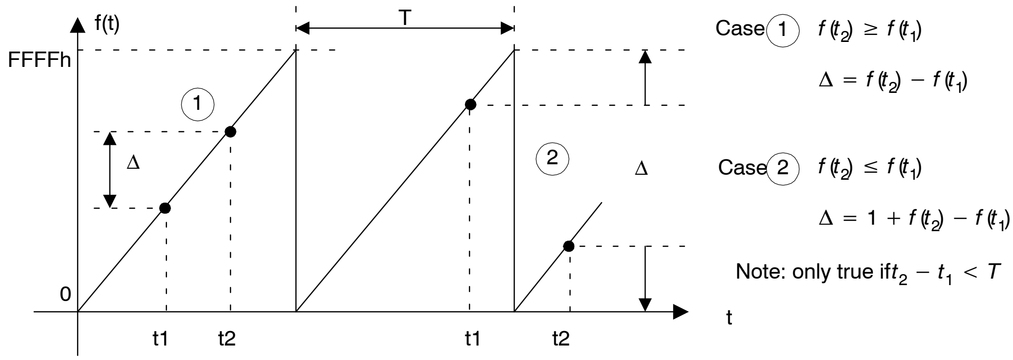


Figure 25. Calculation of Speed

In Figure 25, the vertical axis *f(t)* represents the value of the Timer counter which is running in continuous up count mode and resetting when the period register = FFFFh. Note that two cases need to be accounted for: the simple case where the Timer has not wrapped around and where it has wrapped around. By keeping the current and previous capture values it is easy to test for each of these cases.

Once a “robust” period measurement is extracted from the averaging algorithm, the speed is calculated using the appropriate equations explained before. In order to maintain high precision in the calculation for the full range of motor speeds, a 32-bit/16-bit division is performed as shown in Figure 26 in the following.

$$\frac{1}{period} = \frac{7FFFFFFF(Q31)}{period(Q0)} = speed(Q31 \rightarrow 32bit)$$

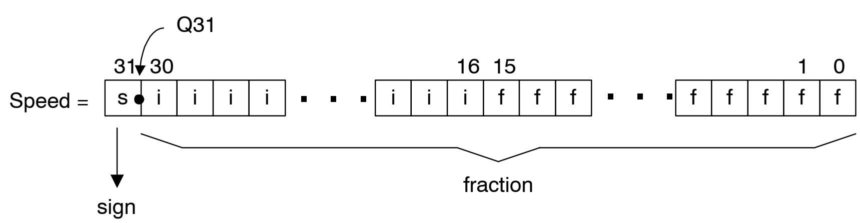


Figure 26. 32-Bit/16-Bit Division

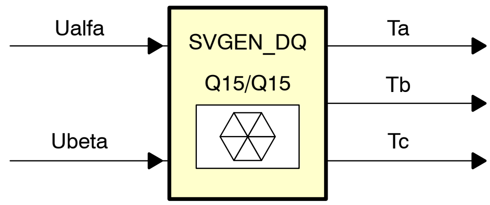
Once complete the result is a 32-bit value in *Q*31 format. This value is subsequently scaled to a 16 bit, *Q*15 format value for later calculation of the speed error (see Figure 26).

Table 65. Variable Cross Ref Table

Variables in the Equations	Variables in the Code
Δ	event_period
ω_N	speed_prd_max
ω_{N1}	speed_prd
N_1	speed_rpm

Description

This module calculates the appropriate duty ratios needed to generate a given stator reference voltage using space vector PWM technique. The stator reference voltage is described by it's (α,β) components, Ualfa and Ubeta.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: svgen_dq.asm

C-Callable Version File Names: svgen_dq.asm,svgen.h

Item	ASM Only	C-Callable ASM	Comments
Code size	179 words	215 words†	
Data RAM	12 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized SVGENDQ structure instance consumes 6 words in the data memory and 8 words in the .cinit section.

Direct ASM Interface

Table 66. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	Ualfa	Component of reference stator voltage vector on direct axis stationary reference frame.	Q15	8000−7FFF
	Ubeta	Component of reference stator voltage vector on quadrature axis stationary reference frame.	Q15	8000−7FFF
Outputs	Ta	Duty ratio of PWM1(CMPR1 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
	Tb	Duty ratio of PWM3(CMPR2 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
	Tc	Duty ratio of PWM5(CMPR3 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
Init / Config	none			

Variable Declaration:

In the system file include the following statements:

```
.ref   SVGEN_DQ, SVGEN_DQ _INIT       ;function call

.ref   Ualfa, Ubeta, Ta, Tb, Tc       ;input/output
```

Memory map:

All variables are mapped to an uninitialized named section ‘svgen_dq’

Example:

```
ldp  #Ualfa               ;Set DP for module input
bldd #input_var1, Ualfa   ;Pass input variables to module inputs
bldd #input_var2, Ubeta
CALL SVGEN_DQ
ldp  #output_var1         ;Set DP for output variable
bldd #Ta, output_var1     ;Pass module outputs to output
                          ;variables
bldd #Tb, output_var2
bldd #Tc, output_var3
```

C/C-Callable ASM Interface

Object Definition

The structure of the SVGENDQ object is defined by the following structure definition

```
/*-----------------------------------------------------------------
Define the structure of the SVGENMF
(Magnitude and angular velocity based Space Vector Waveform Generator)
-----------------------------------------------------------------*/

typedef struct     {   int d;   /* Phase d input Q15             */
                       int q;   /* Phase q input Q15             */
                       int va;  /* Phase A output Q15            */
                       int vb;  /* Phase B output Q15            */
                       int vc;  /* Phase C output Q15            */
                       int (*calc)(); /*Ptr to calculation function*/
                   } SVGENDQ;
```

Table 67. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	d	Component of reference stator voltage vector on direct axis stationary reference frame.	Q15	8000−7FFF
	q	Component of reference stator voltage vector on quadrature axis stationary reference frame.	Q15	8000−7FFF
Outputs	va	Duty ratio of PWM1(CMPR1 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
	vb	Duty ratio of PWM3(CMPR2 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
	vc	Duty ratio of PWM5(CMPR3 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
Init / Config	none			

Special Constants and Datatypes

SVGENDQ
The module definition itself is created as a data type. This makes it convenient to instance a Space Vector Generation module. To create multiple instances of the module simply declare variables of type SVGENDQ

SVGENDQ_handle
Typedef'ed to SVGENDQ

SVGENDQ_DEFAULTS
Initializer for the SVGENDQ object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(SVGENMF_handle)
The default definition of the object implements just one method – the runtime compute function for the generation of the space vector modulation functions. This is implemented by means of a function pointer, and the default initializer sets this to svgenmf_calc. The argument to this function is the address of the SVGENMF object.

Module Usage

Instantiation:

```
SVGENDQ   sv1,sv2;
```

Initialization:
To instance a pre-initialized object

```
SVGENDQ  sv1=SVGENDQ_DEFAULTS;
```

Invoking the compute function:

```
sv1.calc(&sv1);
```

Example:
Lets instance two SVGENMF objects, otherwise identical, but running with different freq values.

```
SVGENMF sv1=SVGENDQ_DEFAULTS; /* Instance the first object */
. . .  Other var declarations . . .
main()
{
}
void interrupt periodic_interrupt_isr()
{

  voltage_d=some_sine_wave_input;
  voltage_q=signal_90_deg_off_phase_wrt_above;

  sv1.d=voltage_d;
  sv1.q=voltage_q;

/* Transform from quadrature sine inputs to three-phase & space vector */
  sv1.calc(&sv1);

  v1=sv1.va; /* Access the outputs of the svgendq */
  v2=sv1.vb;
  v3=sv1.vc;

. . .  Do something with v1,v2,v3 . . .
}
```

Background Information

The Space Vector Pulse Width Modulation (SVPWM) refers to a special switching sequence of the upper three power devices of a three-phase voltage source inverters (VSI) used in application such as AC induction and permanent magnet synchronous motor drives. This special switching scheme for the power devices results in 3 pseudo-sinusoidal currents in the stator phases.

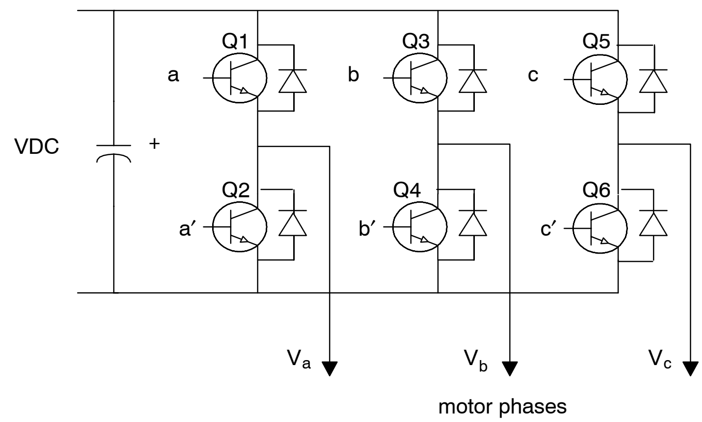


Figure 27. Power Circuit Topology for a Three-Phase VSI

It has been shown that SVPWM generates less harmonic distortion in the output voltages or currents in the windings of the motor load and provides more efficient use of DC supply voltage, in comparison to direct sinusoidal modulation technique.

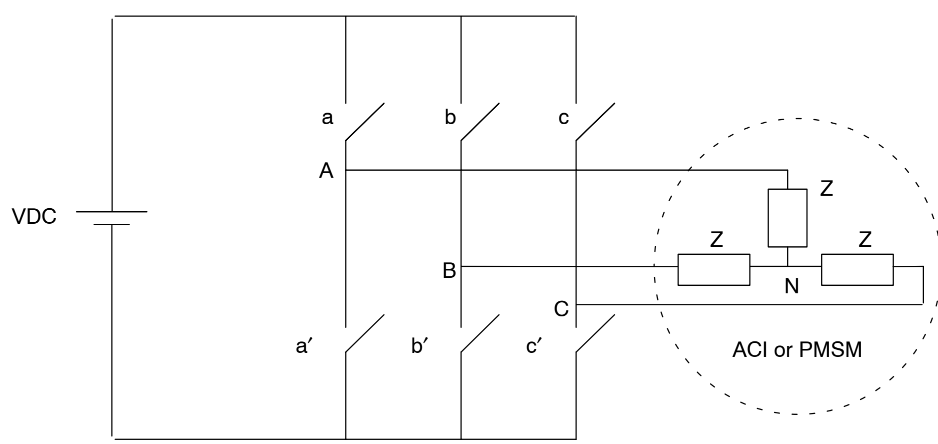


Figure 28. Power Bridge for a Three-Phase VSI

For the three phase power inverter configurations shown in Figure 27 and Figure 28, there are eight possible combinations of on and off states of the upper power transistors. These combinations and the resulting instantaneous output line-to-line and phase voltages, for a dc bus voltage of $V_{DC,}$ are shown in Table 68.

Table 68. Device On/Off Patterns and Resulting Instantaneous Voltages of a 3-Phase Power Inverter

c	b	a	V_{AN}	V_{BN}	V_{CN}	V_{AB}	V_{BC}	V_{CA}
0	0	0	0	0	0	0	0	0
0	0	1	$2V_{DC}/3$	$-V_{DC}/3$	$-V_{DC}/3$	V_{DC}	0	$-V_{DC}$
0	1	0	$-V_{DC}/3$	$2V_{DC}/3$	$-V_{DC}/3$	$-V_{DC}$	V_{DC}	0
0	1	1	$V_{DC}/3$	$V_{DC}/3$	$-2V_{DC}/3$	0	V_{DC}	$-V_{DC}$
1	0	0	$-V_{DC}/3$	$-V_{DC}/3$	$2V_{DC}/3$	0	$-V_{DC}$	V_{DC}
1	0	1	$V_{DC}/3$	$-2V_{DC}/3$	$V_{DC}/3$	V_{DC}	$-V_{DC}$	0
1	1	0	$-2V_{DC}/3$	$V_{DC}/3$	$V_{DC}/3$	$-V_{DC}$	0	V_{DC}
1	1	1	0	0	0	0	0	0

The quadrature quantities (in the (α,β) frame) corresponding to these 3 phase voltages are given by the general Clarke transform equation:

$$V_{s\alpha} = V_{AN}$$

$$V_{s\beta} = (2V_{BN} + V_{AN}) / \sqrt{3}$$

In matrix from the above equation is also expressed as,

$$\begin{bmatrix} V_{s\alpha} \\ V_{s\beta} \end{bmatrix} = \frac{2}{3}\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix}\begin{bmatrix} V_{AN} \\ V_{BN} \\ V_{CN} \end{bmatrix}$$

Due to the fact that only 8 combinations are possible for the power switches, $V_{s\alpha}$ and $V_{s\beta}$ can also take only a finite number of values in the (α,β) frame according to the status of the transistor command signals (c,b,a). These values of $V_{s\alpha}$ and $V_{s\beta}$ for the corresponding instantaneous values of the phase voltages (V_{AN}, V_{BN}, V_{CN}) are listed in Table 69.

Table 69. Switching Patterns, Corresponding Space Vectors and their (α,β) Components

c	b	a	$V_{s\alpha}$	$V_{s\beta}$	Vector
0	0	0	0	0	O_0
0	0	1	$\frac{2}{3}V_{DC}$	0	U_0
0	1	0	$\frac{V_{DC}}{3}$	$\frac{V_{DC}}{\sqrt{3}}$	U_{120}
0	1	1	$\frac{V_{DC}}{3}$	$\frac{V_{DC}}{\sqrt{3}}$	U_{60}
1	0	0	$-\frac{V_{DC}}{3}$	$-\frac{V_{DC}}{\sqrt{3}}$	U_{240}
1	0	1	$\frac{V_{DC}}{3}$	$-\frac{V_{DC}}{\sqrt{3}}$	U_{300}
1	1	0	$-\frac{2}{3}V_{DC}$	0	U_{180}
1	1	1	0	0	O_{111}

These values of $V_{s\alpha}$ and $V_{s\beta}$, listed in Table 69, are called the (α,β) components of the basic space vectors corresponding to the appropriate transistor command signal (c,b,a). The space vectors corresponding to the signal (c,b,a) are listed in the last column in Table 69. For example, (c,b,a)=001 indicates that the space vector is U_0.The eight basic space vectors defined by the combination of the switches are also shown in Figure 29.

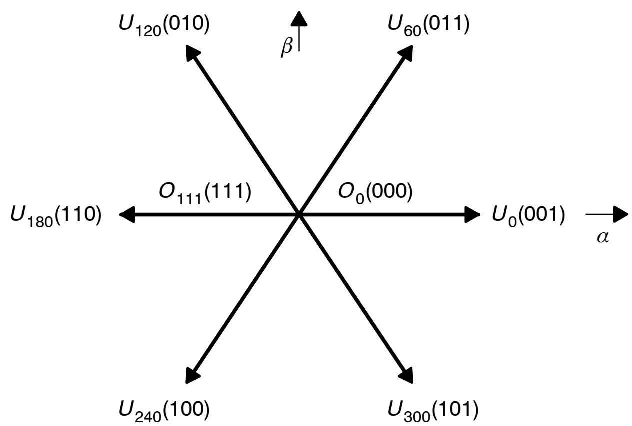


Figure 29. Basic Space Vectors

Projection of the stator reference voltage vector U_{out}

The objective of Space Vector PWM technique is to approximate a given stator reference voltage vector U_{out} by combination of the switching pattern corresponding to the basic space vectors. The reference vector U_{out} is represented by its (α,β) components, Ualfa and Ubeta. Figure 30 shows the reference voltage vector, it's (α,β) components and two of the basic space vectors, U_0 and U_{60}. The figure also indicates the resultant α and β components for the space vectors U_0 and U_{60}. $\Sigma V_{s\beta}$ represents the sum of the β components of U_0 and U_{60}, while $\Sigma V_{s\alpha}$ represents the sum of the α components of U_0 and U_{60}. Therefore,

$$\begin{cases} \sum V_{s\beta} = 0 + \dfrac{V_{DC}}{\sqrt{3}} = \dfrac{V_{DC}}{\sqrt{3}} \\ \sum V_{s\alpha} = \dfrac{2V_{DC}}{3} + \dfrac{V_{DC}}{3} = V_{DC} \end{cases}$$

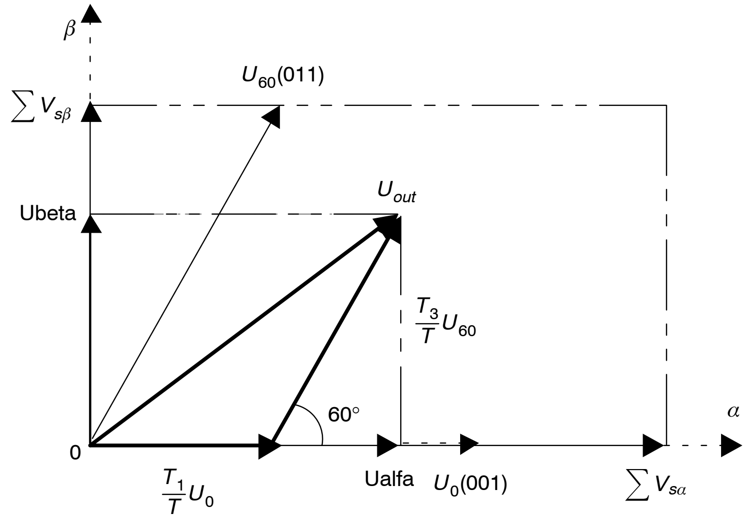


Figure 30. Projection of the Reference Voltage Vector

For the case in Figure 30, the reference vector U_{out} is in the sector contained by U_0 and U_{60}. Therefore U_{out} is represented by U_0 and U_{60}. So we can write,

$$\begin{cases} T = T_1 + T_3 + T_0 \\ U_{out} = \frac{T_1}{T}U_0 + \frac{T_3}{T}U_{60} \end{cases}$$

where, T_1 and T_3 are the respective durations in time for which U_0 and U_{60} are applied within period T. T_0 is the time duration for which the null vector is applied. These time durations can be calculated as follows:

$$\begin{cases} U_{beta} = \frac{T_3}{T}|U_{60}| \sin(60°) \\ U_{alfa} = \frac{T_1}{T}|U_0| + \frac{T_3}{T}|U_{60}| \cos(60°) \end{cases}$$

From Table 69 and Figure 30 it is evident that the magnitude of all the space vectors is $2V_{DC}/3$. When this is normalized by the maximum phase voltage(line to neutral), $V_{DC}/\sqrt{3}$, the magnitude of the space vectors become $2/\sqrt{3}$ i.e., the normalized magnitudes are $|U_0| = |U_{60}| = 2/\sqrt{3}$. Therefore, from the last two equations the time durations are calculated as,

$$T_1 = \frac{T}{2}\left(\sqrt{3}\,U_{alfa} - U_{beta}\right)$$

$$T_3 = TU_{beta}$$

Where, Ualfa and Ubeta also represent the normalized (α,β) components of U_{out} with respect to the maximum phase voltage($V_{DC}/\sqrt{3}$). The rest of the period is spent in applying the null vector T_0. The time durations, as a fraction of the total T, are given by,

$$t1 = \frac{T_1}{T}\left(\sqrt{3}\,U_{alfa} - U_{beta}\right)$$

$$t2 = \frac{T_3}{T} = U_{beta}$$

In a similar manner, if U_{out} is in sector contained by U_{60} and $U_{120,}$ then by knowing $|U60| = |U120| = 2/\sqrt{3}$ (normalized with respect to $V_{DC}/\sqrt{3}$), the time durations can be derived as,

$$t1 = \frac{T_2}{T} = \frac{1}{2}\left(-\sqrt{3}\,U_{alfa} + U_{beta}\right)$$

$$t2 = \frac{T_3}{T} = \frac{1}{2}\left(\sqrt{3}\,U_{alfa} + U_{beta}\right)$$

where, T_2 is the duration in time for which U_{120} is applied within period T

Now, if we define 3 variables X, Y and Z according to the following equations,

$$X = U_{beta}$$

$$Y = \frac{1}{2}\left(\sqrt{3}\,U_{alfa} + U_{beta}\right)$$

$$Z = \frac{1}{2}\left(-\sqrt{3}\,U_{alfa} + U_{beta}\right)$$

Then for the first example, when U_{out} is in sector contained by U_0 and $U_{60,}$ t1= –Z, t2=X.

For the second example, when U_{out} is in sector contained by U_{60} and U_{120}, t1=Z, t2=Y.

In a similar manner t1 and t2 can be calculated for the cases when U_{out} is in sectors contained by other space vectors. For different sectors the expressions for t1 and t2 in terms of X, Y and Z are listed in Table 70.

Table 70. t1 and t2 Definitions for Different Sectors in Terms of X, Y and Z Variables

Sector	U_0, U_{60}	U_{60}, U_{120}	U_{120}, U_{180}	U_{180}, U_{240}	U_{240}, U_{300}	U_{300}, U_0
t1	–Z	Z	X	–X	–Y	Y
t2	X	Y	Y	Z	–Z	–X

In order to know which of the above variables apply, the knowledge of the sector containing the reference voltage vector is needed. This is achieved by first converting the (α,β) components of the reference vector U_{out} into a balanced three phase quantities. That is, Ualfa and Ubeta are converted to a balanced three phase quantities V_{ref1}, V_{ref1} and V_{ref1} according to the following inverse clarke transformation:

$$\begin{cases} V_{ref1} = U_{beta} \\ V_{ref2} = \dfrac{-U_{beta} + U_{alfa} \times \sqrt{3}}{2} \\ V_{ref3} = \dfrac{-U_{beta} - U_{alfa} \times \sqrt{3}}{2} \end{cases}$$

Note that, this transformation projects the quadrature or β component, Ubeta, into $V_{ref1.}$ This means that the voltages V_{ref1} V_{ref2} and V_{ref3} are all phase advanced by 90^O when compared to the corresponding voltages generated by the conventional inverse clarke transformation which projects the α component, Ualfa, into phase voltage V_{AN}. The following equations describe the (α,β) components and the reference voltages:

$$\begin{cases} U_{alfa} = \sin \omega t \\ U_{beta} = \cos \omega t \\ V_{ref1} = \cos \omega t \\ V_{ref2} = \cos(\omega t - 120^\circ) \\ V_{ref3} = \cos(\omega t + 120^\circ) \end{cases}$$

Note that, the above voltages are all normalized by the maximum phase voltage($V_{DC}/\sqrt{3}$).

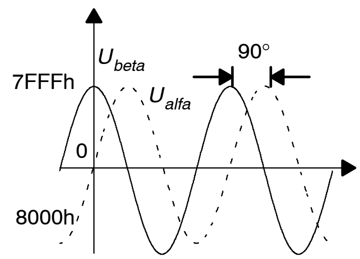


Figure 31. (α,β) Components of Stator Reference Voltage

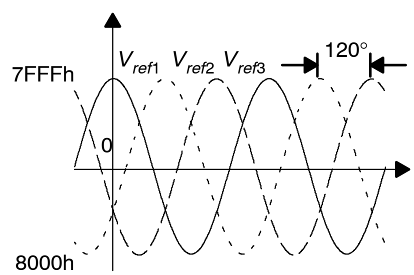


Figure 32. Voltages V_{ref1} V_{ref2} and V_{ref3}

From the last three equations the following decisions can be made on the sector information:

If $V_{ref1} > 0$ then a=1, else a=0
If $V_{ref2} > 0$ then b=1, else b=0
If $V_{ref3} > 0$ then c=1, else c=0

The variable *sector* in the code is defined as, sector = 4∗c+2∗b+a

For example, in Figure 29 a=1 for the vectors U_{300}, U_0 and U_{60}. For these vectors the phase of V_{ref1} are ωt=300°, ωt=0 and ωt=60° respectively. Therefore, $V_{ref1} > 0$ when a=1.

The (α,β) components, Ualfa and Ubeta, defined above represent the output phase voltages V_{AN}, V_{BN} and V_{CN}. The following equations describe these phase voltages:

$$\begin{cases} V_{AN} = \sin \omega t \\ V_{BN} = \sin(\omega t + 120^\circ) \\ V_{CN} = \sin(\omega t - 120^\circ) \end{cases}$$

The Space Vector PWM module is divided in several parts:

- Determination of the sector
- Calculation of *X, Y* and *Z*

- ❑ Calculation of t_1 and t_2
- ❑ Determination of the duty cycle *taon*, *tbon* and *tcon*
- ❑ Assignment of the duty cycles to *Ta*, *Tb* and *Tc*

The variables t_{aon}, t_{bon} and t_{con} are calculated using the following equations:

$$\begin{cases} t_{aon} = \dfrac{PWMPRD - t_1 - t_2}{2} \\ t_{bon} = t_{aon} + t_1 \\ t_{con} = T_{bon} + t_2 \end{cases}$$

Then the right duty cycle (txon) is assigned to the right motor phase (in other words, to Ta, Tb and Tc) according to the sector. Table 71 depicts this determination.

Table 71. Assigning the Right Duty Cycle to the Right Motor Phase

Sector	U_0, U_{60}	U_{60}, U_{120}	U_{120}, U_{180}	U_{180}, U_{240}	U_{240}, U_{300}	U_{300}, U_0
Ta	taon	tbon	tcon	tcon	tbon	taon
Tb	tbon	taon	taon	tbon	tcon	tcon
Tc	tcon	tcon	tbon	taon	taon	tbon

Example:
Sector contained by U_0 and U_{60}.

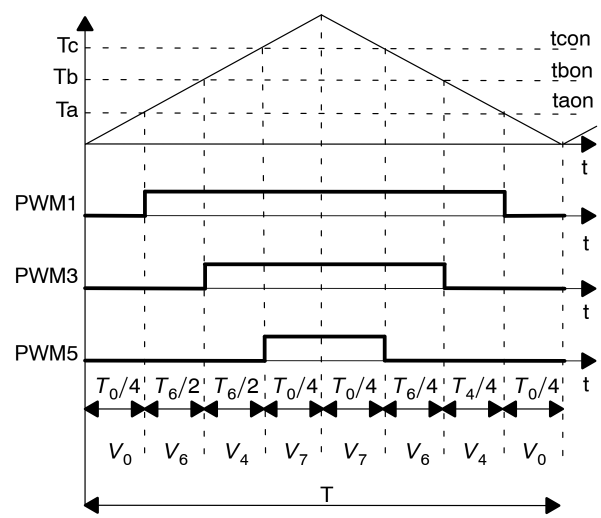


Figure 33. PWM Patterns and Duty Cycles for Sector Contained by U_0 and U_{60}

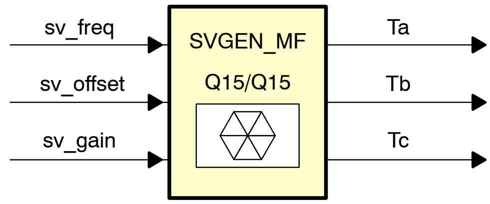
Table 72. Variable Cross Ref Table

Variables in the Equations	Variables in the Code
a	r1
b	r2
c	r3
V_{ref1}	Va

Variables in the Equations	Variables in the Code
V_{ref2}	Vb
V_{ref3}	Vc

Description

This module calculates the appropriate duty ratios needed to generate a given stator reference voltage using space vector PWM technique. The stator reference voltage is described by it's magnitude and frequency.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Independent, Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: svgen_mf.asm

C-Callable Version File Name: svgen_mf.asm

Item	ASM Only	C-Callable ASM	Comments
Code size	427 words	454 words†	
Data RAM	16 words	0 words†	
xDAIS ready	No	Yes	
xDAIS component	No	No	IALG layer not implemented
Multiple instances	No	Yes	

† Each pre-initialized SVGENMF structure consumes 11 words in the .cinit section instance and 9 words in data memory.

Direct ASM Interface

Table 73. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	sv_freq	Normalized frequency of reference voltage vector.	Q15	8000−7FFF
	sv_gain	Normalized gain of the reference voltage vector.	Q15	8000−7FFF
	sv_offset	Normalized offset in the reference voltage vector	Q15	8000−7FFF
Outputs	Ta	Duty ratio of PWM1(CMPR1 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
	Tb	Duty ratio of PWM3(CMPR2 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
	Tc	Duty ratio of PWM5(CMPR3 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
Init / Config	none			

Variable Declaration:
In the system file include the following statements:

```
.ref   SVGEN_MF, SVGEN_MF _INIT                      ;function call

.ref   sv_freq, sv_gain, sv_offset, Ta, Tb, Tc    ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section ‘svgen_mf’

Example:

```
ldp  #sv_freq              ;Set DP for module input
bldd #input_var1, sv_freq  ;Pass input variables to module inputs
bldd #input_var2, sv_gain
bldd #input_var2, sv_offset

CALL SVGEN_MF

ldp  #output_var1          ;Set DP for output variable
bldd #Ta, output_var1      ;Pass module outputs to output variables
bldd #Tb, output_var2
bldd #Tc, output_var3
```

C/C-Callable ASM Interface

Object Definition

The structure of the SVGENMF object is defined by the following structure definition

```
/*-----------------------------------------------------------------------------
Define the structure of the SVGENMF
(Magnitude and angular velocity based Space Vector Waveform Generator)
-----------------------------------------------------------------------------*/

typedef struct  { int gain;      /* Waveform amplitude Q15 Input         */
                  int freq;      /* Frequency setting  Q15 Input         */
                  int freq_max;  /* Frequency setting  Q0  Input         */
                  int alpha;     /* Internal var - Sector angle history  */
                  int sector;    /* Internal var - Sector number history */
                  int va;        /* Phase A output Q15                   */
                  int vb;        /* Phase B output Q15                   */
                  int vc;        /* Phase C output Q15                   */
                  int (*calc)(); /* Pointer to calculation function      */

                } SVGENMF;
```

Table 74. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	freq	Fraction of Frequency of reference voltage vector.	Q15	8000−7FFF
	freq_max	Frequency of reference voltage vector.	Q0	8000−7FFF
	gain	Required gain for the desired reference voltage vector.	Q15	8000−7FFF
Outputs	va	Duty ratio of PWM1(CMPR2 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
	vb	Duty ratio of PWM3(CMPR2 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF
	vc	Duty ratio of PWM5(CMPR3 register value as a fraction of associated period register, TxPR, value).	Q15	8000−7FFF

Special Constants and Datatypes

SVGENMF
The module definition itself is created as a data type. This makes it convenient to instance a Space Vector Generation module. To create multiple instances of the module simply declare variables of type SVGENMF.

SVGENDQ_handle
Typedef'ed to SVGENMF *

SVGENMF_DEFAULTS
Initializer for the SVGENMF Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(SVGENMF_handle)
The default definition of the object implements just one method – the runtime compute function for the generation of the space vector modulation functions. This is implemented by means of a function pointer, and the default initializer sets this to svgenmf_calc. The argument to this function is the address of the SVGENMF object.

Module Usage

Instantiation:
The following example instances two such objects:

```
SVGENMF   sv1,sv2;
```

Initialization:
To instance a pre-initialized object

```
SVGENMF  sv1=SVGEN_DEFAULTS,sv2=SVGEN_DEFAULTS;
```

Invoking the compute function:

```
sv1.calc(&sv1);
```

Example:
Lets instance two SVGENMF objects, otherwise identical, but running with different freq values.

```
SVGENMF sv1=SVGEN_DEFAULTS; /* Instance the first object */
SVGENMF sv2=SVGEN_DEFAULTS; /* Instance the second object*/

main()
{
     sv1.freq=1200;  /* Set properties for sv1 */
     sv2.freq=1800;  /* Set properties for sv2 */
}
void interrupt periodic_interrupt_isr()
{
     sv1.calc(&sv1); /* Call compute function for sv1 */
     sv2.calc(&sv2); /* Call compute function for sv2 */

     x=sv1.va;       /* Access the outputs of sv1 */
     y=sv1.vb;
     z=sv1.vc;

     p=sv2.va;       /* Access the outputs of sv2 */
     q=sv2.vb;
     r=sv2.vc;

/* Do something with the outputs */

     }
```

Background Information

The Space Vector Pulse Width Modulation (SVPWM) refers to a special switching sequence of the upper three power devices of a three-phase voltage source inverters (VSI) used in application such as AC induction and permanent magnet synchronous motor drives. This special switching scheme for the power devices results in 3 pseudo-sinusoidal currents in the stator phases.

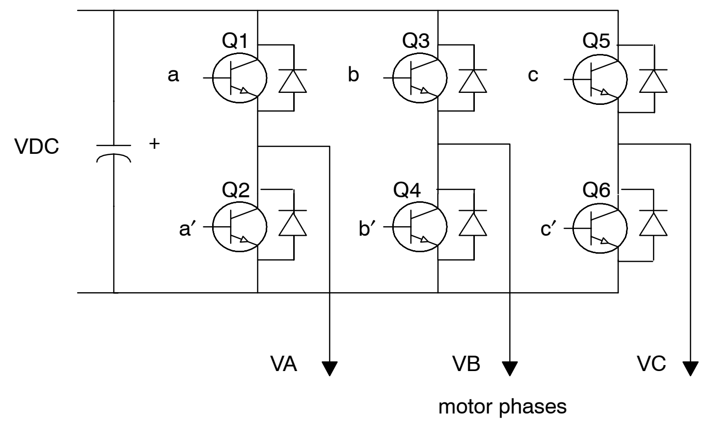


Figure 34. Power Circuit Topology for a Three-Phase VSI

It has been shown that SVPWM generates less harmonic distortion in the output voltages or currents in the windings of the motor load and provides more efficient use of DC supply voltage, in comparison to direct sinusoidal modulation technique.

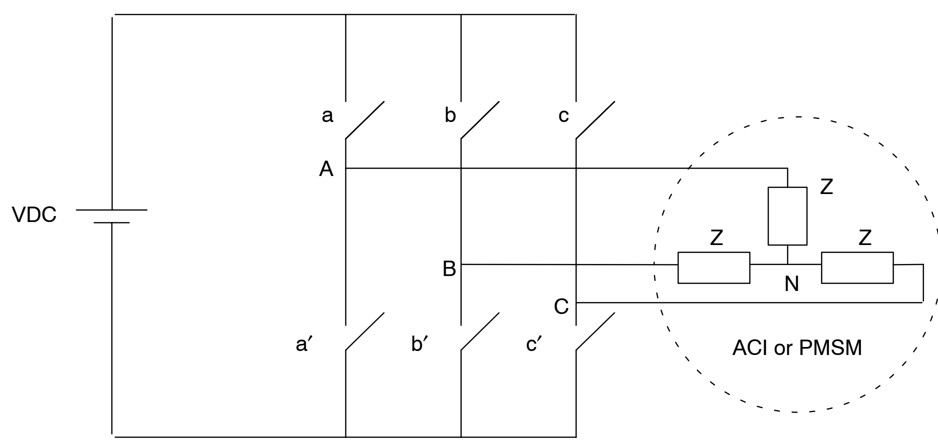


Figure 35. Power Bridge for a Three-Phase VSI

For the three phase power inverter configurations shown in Figure 34 and Figure 35, there are eight possible combinations of on and off states of the upper power transistors. These combinations and the resulting instantaneous output line-to-line and phase voltages, for a dc bus voltage of $V_{DC,}$ are shown in Table 75.

Table 75. Device On/Off Patterns and Resulting Instantaneous Voltages of a 3-Phase Power Inverter

c	b	a	V_{AN}	V_{BN}	V_{CN}	V_{AB}	V_{BC}	V_{CA}
0	0	0	0	0	0	0	0	0
0	0	1	$2V_{DC}/3$	$-V_{DC}/3$	$-V_{DC}/3$	V_{DC}	0	$-V_{DC}$
0	1	0	$-V_{DC}/3$	$2V_{DC}/3$	$-V_{DC}/3$	$-V_{DC}$	V_{DC}	0
0	1	1	$V_{DC}/3$	$V_{DC}/3$	$-2V_{DC}/3$	0	V_{DC}	$-V_{DC}$
1	0	0	$-V_{DC}/3$	$-V_{DC}/3$	$2V_{DC}/3$	0	$-V_{DC}$	V_{DC}
1	0	1	$V_{DC}/3$	$-2V_{DC}/3$	$V_{DC}/3$	V_{DC}	$-V_{DC}$	0
1	1	0	$-2V_{DC}/3$	$V_{DC}/3$	$V_{DC}/3$	$-V_{DC}$	0	V_{DC}
1	1	1	0	0	0	0	0	0

The quadrature quantities (in d−q frame) corresponding to these 3 phase voltages are given by the general Clarke transform equation:

$$V_{ds} = V_{AN}$$

$$V_{qs} = \frac{(2V_{BN} + V_{AN})}{\sqrt{3}}$$

In matrix from the above equation is also expressed as,

$$\begin{bmatrix} V_{ds} \\ V_{qs} \end{bmatrix} = \frac{2}{3}\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix}\begin{bmatrix} V_{AN} \\ V_{BN} \\ V_{CN} \end{bmatrix}$$

Due to the fact that only 8 combinations are possible for the power switches, V_{ds} and V_{qs} can also take only a finite number of values in the (d−q) frame according to the status of the transistor command signals (c,b,a). These values of V_{ds} and V_{qs} for the corresponding instantaneous values of the phase voltages (V_{AN}, $V_{BN,}$ V_{CN}) are listed in Table 76.

Table 76. Switching Patterns, Corresponding Space Vectors, and their (d−q) Components

c	b	a	V_{ds}	V_{qs}	Vector
0	0	0	0	0	O_0
0	0	1	$\frac{2V_{DC}}{3}$	0	U_0
0	1	0	$-\frac{V_{DC}}{3}$	$\frac{V_{DC}}{\sqrt{3}}$	U_{120}
0	1	1	$\frac{V_{DC}}{3}$	$\frac{V_{DC}}{\sqrt{3}}$	U_{60}
1	0	0	$-\frac{V_{DC}}{3}$	$-\frac{V_{DC}}{\sqrt{3}}$	U_{240}
1	0	1	$\frac{V_{DC}}{3}$	$-\frac{V_{DC}}{\sqrt{3}}$	U_{300}
1	1	0	$-\frac{2V_{DC}}{3}$	0	U_{180}
1	1	1	0	0	O_{111}

These values of V_{ds} and $V_{qs,}$ listed in Table 76, are called the (d−q) components of the basic space vectors corresponding to the appropriate transistor command signal (c,b,a). The space vectors corresponding to the signal (c,b,a) are listed in the last column in Table 76. For example, (c,b,a)=001 indicates that the space vector is U_0.The eight basic space vectors defined by the combination of the switches are also shown in Figure 36.

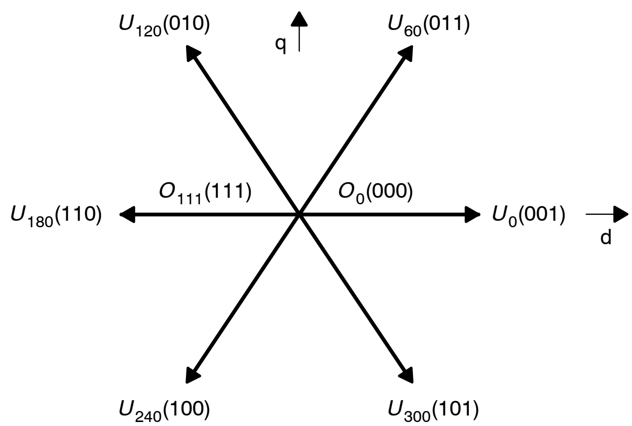


Figure 36. Basic Space Vectors

In Figure 36, vectors corresponding to states 0 (000) and 7 (111) of the switching variables are called the zero vectors.

Decomposing the reference voltage vector V*

The objective of Space Vector PWM technique is to approximate a given stator reference voltage vector V* by combination of the switching pattern corresponding to the basic space vectors. The reference voltage vector V* is obtained by mapping the desired three phase output voltages(line to neutral) in the (d−q) frame through the Clarke transform defined earlier. When the desired output phase voltages are balanced three phase sinusoidal voltages, V* becomes a vector rotating around the origin of the (d−q) plane with a frequency corresponding to that of the desired three phase voltages.

The magnitude of each basic space vector, as shown in Figure 37, is normalized by the maximum value of the phase voltages. Therefore, when the maximum bus voltage is V_{DC}, the maximum line to line voltage is also V_{DC}, and so the maximum phase voltage(line to neutral) is $V_{DC}/\sqrt{3}$. From Table 76, the magnitude of the basic space vectors is $2V_{DC}/3$. When this is normalized by the maximum phase voltage($V_{DC}/\sqrt{3}$), the magnitude of the basic space vectors becomes $2/\sqrt{3}$. These magnitudes of the basic space vectors are indicated in Figure 37.

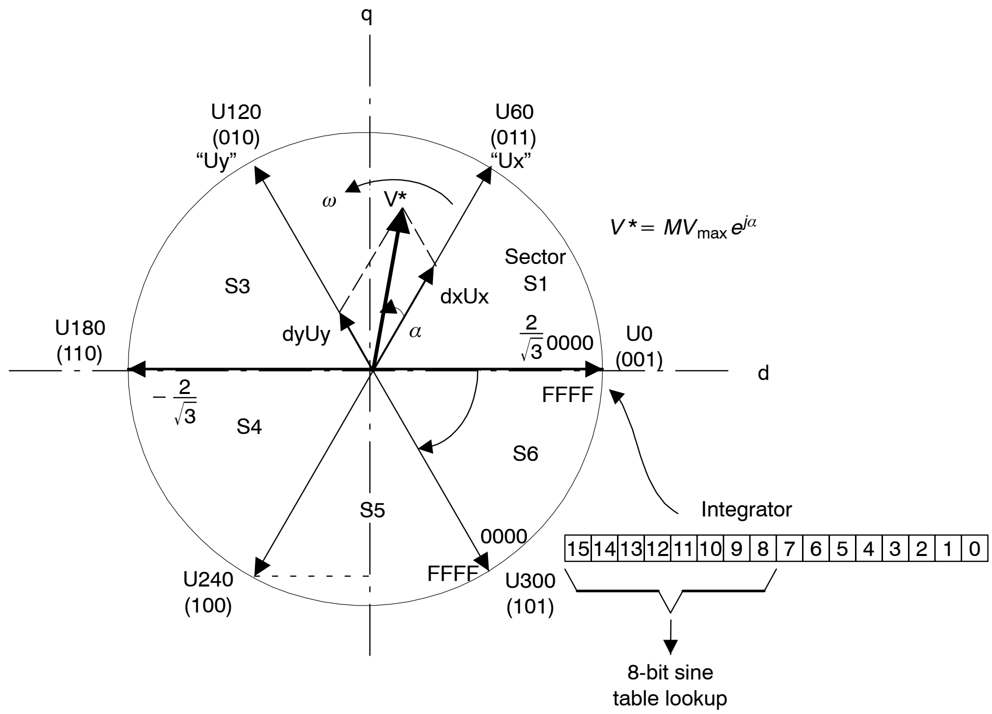


Figure 37. Projection of the Reference Voltage Vector

Representing the reference vector V* with the basic space vectors requires precise control of both the vector magnitude M (also called the modulation index) and the angle α. The aim here is to rotate V* in the d−q plane at a given angular speed (frequency) ω. The vector magnitude M controls the resultant peak phase voltage generated by the inverter.

In order to generate the reference vector V*, a time average of the associated basic space vectors is required, i.e. the desired voltage vector V* located in a given sector, can be synthesized as a linear combination of the two adjacent space vectors, Ux and Uy which frame the sector, and either one of the two zero vectors. Therefore,

$$V* = dxUx + dyUy + dzUz$$

where Uz is the zero vector, and dx, dy and dz are the duty ratios of the states X, Y and Z within the PWM switching interval. The duty ratios must add to 100% of the PWM period, i.e: dx + dy + dz = 1.

Vector V* in Figure 37 can also be written as:

$$V* = MV_{\max}\, e^{j\alpha} = dxUx + dyUy + dzUz$$

where M is the modulation index and V_{max} is the maximum value of the desired phase voltage.

By projecting V* along the two adjacent space vectors Ux and Uy, we have,

$$\begin{cases} MV_{\max}\cos\alpha = dx|Ux| + dy|Uy|\cos 60^\circ \\ MV_{\max}\sin\alpha = dy|Uy|\sin 60^\circ \end{cases}$$

Since the voltages are normalized by the maximum phase voltage, V_{max}=1. Then by knowing |Ux| = |Uy| = 2/√3 (when normalized by maximum phase voltage), the duty ratios can be derived as,

$$dx = M\sin(60 - \alpha)$$

$$dy = M\sin(\alpha)$$

These same equations apply to any sector, since the d−q reference frame, which has here no specific orientation in the physical space, can be aligned with any space vector.

Implementation of sin function

In this implementation the angular speed ω is controlled by a precision frequency generation algorithm which relies on the modulo nature (i.e. wrap-around) of a finite length register, called Integrator in Figure 37. The upper 8 bits of this integrator (a data memory location in 24x/24xx) is used as a pointer to a 256 word Sine lookup table. By adding a fixed value (step size) to this register, causes the 8 bit pointer to cycle at a constant rate through the Sine table. In effect we are integrating angular velocity to give angular position. At the end limit the pointer simply wraps around and continues at the next modulo value given by the step size. The rate of cycling through the table is very easily and accurately controlled by the value of step size.

As shown in Figure 37, sine of α is needed to decompose the reference voltage vector onto the basic space vectors of the sector the voltage vector is in. Since this decomposition is identical among the six sectors, only a 60° sine lookup table is needed. In order to complete one revolution (360°) the sine table must be cycled through 6 times.

For a given step size the angular frequency (in cycles/sec) of V* is given by:

$$\omega = \frac{STEP \times f_s}{6 \times 2^m}$$

where

f_s = sampling frequency (i.e. PWM frequency)

STEP = angle stepping increment

m = # bits in the integration register.

For example, if f_s = 24KHz, m=16 bits & STEP ranges from 0à2048 then the resulting angular frequencies will be as shown in Table 77.

Table 77. Frequency Mapping

STEP	Freq (Hz)	STEP	Freq (Hz)	STEP	Freq (Hz)
1	0.061	600	36.62	1700	103.76
20	1.22	700	42.72	1800	109.86
40	2.44	800	48.83	1900	115.97
60	3.66	900	54.93	2000	122.07
80	4.88	1000	61.04	2100	128.17
100	6.10	1100	67.14	2200	134.28

From the table it is clear that a STEP value of 1 gives a frequency of 0.061Hz, this defines the frequency setting resolution, i.e. the actual line voltage frequency delivered to the AC motor can be controlled to better than 0.1 Hz.

For a given f_s the frequency setting resolution is determined by m the number of bits in the integration register. Table 78 shows the theoretical resolution which results from various sizes of m.

Table 78. Resolution of Frequency Mapping

m (# bits)	Freq res (Hz)	m (# bits)	Freq res (Hz)
8	15.6250	17	0.0305
12	0.9766	18	0.0153
14	0.2441	19	0.0076
16	0.0610	20	0.0038

Another important parameter is the size of the lookup table. This directly effects the harmonic distortion produced in the resulting synthesized sine wave. As mentioned previously a 256 entry sine table is used which has a range of 60°. This gives an angle lookup resolution of 60° / 256 = 0.23°. The table entries are given in Q15 format and a summarized version is shown below.

```
;--------------------------------------------------------
;No. Samples: 256,  Angle Range: 60, Format: Q15
;--------------------------------------------------------
;             SINVAL ;   Index  Angle    Sin(Angle)
;--------------------------------------------------------
STABLE  .word   0     ;  0    0        0.00
        .word   134   ;  1    0.23     0.00
        .word   268   ;  2    0.47     0.01
        .word   402   ;  3    0.70     0.01
        .word   536   ;  4    0.94     0.02
        .word   670   ;  5    1.17     0.02
          "         "        "        "     "
          "         "        "        "     "
          "         "        "        "     "
        .word   28106 ;  252  59.06    0.86
        .word   28175 ;  253  59.30    0.86
        .word   28243 ;  254  59.53    0.86
        .word   28311 ;  255  59.77    0.86
```

Realization of the PWM Switching Pattern

Once the PWM duty ratios dx, dy and dz are calculated, the appropriate compare values for the compare registers in 24x/24xx can be determined. The switching pattern in Figure 38 is adopted here and is implemented with the Full Compare Units of 24x/24xx. A set of 3 new compare values, Ta, Tb and Tc, need to be calculated every PWM period to generate this switching pattern.

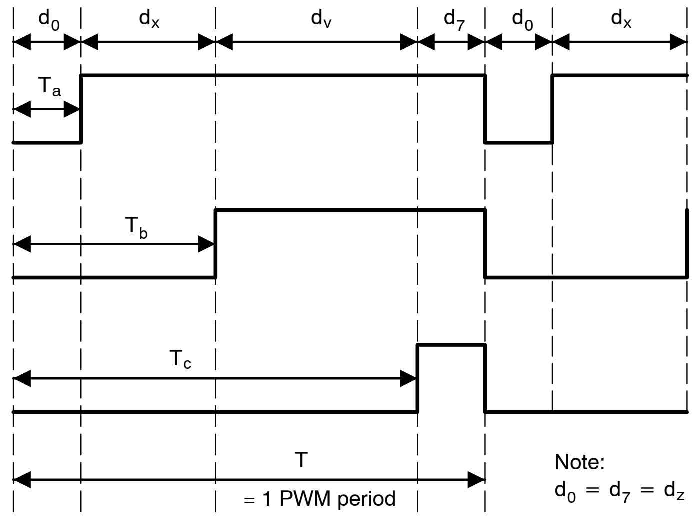


Figure 38. PWM Output Switching Pattern

From Figure 38, it can be seen:

$$Ta = \frac{(T - dx - dy)}{2}$$

$$Tb = dx + Ta$$

$$Tc = T - Ta$$

If we define an intermediate variable T1 using the following equation:

$$T1 = \frac{T - dx - dy}{2}$$

Then for different sectors Ta, Tb and Tc can be expressed in terms of T1. Table 79 depicts this determination.

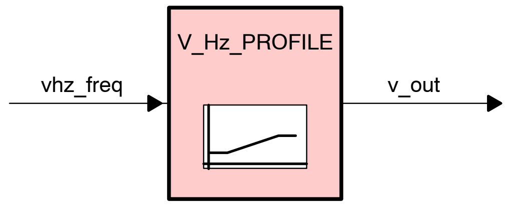
Table 79. Calculation of Duty Cycle for Different Sectors

Sector	U_0, U_{60}	U_{60}, U_{120}	U_{120}, U_{180}	U_{180}, U_{240}	U_{240}, U_{300}	U_{300}, U_0
Ta	T1	dy+Tb	T−Tb	T−Tc	dx+Tc	T1
Tb	dx+Ta	T1	T1	dy+Tc	T−Tc	T−Ta
Tc	T−Ta	T−Tb	dx+Tb	T1	T1	dy+Ta

The switching pattern shown in Figure 38 is an asymmetric PWM implementation. However, 24x/24xx devices can also generate symmetric PWM. Little change to the above implementation is needed to accommodate for this change. The choice between the symmetrical and asymmetrical case depends on the other care-about in the final implementation.

Description

This module generates an output command voltage for a specific input command frequency according to the specified volts/hertz profile. This is used for variable speed implementation of AC induction motor drives.



Availability

This module is available in two interface formats:

1) The direct-mode assembly-only interface (Direct ASM)

2) The C-callable interface version.

Module Properties

Type: Target Independent/Application Dependent

Target Devices: x24x/x24xx

Direct ASM Version File Name: vhz_prof.asm

C-Callable Version File Names: vhzprof.asm, vhzprof.h

Item	ASM Only	C-Callable ASM	Comments
Code size	42 words	48 words†	
Data RAM	9 words	0 words†	
xDAIS module	No	Yes	
xDAIS component	No	No	IALG layer not implemented

† Each pre-initialized VHZPROFILE struction consumes 10 words in the .cinit section instance and 8 words in data memory.

Direct ASM Interface

Table 80. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	vhz_freq	Command frequency of the stator voltage	Q15	0–7FFF
Outputs	v_out	Command stator output voltage	Q15	0–7FFF
Init / Config	FL†	Low frequency point on v/f profile.	Q15	Application dependent
	FH†	High frequency point on v/f profile.	Q15	Application dependent
	Fmax†	Maximum frequency	Q15	Application dependent
	vf_slope†	Slope of the v/f profile	Q12	Application dependent
	Vmax†	Voltage corresponding to FH	Q15	Application dependent
	Vmin†	Voltage corresponding to FL	Q15	Application dependent

† These parameters are initialized to some default values in the module initialization routine. Initialize these from the system file if the default values are not used.

Variable Declaration:
In the system file include the following statements:

```
.ref   V_Hz_PROFILE, V_Hz_PROFILE _INIT    ;function call

.ref   vhz_freq, v_out                     ;input/output
```

Memory map:
All variables are mapped to an uninitialized named section ‘vhz_prof’

Example:

```
ldp  #vhz_freq               ;Set DP for module input
bldd #input_var1, vhz_freq   ;Pass input variable to module
                             ;input

CALL V_Hz_PROFILE

ldp  #output_var1            ;Set DP for output variable
bldd #v_out, output_var1     ;Pass module output to output
                             ; variable
```

C/C-Callable ASM Interface

Object Definition

The object is defined as

```
typedef struct { int freq;   /* Frequency input Q15 */
                 int fl;     /* Freq below which vout=vmin:Q15 Input  */
                 int fh;     /* Freq above which vout=vmax Q15 Input  */
                 int slope;  /* Slope of the Vhz profile:  Q15 Input  */
                 int vmax;   /* Voltage output above fmax  Q15 Input  */
                 int vmin;   /* Voltage output below fmin  Q15 Input  */
                 int vout;   /* Computed output voltage    Q15 Output */
                 int (*calc)();  /* Ptr to the calculation function   */
               } VHZPROFILE;
```

Table 81. Module Terminal Variables/Functions

	Name	Description	Format	Range
Inputs	freq	Command frequency of the stator voltage	Q15	0−7FFF
Outputs	vout	Command stator output voltage	Q15	0−7FFF
Init / Config	fl†	Low frequency point on v/f profile.	Q15	Application dependent
	fh†	High frequency point on v/f profile.	Q15	Application dependent
	slope†	Slope of the v/f profile	Q12	Application dependent
	vmax†	Voltage corresponding to fl	Q15	Application dependent
	vmin†	Voltage corresponding to fh	Q15	Application dependent

† These parameters are initialized to some default values in the module initialization routine. Initialize these from the system file if the default values are not used.

Special Constants and Datatypes

VHZPROFILE
The module definition itself is created as a data type. This makes it convenient to instance a VHZ Profile module. To create multiple instances of the module simply declare variables of type VHZPROFILE.

DEFAULT_PROFILE
Initializer for the SVGENMF Object. This provides the initial values to the terminal variables, internal variables, as well as method pointers.

Methods

void calc(VHZPROFILE *)
The only method implemented for this object is the runtime compute function for the calculation of the vout value depending on the object parameters. The argument to this function is the address of the VHZPROFILE object.

Module Usage

Instantiation:
The following example instances two such objects:

```
VHZPROFILE   vhz1,vhz2;
```

Initialization:

To instance a pre-initialized object

```
VHZPROFILE   vhz1=DEFAULT_PROFILE;
```

Invoking the compute function:

```
vhz1.calc(&vhz1);
```

Example:

Lets instance two SVGENMF objects, otherwise identical, but running with different freq values. These SVGENMF objects need the computed value of the envelope for the SVGEN waveforms, and this is computed by the VHZPROFILE objects.

```
  SVGENMF sv1=SVGEN_DEFAULTS;  /* Instance the first object */
  SVGENMF sv2=SVGEN_DEFAULTS;  /* Instance the second object*/

  VHZPROFILE vhz1=DEFAULT_PROFILE;
  VHZPROFILE vhz2=DEFAULT_PROFILE;

main()
{
  sv1.freq=1200;               /* Set properties for sv1 */
  sv2.freq=1800;               /* Set properties for sv2 */
}

void interrupt periodic_interrupt_isr()
{
  vhz1.freq=sv1.freq;          /* Connect the sv1, sv2 freq to vhz1 and vhz2 */
  vhz1.freq=sv1.freq;

  vhz2.calc(&vhz1);            /* Call the compute functions */
  vhz2.calc(&vhz1);

  sv1.gain=vhz1.gain;          /* Pass the computed output voltages back to the svgens */

  sv2.gain=vhz2.gain;

  sv1.calc(&sv1);              /* Call compute function for sv1 */
  sv2.calc(&sv2);              /* Call compute function for sv2 */

  x=sv1.va;                    /* Access the outputs of sv1 */
  y=sv1.vb;
  z=sv1.vc;

  p=sv2.va;                    /* Access the outputs of sv2 */
  q=sv2.vb;
  r=sv2.vc;

/* Do something with the outputs. Something is probably modulate PWMs to drive motors with.
*/

  }
```

Background Information

If the voltage applied to a three phase AC Induction motor is sinusoidal, then by neglecting the small voltage drop across the stator resistor, we have, at steady state,

$$\hat{V} \approx j\omega \hat{\Lambda}$$

i.e.,

$$V \approx \omega \Lambda$$

where $\hat{V}$ and $\hat{\Lambda}$ are the phasor representations of stator voltage and stator flux, and V and Λ are their magnitude, respectively. Thus, we get

$$\Lambda = \frac{V}{\omega} = \frac{1}{2\pi}\frac{V}{f}$$

From the last equation, it follows that if the ratio *V/f* remains constant for any change in *f*, then flux remains constant and the torque becomes independent of the supply frequency. In actual implementation, the ratio of the magnitude to frequency is usually based on the rated values of these parameters, i.e., the motor rated parameters. However, when the frequency, and hence the voltage, is low, the voltage drop across the stator resistor cannot be neglected and must be compensated for. At frequencies higher than the rated value, maintaining constant V/Hz means exceeding rated stator voltage and thereby causing the possibility of insulation break down. To avoid this, constant V/Hz principle is also violated at such frequencies. This principle is illustrated in Figure 39.

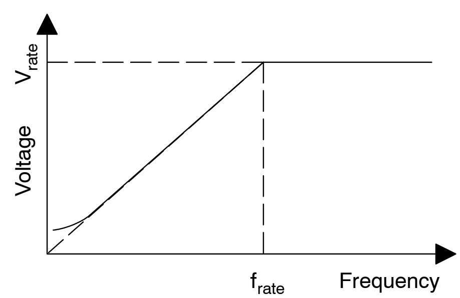


Figure 39. Voltage Versus Frequency Under the Constant V/Hz Principle

Since the stator flux is maintained constant (independent of the change in supply frequency), the torque developed depends only on the slip speed. This is shown in Figure 40. So by regulating the slip speed, the torque and speed of an AC Induction motor can be controlled with the constant V/Hz principle.

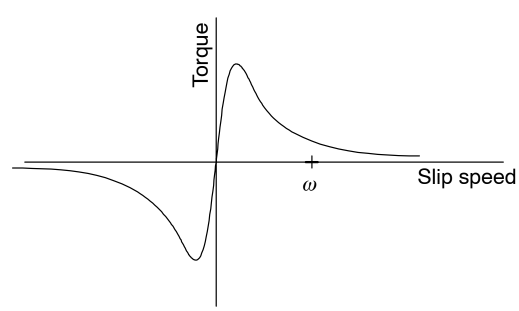


Figure 40. Toque Versus Slip Speed of an Induction Motor With Constant Stator Flux

Both open and closed-loop control of the speed of an AC induction motor can be implemented based on the constant V/Hz principle. Open-loop speed control is used when accuracy in speed response is not a concern such as in HVAC (heating, ventilation and air conditioning), fan or blower applications. In this case, the supply frequency is determined based on the desired speed and the assumption that the motor will roughly follow its synchronous speed. The error in speed resulted from slip of the motor is considered acceptable.

In this implementation, the profile in Figure 39 is modified by imposing a lower limit on frequency. This is shown in Figure 41. This approach is acceptable to applications such as fan and blower drives where the speed response at low end is not critical. Since the rated voltage, which is also the maximum voltage, is applied to the motor at rated frequency, only the rated minimum and maximum frequency information is needed to implement the profile.

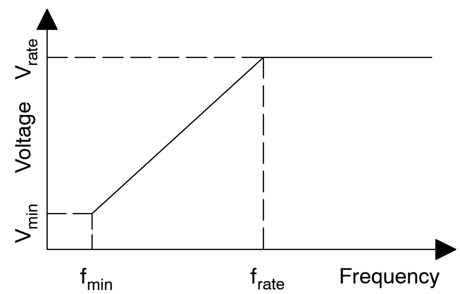


Figure 41. Modified V/Hz Profile

The command frequency is allowed to go below the minimum frequency, $f_{min,}$ with the output voltage saturating at a minimum value, V_{min}. Also, when the command frequency is higher than the maximum frequency, f_{max}, the output voltage is saturated at a maximum value, V_{max}.