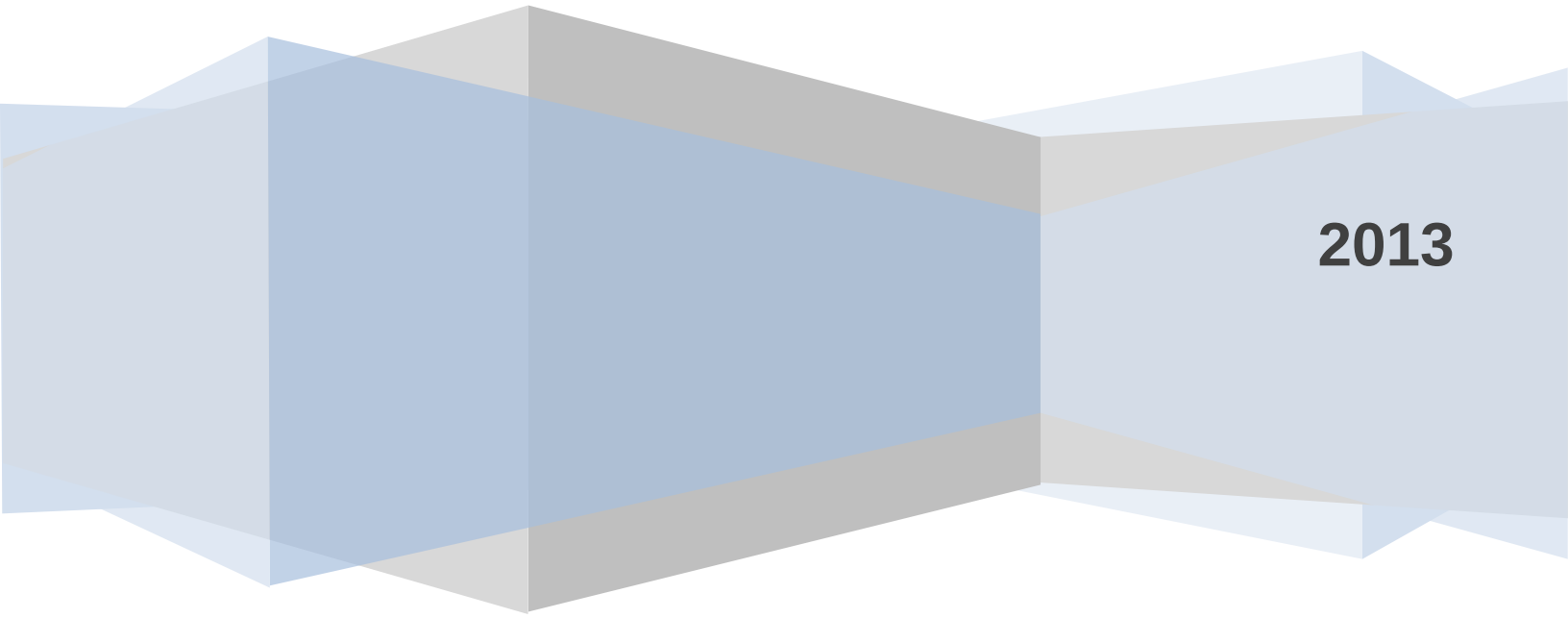
Texas Instruments, Inc.
C2000 Systems and Applications

Digital Motor Control

Software Library:
Target Independent Math Blocks



2013

v13.1

Contents

INTRODUCTION 3
ACI_SE 4
ACI_FE 16
CLARKE 31
COMTN_TRIG 35
CUR_MOD 43
IPARK 50
IMPULSE 54
MOD6_CNT 58
PARK 62
PHASE_VOLT_CALC 66
PI 72
PI_REG4 78
PI_POS 81
PI_POS_REG4 83
PID 85
RAMPGEN 94
RMP_CNTL 98
RMP2_CNTL 102
RMP3_CNTL 106
RESOLVER 110
SMO 114
SPEED_EST 125
SPEED_FR 130
SPEED_PRD 135
SVGEN 142
SVGEN_COMM 145
SVGEN_DPWM 148
SVGEN_MF 151
VHZ_PROFILE 162

Introduction

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

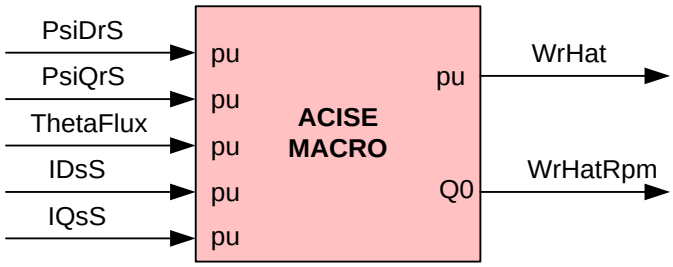
Transformation and Observer Modules	Clarke, Park, Phase Voltage Calculation, Sliding Mode Observer, BEMF Commutation, Direct Flux Estimator, Speed Calculators and Estimators, Position Calculators and Estimators etc.
Signal Generators and Control Modules	PID, Commutation Trigger Generator, V/f Controller,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

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

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

Description

This software module implements a speed estimator of the 3-ph induction motor based upon its mathematics model. The estimator's accuracy relies heavily on knowledge of critical motor parameters.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: aci_se.h

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

C Interface

Object Definition

The structure of ACISE object is defined by following structure definition

```
typedef struct {  _iq  IQsS;          // Input: Stationary q-axis stator current
                  _iq  PsiDrS;        // Input: Stationary d-axis rotor flux
                  _iq  IDsS;          // Input: Stationary d-axis stator current
                  _iq  PsiQrS;        // Input: Stationary q-axis rotor flux
                  _iq  K1;            // Parameter: Constant using in speed computation
                 _iq  SquaredPsi;     // Variable: Squared rotor flux
                 _iq  ThetaFlux;      // Input: Rotor flux angle
                 _iq21  K2;           // Parameter: Constant using in differentiator (Q21)
                 _iq  OldThetaFlux;   // Variable: Previous rotor flux angle
                 _iq  K3;             // Parameter: Constant using in low-pass filter
                 _iq21  WPsi;         // Variable: Synchronous rotor flux speed in pu  (Q21)
                 _iq  K4;             // Parameter: Constant using in low-pass filter
                 _iq  WrHat;          // Output: Estimated speed in per unit
                 Uint32  BaseRpm;     // Parameter: Base rpm speed (Q0)
                 int32  WrHatRpm;     // Output: Estimated speed in rpm (Q0)
                 _iq Wslip;           // Variable: Slip
                 _iq Wsyn;            // Variable: Synchronous speed
               } ACISE;
```

Module Terminal Variables

Item	Name	Description	Format*	Range(Hex)
Inputs	PsiDrS	stationary d-axis rotor flux	GLOBAL_Q	80000000-7FFFFFFF
	PsiDrS	stationary q-axis rotor flux	GLOBAL_Q	80000000-7FFFFFFF
	ThetaFlux	rotor flux linkage angle	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)
	IDsS	stationary d-axis stator current	GLOBAL_Q	80000000-7FFFFFFF
	IQsS	stationary q-axis stator current	GLOBAL_Q	80000000-7FFFFFFF
Outputs	WrHat	estimated rotor speed	GLOBAL_Q	80000000-7FFFFFFF
	WrHatRpm	estimated rotor speed in rpm	Q0	80000000-7FFFFFFF
ACISE parameter	K1	K1 = 1/(Wb*Tr)	GLOBAL_Q	80000000-7FFFFFFF
	K2	K2 = 1/(fb*T)	Q21	80000000-7FFFFFFF
	K3	K3 = Tau/(Tau+T)	GLOBAL_Q	80000000-7FFFFFFF
	K4	K4 = T/(Tau+T)	GLOBAL_Q	80000000-7FFFFFFF
	BaseRpm	base speed in rpm	Q0	80000000-7FFFFFFF
Internal	OldThetaFlux	previous rotor flux linkage angle	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)
	WPsi	synchronous rotor flux speed	GLOBAL_Q	80000000-7FFFFFFF
	SquaredPsi	squared magnitude of rotor flux	GLOBAL_Q	80000000-7FFFFFFF
	Wslip	slip	GLOBAL_Q	80000000-7FFFFFFF
	WSyn	synchronous speed	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

ACISE
The module definition is created as a data type. This makes it convenient to instance an interface to the speed estimator of Induction Motor module. To create multiple instances of the module simply declare variables of type ACISE.

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

Module Usage

Instantiation
The following example instances two ACISE objects
ACISE se1, se2;

Initialization
To Instance pre-initialized objects
ACISE se1 = ACISE_DEFAULTS;
ACISE se2 = ACISE_DEFAULTS;

Invoking the computation macro
ACISE_MACRO(se1);
ACISE_MACRO(se2);

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

```
main()
{
        se1.K1 = parem1_1;              // Pass parameters to se1
        se1.K2 = parem1_2;              // Pass parameters to se1
        se1.K3 = parem1_3;              // Pass parameters to se1
        se1.K4 = parem1_4;              // Pass parameters to se1
        se1.BaseRpm = base_speed_1;     // Pass parameters to se1

        se2.K1 = parem2_1;              // Pass parameters to se2
        se2.K2 = parem2_2;              // Pass parameters to se2
        se2.K3 = parem2_3;              // Pass parameters to se2
        se2.K4 = parem2_4;              // Pass parameters to se2
        se2.BaseRpm = base_speed_2;     // Pass parameters to se2

}
```

```
void interrupt periodic_interrupt_isr()
{
        se1.PsiDrS= flux_dq1.d;              // Pass inputs to se1
        se1.PsiQrS= flux_dq1.q;              // Pass inputs to se1
        se1.IDsS=current_dq1.d;              // Pass inputs to se1
        se1.IQsS=current_dq1.q;              // Pass inputs to se1
        se1.ThetaFlux=angle1;                // Pass inputs to se1

        se2.PsiDrS= flux_dq2.d;              // Pass inputs to se2
        se2.PsiQrS= flux_dq2.q;              // Pass inputs to se2
        se2.IDsS=current_dq2.d;              // Pass inputs to se2
        se2.IQsS=current_dq2.q;              // Pass inputs to se2
        se2.ThetaFlux=angle2;                // Pass inputs to se2

        ACISE_MACRO(se1);                   // Call compute macro for se1
        ACISE_MACRO(se2);                    // Call compute macro for se2

        speed_pu1 = se1.WrHat;               // Access the outputs of se1
        speed_rpm1 = se1.WrHatRpm;           // Access the outputs of se1

        speed_pu2 = se2.WrHat;               // Access the outputs of se2
        speed_rpm2 = se2.WrHatRpm;           // Access the outputs of se2
}
```

Constant Computation Macro

Since the speed estimator of Induction motor module requires four constants (K1,…, K4) to be input basing on the machine parameters, base quantities, mechanical parameters, and sampling period. These four constants can be internally computed by the macro (aci_se_const.h). The followings show how to use the constant computation macro.

Object Definition

The structure of ACISE_CONST object is defined by following structure definition

```
typedef struct  { float32  Rr;     // Input: Rotor resistance (ohm)
                  float32  Lr;     // Input: Rotor inductance (H)
                  float32  fb;     // Input: Base electrical frequency (Hz)
                  float32  fc;     // Input: Cut-off frequency of low-pass filter (Hz)
                  float32  Ts;     // Input: Sampling period in sec
                  float32  K1;     // Output: constant using in rotor flux calculation
                  float32  K2;     // Output: constant using in rotor flux calculation
                  float32  K3;     // Output: constant using in rotor flux calculation
                  float32  K4;     // Output: constant using in stator current calculation
                } ACISE_CONST;
```

Module Terminal Variables

Item	Name	Description	Format	Range(Hex)
Inputs	Rr	Rotor resistance (ohm)	Floating	N/A
	Lr	Rotor inductance (H)	Floating	N/A
	fb	Base electrical frequency (Hz)	Floating	N/A
	fc	Cut-off frequency of low-pass filter (Hz)	Floating	N/A
	Ts	Sampling period (sec)	Floating	N/A
Outputs	K1	constant using in rotor flux calculation	Floating	N/A
	K2	constant using in rotor flux calculation	Floating	N/A
	K3	constant using in rotor flux calculation	Floating	N/A
	K4	constant using in stator current cal.	Floating	N/A

Special Constants and Data types

ACISE_CONST
The module definition is created as a data type. This makes it convenient to instance an interface to the speed estimation of Induction Motor constant computation module. To create multiple instances of the module simply declare variables of type ACISE_CONST.

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

Module Usage

Instantiation
The following example instances two ACISE_CONST objects
ACISE_CONST se1_const, se2_const;

Initialization
To Instance pre-initialized objects
ACISE_CONST se1_const = ACISE_CONST_DEFAULTS;
ACISE_CONST se2_const = ACISE_CONST_DEFAULTS;

Invoking the computation macro
ACISE_CONST_MACRO(se1_const);
ACISE_CONST_MACRO (se2_const);

Example

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

```
main()
{

        se1_const.Rr = Rr1;             // Pass floating-point inputs to se1_const
        se1_const.Lr = Lr1;             // Pass floating-point inputs to se1_const
        se1_const.fb = Fb1;             // Pass floating-point inputs to se1_const
        se1_const.fc = Fc1;             // Pass floating-point inputs to se1_const
        se1_const.Ts = Ts1;             // Pass floating-point inputs to se1_const

        se2_const.Rr = Rr2;             // Pass floating-point inputs to se2_const
        se2_const.Lr = Lr2;             // Pass floating-point inputs to se2_const
        se2_const.fb = Fb2;             // Pass floating-point inputs to se2_const
        se2_const.fc = Fc2;             // Pass floating-point inputs to se2_const
        se2_const.Ts = Ts2;             // Pass floating-point inputs to se2_const

        ACISE_CONST_MACRO (se1_const);  // Call compute macro for se1_const
        ACISE_CONST_MACRO (se2_const);  // Call compute macro for se2_const

        se1.K1 = _IQ(se1_const.K1);  // Access the floating-point outputs of se1_const
        se1.K2 = _IQ(se1_const.K2);  // Access the floating-point outputs of se1_const
        se1.K3 = _IQ(se1_const.K3);  // Access the floating-point outputs of se1_const
        se1.K4 = _IQ(se1_const.K4);  // Access the floating-point outputs of se1_const

        se2.K1 = _IQ(se2_const.K1);  // Access the floating-point outputs of se2_const
        se2.K2 = _IQ(se2_const.K2);  // Access the floating-point outputs of se2_const
        se2.K3 = _IQ(se2_const.K3);  // Access the floating-point outputs of se2_const
        se2.K4 = _IQ(se2_const.K4);  // Access the floating-point outputs of se2_const

}
```

Technical Background

The open-loop speed estimator [1] is derived basing on the mathematics equations of induction motor in the stationary reference frame. The precise values of machine parameters are unavoidably required, otherwise the steady-state speed error may happen. However, the structure of the estimator is much simple comparing with other advanced techniques. All equations represented here are in the stationary reference frame (with superscript "s"). Firstly, the rotor flux linkage equations can be shown as below:

$$\lambda_{dr}^{s} = L_r i_{dr}^{s} + L_m i_{ds}^{s} \quad (1)$$

$$\lambda_{qr}^{s} = L_r i_{qr}^{s} + L_m i_{qs}^{s} \quad (2)$$

where L_r, and L_m are rotor, and magnetizing inductance (H), respectively.
According to equations (1)-(2), the rotor currents can be expressed as

$$i_{dr}^{s} = \frac{1}{L_r}\left(\lambda_{dr}^{s} - L_m i_{ds}^{s}\right) \quad (3)$$

$$i_{qr}^{s} = \frac{1}{L_r}\left(\lambda_{qr}^{s} - L_m i_{qs}^{s}\right) \quad (4)$$

Secondly, the rotor voltage equations are used to find the rotor flux linkage dynamics.

$$0 = R_r i_{dr}^{s} + \omega_r \lambda_{qr}^{s} + \frac{d\lambda_{dr}^{s}}{dt} \quad (5)$$

$$0 = R_r i_{qr}^{s} - \omega_r \lambda_{dr}^{s} + \frac{d\lambda_{qr}^{s}}{dt} \quad (6)$$

where ω_r is electrically angular velocity of rotor (rad/sec), and R_r is rotor resistance (Ω).
Substituting the rotor currents from (3)-(4) into (5)-(6), then the rotor flux linkage dynamics can be found as

$$\frac{d\lambda_{dr}^{s}}{dt} = -\frac{1}{\tau_r}\lambda_{dr}^{s} + \frac{L_m}{\tau_r} i_{ds}^{s} - \omega_r \lambda_{qr}^{s} \quad (7)$$

$$\frac{d\lambda_{qr}^{s}}{dt} = -\frac{1}{\tau_r}\lambda_{qr}^{s} + \frac{L_m}{\tau_r} i_{qs}^{s} + \omega_r \lambda_{dr}^{s} \quad (8)$$

where $\tau_r = \frac{L_r}{R_r}$ is rotor time constant (sec).

Suppose that the rotor flux linkages in (7)-(8) are known, therefore, its magnitude and angle can be computed as

$$\lambda_r^{s} = \sqrt{\left(\lambda_{dr}^{s}\right)^2 + \left(\lambda_{qr}^{s}\right)^2} \quad (9)$$

$$\theta_{\lambda_r} = \tan^{-1}\left(\frac{\lambda_{qr}^{s}}{\lambda_{dr}^{s}}\right) \quad (10)$$

Next, the rotor flux (i.e., synchronous) speed, ω_e, can be easily calculated by derivative of the rotor flux angle in (10).

$$\omega_e = \frac{d\theta_{\lambda_r}}{dt} = \frac{d\left(\tan^{-1}\left(\frac{\lambda^s_{qr}}{\lambda^s_{dr}}\right)\right)}{dt} \quad (11)$$

Referring to the derivative table, equation (11) can be solved as

$$\frac{d\left(\tan^{-1} u\right)}{dt} = \frac{1}{1+u^2}\frac{du}{dt} \quad (12)$$

where $u = \frac{\lambda^s_{qr}}{\lambda^s_{dr}}$, yields

$$\omega_e = \frac{d\theta_{\lambda_r}}{dt} = \frac{\left(\lambda^s_{dr}\right)^2}{\left(\lambda^s_r\right)^2}\left(\frac{\lambda^s_{dr}\frac{d\lambda^s_{qr}}{dt} - \lambda^s_{qr}\frac{d\lambda^s_{dr}}{dt}}{\left(\lambda^s_{dr}\right)^2}\right) \quad (13)$$

Substituting (7)-(8) into (13), and rearranging, then finally it gives

$$\omega_e = \frac{d\theta_{\lambda_r}}{dt} = \omega_r + \frac{1}{\left(\lambda^s_r\right)^2}\frac{L_m}{\tau_r}\left(\lambda^s_{dr} i^s_{qs} - \lambda^s_{qr} i^s_{ds}\right) \quad (14)$$

The second term of the left hand in (14) is known as slip that is proportional to the electromagnetic torque when the rotor flux magnitude is maintaining constant. The electromagnetic torque can be shown here for convenience.

$$T_e = \frac{3}{2}\frac{p}{2}\frac{L_m}{L_r}\left(\lambda^s_{dr} i^s_{qs} - \lambda^s_{qr} i^s_{ds}\right) \quad (15)$$

where p is the number of poles. Thus, the rotor speed can be found as

$$\omega_r = \omega_e - \frac{1}{\left(\lambda^s_r\right)^2}\frac{L_m}{\tau_r}\left(\lambda^s_{dr} i^s_{qs} - \lambda^s_{qr} i^s_{ds}\right) \quad (16)$$

Now, the per-unit concept is applied to (16), then, the equation (16) becomes

$$\omega_{r,pu} = \omega_{e,pu} - \frac{1}{\omega_b \tau_r}\left(\frac{\lambda^s_{dr,pu} i^s_{qs,pu} - \lambda^s_{qr,pu} i^s_{ds,pu}}{\left(\lambda^s_{r,pu}\right)^2}\right) \quad \text{pu} \quad (17)$$

where $\omega_b = 2\pi f_b$ is the base electrically angular velocity (rad/sec), $\lambda_b = L_m I_b$ is the base flux linkage (volt.sec), and I_b is the base current (amp). Equivalently, another form is

$$\omega_{r,pu} = \omega_{e,pu} - K_1\left(\frac{\lambda^s_{dr,pu} i^s_{qs,pu} - \lambda^s_{qr,pu} i^s_{ds,pu}}{\left(\lambda^s_{r,pu}\right)^2}\right) \quad \text{pu} \quad (18)$$

where $K_1 = \frac{1}{\omega_b \tau_r}$.

The per-unit synchronous speed can be calculated as

$$\omega_{e,pu} = \frac{1}{2\pi f_b}\frac{d\theta_{\lambda_r}}{dt} = \frac{1}{f_b}\frac{d\theta_{\lambda_r,pu}}{dt} \quad \text{pu} \qquad (19)$$

where f_b is the base electrical (supplied) frequency (Hz) and 2π is the base angle (rad).

Discretizing equation (19) by using the backward approximation, yields

$$\omega_{e,pu}(k) = \frac{1}{f_b}\left(\frac{\theta_{\lambda_r,pu}(k) - \theta_{\lambda_r,pu}(k-1)}{T}\right) \quad \text{pu} \qquad (20)$$

where T is the sampling period (sec). Equivalently, another form is

$$\omega_{e,pu}(k) = K_2\left(\theta_{\lambda_r,pu}(k) - \theta_{\lambda_r,pu}(k-1)\right) \quad \text{pu} \qquad (21)$$

where $K_2 = \frac{1}{f_b T}$ is usually a large number.

In practice, the typical waveforms of the rotor flux angle, $\theta_{\lambda_r,pu}$, in both directions can be seen in Figure 1. To take care the discontinuity of angle from 360° to 0° (CCW) or from 0° to 360° (CW), the differentiator is simply operated only within the differentiable range as seen in this Figure. This differentiable range does not significantly lose the information to compute the estimated speed.

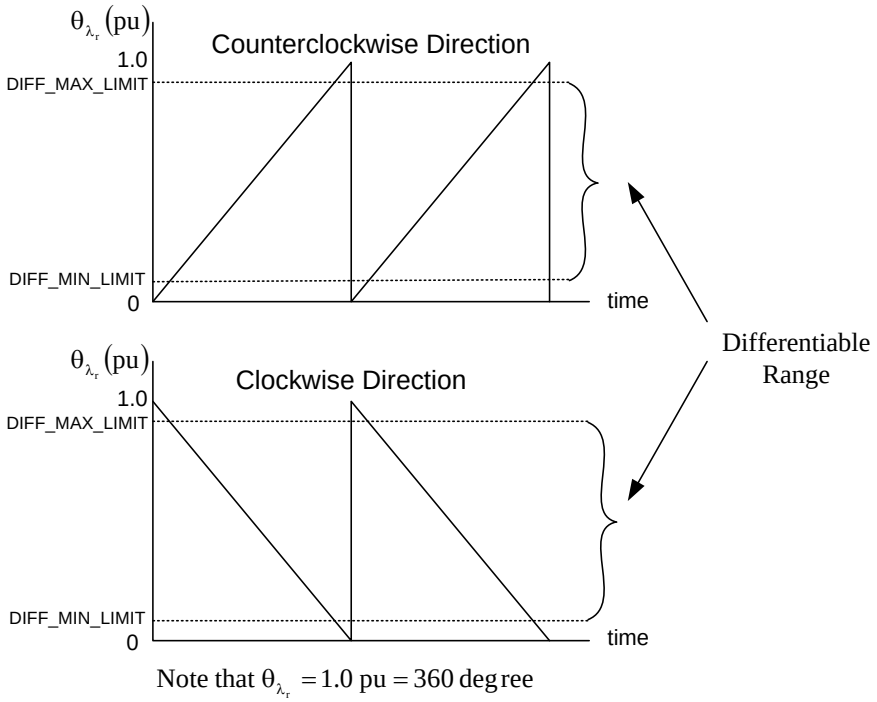


Figure 1: The waveforms of rotor flux angle in both directions

In addition, the synchronous speed in (21) is necessary to be filtered out by the low-pass filter in order to reduce the amplifying noise generated by the pure differentiator in (21). The simple 1st-order low-pass filter is used, then the actual synchronous speed to be used is the output of the low-pass filter, $\hat{\omega}_{e,pu}$, seen in following equation. The continuous-time equation of 1st-order low-pass filter is as

$$\frac{d\hat{\omega}_{e,pu}}{dt} = \frac{1}{\tau_c}\left(\omega_{e,pu} - \hat{\omega}_{e,pu}\right) \quad \text{pu} \qquad (22)$$

where $\tau_c = \frac{1}{2\pi f_c}$ is the low-pass filter time constant (sec), and f_c is the cut-off frequency (Hz).

Using backward approximation, then (22) finally becomes

$$\hat{\omega}_{e,pu}(k) = K_3\hat{\omega}_{e,pu}(k-1) + K_4\omega_{e,pu}(k) \quad \text{pu} \qquad (23)$$

where $K_3 = \frac{\tau_c}{\tau_c + T}$, and $K_4 = \frac{T}{\tau_c + T}$.

In fact, only three equations (18), (21), and (23) are mainly employed to compute the estimated speed in per-unit. The required parameters for this module are summarized as follows:

The machine parameters:
- number of poles (p)
- rotor resistance (R_r)
- rotor leakage inductance (L_{rl})
- magnetizing inductance (L_m)

The based quantities:
- base current (I_b)
- base electrically angular velocity (ω_b)

The sampling period:
- sampling period (T)

Low-pass filter:
- cut-off frequency (f_c)

Notice that the rotor self inductance is $L_r = L_{rl} + L_m$ (H).

Next, Table 1 shows the correspondence of notations between variables used here and variables used in the program (i.e. aci_se.h). The software module requires that both input and output variables are in per unit values.

	Equation Variables	Program Variables
Inputs	λ_{dr}^{s}	PsiDrS
	λ_{qr}^{s}	PsiQrS
	θ_{λ_r}	ThetaFlux
	i_{ds}^{s}	IDsS
	i_{qs}^{s}	IQsS
Output	ω_r	WrHat
Others	$\left(\lambda_r^s\right)^2$	SquaredPsi
	ω_e	WPsi

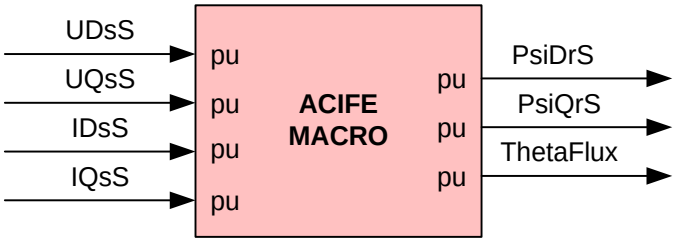
Table 1: Correspondence of notations

References:

[1] A.M. Trzynadlowski, The Field Orientation Principle in Control of Induction Motors, Kluwer Academic Publishers, 1994, pp. 176-180.

Description

This software module implements the flux estimator with the rotor flux angle for the 3-ph induction motor based upon the integral of back emf's (voltage model) approach. To reduce the errors due to pure integrator and stator resistance measurement, the compensated voltages produced by PI compensators are introduced. Therefore, this flux estimator can be operating over a wide range of speed, even at very low speed.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: aci_fe.h

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

C Interface

Object Definition

The structure of ACIFE object is defined by following structure definition

```
typedef struct { _iq  ThetaFlux;      // Output: Rotor flux angle
                 _iq  IQsS;           // Input: Stationary q-axis stator current
                 _iq  IDsS;           // Input: Stationary d-axis stator current
                 _iq  K1;             // Parameter: Constant using in current model
                 _iq  FluxDrE;        // Variable: Rotating d-axis rotor flux (current model)
                 _iq  K2;             // Parameter: Constant using in current model
                 _iq  FluxQrS;        // Variable: Stationary q-axis rotor flux (current model)
                 _iq  FluxDrS;        // Variable: Stationary d-axis rotor flux (current model)
                 _iq  K3;             // Parameter: Constant using in stator flux computation
                 _iq  K4;             // Parameter: Constant using in stator flux computation
                 _iq  FluxDsS;        // Variable: Stationary d-axis stator flux (current model)
                 _iq  FluxQsS;        // Variable: Stationary q-axis stator flux (current model)
                 _iq  PsiQsS;         // Variable: Stationary d-axis stator flux (voltage model)
                 _iq  Kp;             // Parameter: PI proportional gain
                 _iq  UiDsS;          // Variable: Stationary d-axis integral term
                 _iq  UCompDsS;       // Variable: Stationary d-axis compensated voltage
                 _iq  Ki;             // Parameter: PI integral gain
                 _iq  PsiQsS;         // Variable: Stationary q-axis stator flux (voltage model)
                 _iq  UiQsS;          // Variable: Stationary q-axis integral term
                 _iq  UCompQsS;       // Variable: Stationary q-axis compensated voltage
                 _iq  EmfDsS;         // Variable: Stationary d-axis back emf
                 _iq  UDsS;           // Input: Stationary d-axis stator voltage
                 _iq  K5;             // Parameter: Constant using in back emf computation
                 _iq  K6;             // Parameter: Constant using in back emf computation
                 _iq  EmfQsS;         // Variable: Stationary q-axis back emf
                 _iq  UQsS;           // Input: Stationary q-axis stator voltage
                 _iq  K8;             // Parameter: Constant using in rotor flux computation
                 _iq  K7;             // Parameter: Constant using in rotor flux computation
                 _iq  PsiDrS;         // Output: Stationary d-axis estimated rotor flux
                 _iq  PsiQrS;         // Output: Stationary q-axis estimated rotor flux
                 _iq  OldEmf;         // Variable: Old back emf term
                 _iq  Sine;           // Variable: Sine term
                 _iq  Cosine;         // Variable: Cosine term

               } ACIFE;
```

Module Terminal Variables

Item	Name	Description	Format*	Range(Hex)
Inputs	UDsS	stationary d-axis stator voltage	GLOBAL_Q	80000000-7FFFFFFF
	UQsS	stationary q-axis stator voltage	GLOBAL_Q	80000000-7FFFFFFF
	IDsS	stationary d-axis stator current	GLOBAL_Q	80000000-7FFFFFFF
	IQsS	stationary q-axis stator current	GLOBAL_Q	80000000-7FFFFFFF
Outputs	PsiDrS	stationary d-axis rotor flux linkage	GLOBAL_Q	80000000-7FFFFFFF
	PsiQrS	stationary q-axis rotor flux linkage	GLOBAL_Q	80000000-7FFFFFFF
	ThetaFlux	rotor flux linkage angle	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)
ACIFE parameter	K1	K1 = Tr/(Tr+T)	GLOBAL_Q	80000000-7FFFFFFF
	K2	K2 = T/(Tr+T)	GLOBAL_Q	80000000-7FFFFFFF
	K3	K3 = Lm/Lr	GLOBAL_Q	80000000-7FFFFFFF
	K4	K4 = (Ls*Lr-Lm*Lm)/(Lr*Lm)	GLOBAL_Q	80000000-7FFFFFFF
	K5	K5 = Rs*Ib/Vb	GLOBAL_Q	80000000-7FFFFFFF
	K6	K6 = T*Vb/(Lm*Ib)	GLOBAL_Q	80000000-7FFFFFFF
	K7	K7 = Lr/Lm	GLOBAL_Q	80000000-7FFFFFFF
	K8	K8 = (Ls*Lr-Lm*Lm)/(Lm*Lm)	GLOBAL_Q	80000000-7FFFFFFF
Internal	FluxDrE	stationary d-axis rotor flux	GLOBAL_Q	80000000-7FFFFFFF
	FluxQrE	stationary q-axis rotor flux	GLOBAL_Q	80000000-7FFFFFFF
	FluxDsS	stationary d-axis stator flux	GLOBAL_Q	80000000-7FFFFFFF
	FluxQsS	stationary q-axis stator flux	GLOBAL_Q	80000000-7FFFFFFF
	PsiQsS	stationary d-axis stator flux	GLOBAL_Q	80000000-7FFFFFFF
	PsiQsS	stationary q-axis stator flux	GLOBAL_Q	80000000-7FFFFFFF
	UiDsS	stationary d-axis integral term	GLOBAL_Q	80000000-7FFFFFFF
	UiQsS	stationary q-axis integral term	GLOBAL_Q	80000000-7FFFFFFF
	EmfDsS	stationary d-axis back emf	GLOBAL_Q	80000000-7FFFFFFF
	EmfQsS	stationary q-axis back emf	GLOBAL_Q	80000000-7FFFFFFF
	UCompDsS	stationary d-axis comp. voltage	GLOBAL_Q	80000000-7FFFFFFF
	UCompQsS	stationary q-axis comp. voltage	GLOBAL_Q	80000000-7FFFFFFF
	OldEmf	old abck emf term	GLOBAL_Q	80000000-7FFFFFFF
	Sine	sine term	GLOBAL_Q	80000000-7FFFFFFF
	Cosine	cosine term	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

ACIFE

The module definition is created as a data type. This makes it convenient to instance an interface to the flux estimator of Induction Motor module. To create multiple instances of the module simply declare variables of type ACIFE.

ACIFE_DEFAULTS

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

Module Usage

Instantiation
The following example instances two ACIFE objects
ACIFE fe1, fe2;

Initialization
To Instance pre-initialized objects
ACIFE fe1 = ACIFE_DEFAULTS;
ACIFE fe2 = ACIFE_DEFAULTS;
Invoking the computation macro
ACIFE_MACRO(fe1);
ACIFE_MACRO(fe2);

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

```
main()
{
        fe1.K1 = parem1_1;              // Pass parameters to fe1
                .                               .
                .                               .
                .                               .
        fe1.K8 = parem1_8;              // Pass parameters to fe1

        fe2.K1 = parem2_1;              // Pass parameters to fe2
                .                               .
                .                               .
                .                               .
        fe2.K10_fe = parem2_8;          // Pass parameters to fe2

}

void interrupt periodic_interrupt_isr()
{
        fe1.UDsS= voltage_dq1.d;        // Pass inputs to fe1
        fe1.UQsS= voltage_dq1.q;        // Pass inputs to fe1
        fe1.IQsS=current_dq1.d;         // Pass inputs to fe1
        fe1.IDsS=current_dq1.q;         // Pass inputs to fe1

        fe2.UDsS= voltage_dq2.d;        // Pass inputs to fe2
        fe2.UQsS= voltage_dq2.q;        // Pass inputs to fe2
        fe2.IQsS=current_dq2.d;         // Pass inputs to fe2
        fe2.IDsS=current_dq2.q;         // Pass inputs to fe2
```

```
        ACIFE_MACRO(fe1);                  // Call compute macro for fe1
        ACIFE_MACRO(fe2);                  // Call compute macro for fe2

        flux1.d = fe1.PsiDrS;               // Access the outputs of fe1
        flux1.q = fe1.PsiQrS;               // Access the outputs of fe1
       angle1 = fe1.ThetaFlux;              // Access the outputs of fe1

        flux2.d = fe2.PsiDrS;               // Access the outputs of fe2
        flux2.q = fe2.PsiQrS;               // Access the outputs of fe2
        angle2 = fe2.ThetaFlux;             // Access the outputs of fe2

}
```

Constant Computation Macro

Since the flux estimator of Induction motor module requires eight constants (K1,…, K8) to be input basing on the machine parameters, base quantities, mechanical parameters, and sampling period. These eight constants can be internally computed by the macro (aci_fe_const.h). The followings show how to use the C constant computation macro.

Object Definition

The structure of ACIFE_CONST object is defined by following structure definition

```
typedef struct { float32  Rs;        // Input: Stator resistantance
                 float32  Rr;        // Input: Rotor resistance (ohm)
                 float32  Ls;        // Input: Stator inductance (H)
                 float32 Lr;         // Input: Rotor inductance (H)
                 float32  Lm;        // Input: Magnetizing inductance (H)
                 float32  Ib;        // Input: Base phase current (amp)
                 float32  Vb;        // Input: Base phase voltage (volt)
                 float32  Ts;        // Input: Sampling period in sec
                 float32  K1;        // Output: constant using in rotor flux calculation
                 float32  K2;        // Output: constant using in rotor flux calculation
                 float32  K3;        // Output: constant using in rotor flux calculation
                 float32  K4;        // Output: constant using in stator current calculation
                 float32  K5;        // Output: constant using in stator current calculation
                 float32  K6;        // Output: constant using in stator current calculation
                 float32  K7;        // Output: constant using in stator current calculation
                 float32  K8;        // Output: constant using in torque calculation
               } ACIFE_CONST;
```

Module Terminal Variables

Item	Name	Description	Format	Range(Hex)
Inputs	Rs	Stator resistance (ohm)	Floating	N/A
	Rr	Rotor resistance (ohm)	Floating	N/A
	Ls	Stator inductance (H)	Floating	N/A
	Lr	Rotor inductance (H)	Floating	N/A
	Lm	Magnetizing inductance (H)	Floating	N/A
	Ib	Base phase current (amp)	Floating	N/A
	Vb	Base phase voltage (volt)	Floating	N/A
	Ts	Sampling period (sec)	Floating	N/A
Outputs	K1	constant using in rotor flux calculation	Floating	N/A
	K2	constant using in rotor flux calculation	Floating	N/A
	K3	constant using in rotor flux calculation	Floating	N/A
	K4	constant using in stator current cal.	Floating	N/A
	K5	constant using in stator current cal.	Floating	N/A
	K6	constant using in stator current cal.	Floating	N/A
	K7	constant using in stator current cal.	Floating	N/A
	K8	constant using in torque calculation	Floating	N/A

Special Constants and Data types

ACIFE_CONST
The module definition is created as a data type. This makes it convenient to instance an interface to the flux estimation of Induction Motor constant computation module. To create multiple instances of the module simply declare variables of type ACIFE_CONST.

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

Module Usage

Instantiation
The following example instances two ACIFE_CONST objects
ACIFE_CONST fe1_const, fe2_const;

Initialization

To Instance pre-initialized objects
ACIFE_CONST fe1_const = ACIFE_CONST_DEFAULTS;
ACIFE_CONST fe2_const = ACIFE_CONST_DEFAULTS;

Invoking the computation macro
ACIFE_CONST_MACRO(fe1_const);
ACIFE_CONST_MACRO(fe2_const);

Example

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

```
main()
{

        fe1_const.Rs = Rs1;             // Pass floating-point inputs to fe1_const
        fe1_const.Rr = Rr1;             // Pass floating-point inputs to fe1_const
        fe1_const.Ls = Ls1;             // Pass floating-point inputs to fe1_const
        fe1_const.Lr = Lr1;             // Pass floating-point inputs to fe1_const
        fe1_const.Lm = Lm1;             // Pass floating-point inputs to fe1_const
        fe1_const.Ib = Ib1;             // Pass floating-point inputs to fe1_const
        fe1_const.Vb = Vb1;             // Pass floating-point inputs to fe1_const
        fe1_const.Ts = Ts1;             // Pass floating-point inputs to fe1_const

        fe2_const.Rs = Rs2;             // Pass floating-point inputs to fe2_const
        fe2_const.Rr = Rr2;             // Pass floating-point inputs to fe2_const
        fe2_const.Ls = Ls2;             // Pass floating-point inputs to fe2_const
        fe2_const.Lr = Lr2;             // Pass floating-point inputs to fe2_const
        fe2_const.Lm = Lm2;             // Pass floating-point inputs to fe2_const
        fe2_const.Ib = Ib2;             // Pass floating-point inputs to fe2_const
        fe2_const.Vb = Vb2;             // Pass floating-point inputs to fe2_const
        fe2_const.Ts = Ts2;             // Pass floating-point inputs to fe2_const

        ACIFE_CONST_MACRO(fe1_const);   // Call compute macro for fe1_const
        ACIFE_CONST_MACRO (fe2_const);  // Call compute macro for fe2_const

        fe1.K1 = _IQ(fe1_const.K1);     // Access the floating-point outputs of fe1_const
                .                                       .
                .                                       .
                .                                       .
        fe1.K8 = _IQ(fe1_const.K8); // Access the floating-point outputs of fe1_const


        fe2.K1 = _IQ(fe2_const.K1);     // Access the floating-point outputs of fe2_const
                .                                       .
                .                                       .
                .                                       .
        fe2.K8 = _IQ(fe2_const.K8);   // Access the floating-point outputs of fe2_const

}
```

Technical Background

The overall of the flux estimator [1] can be shown in Figure 1. The rotor flux linkages in the stationary reference frame are mainly computed by means of the integral of back emf's in the voltage model. By introducing the compensated voltages generated by PI compensators, the errors associated with pure integrator and stator resistance measurement can be taken care. The equations derived for this flux estimator are summarized as follows:

Continuous time:

Firstly, the rotor flux linkage dynamics in synchronously rotating reference frame $\left(\omega = \omega_e = \omega_{\psi_r}\right)$ can be shown as below:

$$\frac{d\psi_{dr}^{e,i}}{dt} = \frac{L_m}{\tau_r} i_{ds}^{e} - \frac{1}{\tau_r}\psi_{dr}^{e,i} + \left(\omega_e - \omega_r\right)\psi_{qr}^{e,i} \quad (1)$$

$$\frac{d\psi_{qr}^{e,i}}{dt} = \frac{L_m}{\tau_r} i_{qs}^{e} - \frac{1}{\tau_r}\psi_{qr}^{e,i} - \left(\omega_e - \omega_r\right)\psi_{dr}^{e,i} \quad (2)$$

where L_m is the magnetizing inductance (H), $\tau_r = \frac{L_r}{R_r}$ is the rotor time constant (sec), and ω_r is the electrically angular velocity of rotor (rad/sec).

In the current model, total rotor flux linkage is aligned into the d-axis component, which is modeled by the stator currents, thus

$$\psi_r^{e,i} = \psi_{dr}^{e,i} \text{ and } \psi_{qr}^{e,i} = 0 \quad (3)$$

Substituting $\psi_{qr}^{e,i} = 0$ into (1)-(2), yields the oriented rotor flux dynamics are

$$\frac{d\psi_{dr}^{e,i}}{dt} = \frac{L_m}{\tau_r} i_{ds}^{e} - \frac{1}{\tau_r}\psi_{dr}^{e,i} \quad (4)$$

$$\psi_{qr}^{e,i} = 0 \quad (5)$$

Note that (4) and (5) are the classical rotor flux vector control equations. Then, the rotor flux linkages in (4)-(5) are transformed into the stationary reference frame performed by inverse park transformation.

$$\psi_{dr}^{s,i} = \psi_{dr}^{e,i}\cos\left(\theta_{\psi_r}\right) - \psi_{qr}^{e,i}\sin\left(\theta_{\psi_r}\right) = \psi_{dr}^{e,i}\cos\left(\theta_{\psi_r}\right) \quad (6)$$

$$\psi_{qr}^{s,i} = \psi_{dr}^{e,i}\sin\left(\theta_{\psi_r}\right) + \psi_{qr}^{e,i}\cos\left(\theta_{\psi_r}\right) = \psi_{dr}^{e,i}\sin\left(\theta_{\psi_r}\right) \quad (7)$$

where θ_{ψ_r} is the rotor flux angle (rad).

Then, the stator flux linkages in stationary reference frame are computed from the rotor flux linkages in (6)-(7)

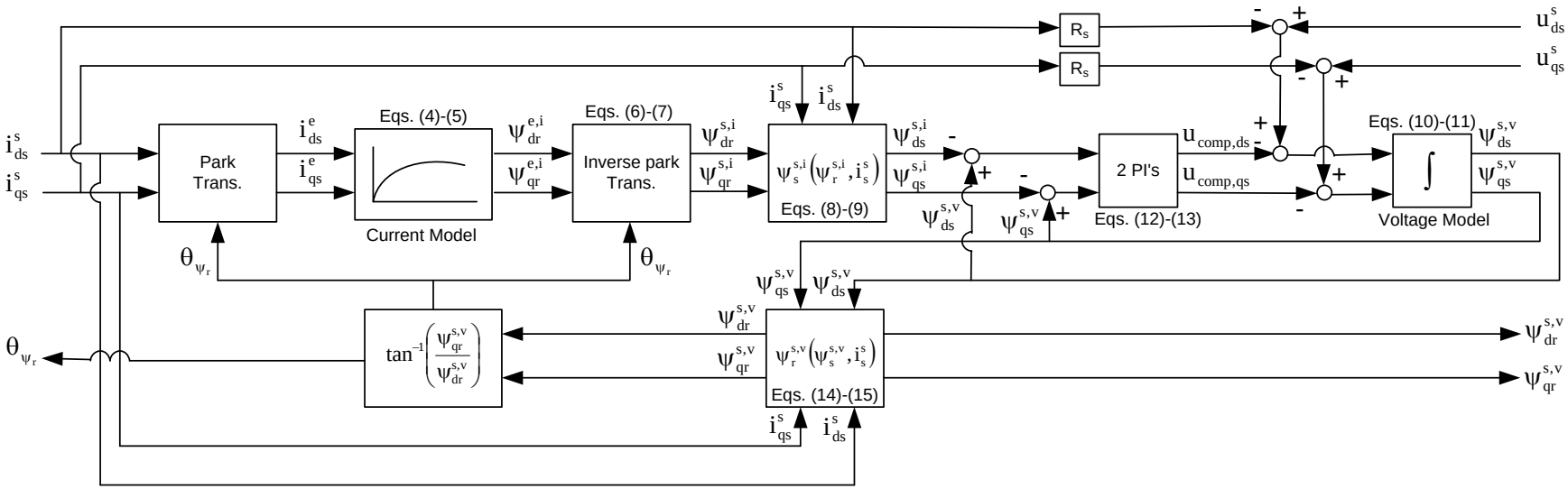


Figure 1: Overall system of flux estimator

$$\psi_{ds}^{s,i} = L_s i_{ds}^s + L_m i_{dr}^s = \left(\frac{L_s L_r - L_m^2}{L_r}\right) i_{ds}^s + \frac{L_m}{L_r}\psi_{dr}^{s,i} \quad (8)$$

$$\psi_{qs}^{s,i} = L_s i_{qs}^s + L_m i_{qr}^s = \left(\frac{L_s L_r - L_m^2}{L_r}\right) i_{qs}^s + \frac{L_m}{L_r}\psi_{qr}^{s,i} \quad (9)$$

where L_s and L_r are the stator and rotor self inductance (H), respectively.

Next, the stator flux linkages in the voltage model is computed by means of back emf's integration with compensated voltages.

$$\psi_{ds}^{s,v} = \int \left(u_{ds}^s - i_{ds}^s R_s - u_{comp,ds}\right) dt \quad (10)$$

$$\psi_{qs}^{s,v} = \int \left(u_{qs}^s - i_{qs}^s R_s - u_{comp,qs}\right) dt \quad (11)$$

where R_s is the stator resistance (Ω), u_{ds}^s, u_{qs}^s are stationary dq-axis stator voltages, and the compensated voltages are computed by the PI control law as follows:

$$u_{comp,ds} = K_p\left(\psi_{ds}^{s,v} - \psi_{ds}^{s,i}\right) + \frac{K_p}{T_I}\int \left(\psi_{ds}^{s,v} - \psi_{ds}^{s,i}\right) dt \quad (12)$$

$$u_{comp,qs} = K_p\left(\psi_{qs}^{s,v} - \psi_{qs}^{s,i}\right) + \frac{K_P}{T_I}\int \left(\psi_{qs}^{s,v} - \psi_{qs}^{s,i}\right) dt \quad (13)$$

The proportional gain K_P and the reset time T_I are chosen such that the flux linkages computed by current model is dominant at low speed because the back emf's computed by the voltage model are extremely low at this speed range (even zero back emf's at zero speed). While at high speed range, the flux linkages computed by voltage model is dominant.

Once the stator flux linkages in (10)-(11) are calculated, the rotor flux linkages based on the voltage model are further computed, by rearranging (8)-(9), as

$$\psi_{dr}^{s,v} = -\left(\frac{L_s L_r - L_m^2}{L_m}\right) i_{ds}^s + \frac{L_r}{L_m}\psi_{ds}^{s,v} \quad (14)$$

$$\psi_{qr}^{s,v} = -\left(\frac{L_s L_r - L_m^2}{L_m}\right) i_{qs}^s + \frac{L_r}{L_m}\psi_{qs}^{s,v} \quad (15)$$

Then, the rotor flux angle based on the voltage model is finally computed as

$$\theta_{\psi_r} = \tan^{-1}\left(\frac{\psi_{qr}^{s,v}}{\psi_{dr}^{s,v}}\right) \quad (16)$$

Discrete time:

The oriented rotor flux dynamics in (4) is discretized by using backward approximation as follows:

$$\frac{\psi_{dr}^{e,i}(k) - \psi_{dr}^{e,i}(k-1)}{T} = \frac{L_m}{\tau_r} i_{ds}^e(k) - \frac{1}{\tau_r}\psi_{dr}^{e,i}(k) \quad (17)$$

where T is the sampling period (sec). Rearranging (17), then it gives

$$\psi_{dr}^{e,i}(k) = \left(\frac{\tau_r}{\tau_r + T}\right)\psi_{dr}^{e,i}(k\text{-}1) + \left(\frac{L_m T}{\tau_r + T}\right) i_{ds}^e(k) \quad (18)$$

Next, the stator flux linkages in (10)-(11) are discretized by using trapezoidal (or tustin) approximation as

$$\psi_{ds}^{s,v}(k) = \psi_{ds}^{s,v}(k-1) + \frac{T}{2}\left(e_{ds}^{s}(k) + e_{ds}^{s}(k-1)\right) \quad (19)$$

$$\psi_{qs}^{s,v}(k) = \psi_{qs}^{s,v}(k-1) + \frac{T}{2}\left(e_{qs}^{s}(k) + e_{qs}^{s}(k-1)\right) \quad (20)$$

where the back emf's are computed as

$$e_{ds}^{s}(k) = u_{ds}^{s}(k) - i_{ds}^{s}(k)R_s - u_{comp,ds}(k) \quad (21)$$

$$e_{qs}^{s}(k) = u_{qs}^{s}(k) - i_{qs}^{s}(k)R_s - u_{comp,qs}(k) \quad (22)$$

Similarly, the PI control laws in (12)-(13) are also discretized by using trapezoidal approximation as

$$u_{comp,ds}(k) = K_p\left(\psi_{ds}^{s,v}(k) - \psi_{ds}^{s,i}(k)\right) + u_{comp,ds,i}(k-1) \quad (23)$$

$$u_{comp,qs}(k) = K_p\left(\psi_{qs}^{s,v}(k) - \psi_{qs}^{s,i}(k)\right) + u_{comp,qs,i}(k-1) \quad (24)$$

where the accumulating integral terms are as

$$\begin{aligned} u_{comp,ds,i}(k) &= u_{comp,ds,i}(k-1) + \frac{K_p T}{T_I}\left(\psi_{ds}^{s,v}(k) - \psi_{ds}^{s,i}(k)\right) \\ &= u_{comp,ds,i}(k-1) + K_p K_I\left(\psi_{ds}^{s,v}(k) - \psi_{ds}^{s,i}(k)\right) \end{aligned} \quad (25)$$

$$\begin{aligned} u_{comp,qs,i}(k) &= u_{comp,qs,i}(k-1) + \frac{K_p T}{T_I}\left(\psi_{qs}^{s,v}(k) - \psi_{qs}^{s,i}(k)\right) \\ &= u_{comp,qs,i}(k-1) + K_p K_I\left(\psi_{qs}^{s,v}(k) - \psi_{qs}^{s,i}(k)\right) \end{aligned} \quad (26)$$

where $K_I = \frac{T}{T_I}$.

Discrete time and Per-unit:

Now all equations are normalized into the per-unit by the specified base quantities. Firstly, the rotor flux linkage in current model (18) is normalized by dividing the base flux linkage as

$$\psi_{dr,pu}^{e,i}(k) = \left(\frac{\tau_r}{\tau_r + T}\right)\psi_{dr,pu}^{e,i}(k-1) + \left(\frac{T}{\tau_r + T}\right)i_{ds,pu}^{e}(k) \quad \text{pu} \quad (27)$$

where $\psi_b = L_m I_b$ is the base flux linkage (volt.sec) and I_b is the base current (amp).

Next, the stator flux linkages in the current model (8)-(9) are similarly normalized by dividing the base flux linkage as

$$\psi_{ds,pu}^{s,i}(k) = \left(\frac{L_s L_r - L_m^2}{L_r L_m}\right)i_{ds,pu}^{s}(k) + \frac{L_m}{L_r}\psi_{dr,pu}^{s,i}(k) \quad \text{pu} \quad (28)$$

$$\psi^{s,i}_{qs,pu}(k) = \left(\frac{L_s L_r - L_m^2}{L_r L_m}\right) i^s_{qs,pu}(k) + \frac{L_m}{L_r}\psi^{s,i}_{qr,pu}(k) \quad \text{pu} \quad (29)$$

Then, the back emf's in (21)-(22) are normalized by dividing the base phase voltage V_b

$$e^s_{ds,pu}(k) = u^s_{ds,pu}(k) - \frac{I_b R_s}{V_b} i^s_{ds,pu}(k) - u_{comp,ds,pu}(k) \quad \text{pu} \quad (30)$$

$$e^s_{qs,pu}(k) = u^s_{qs,pu}(k) - \frac{I_b R_s}{V_b} i^s_{qs,pu}(k) - u_{comp,qs,pu}(k) \quad \text{pu} \quad (31)$$

Next, the stator flux linkages in the voltage model (19)-(20) are divided by the base flux linkage.

$$\psi^{s,v}_{ds,pu}(k) = \psi^{s,v}_{ds,pu}(k-1) + \frac{V_b T}{L_m I_b}\left(\frac{e^s_{ds,pu}(k) + e^s_{ds,pu}(k-1)}{2}\right) \quad \text{pu} \quad (32)$$

$$\psi^{s,v}_{qs,pu}(k) = \psi^{s,v}_{qs,pu}(k-1) + \frac{V_b T}{L_m I_b}\left(\frac{e^s_{qs,pu}(k) + e^s_{qs,pu}(k-1)}{2}\right) \quad \text{pu} \quad (33)$$

Similar to (28)-(29), the normalized rotor flux linkages in voltage model are

$$\psi^{s,v}_{dr,pu}(k) = -\left(\frac{L_s L_r - L_m^2}{L_m L_m}\right) i^s_{ds,pu}(k) + \frac{L_r}{L_m}\psi^{s,v}_{ds,pu}(k) \quad \text{pu} \quad (34)$$

$$\psi^{s,v}_{qr,pu}(k) = -\left(\frac{L_s L_r - L_m^2}{L_m L_m}\right) i^s_{qs,pu}(k) + \frac{L_r}{L_m}\psi^{s,v}_{qs,pu}(k) \quad \text{pu} \quad (35)$$

In conclusion, the discrete-time, per-unit equations are rewritten in terms of constants.

Current model – rotor flux linkage in synchronously rotating reference frame $\left(\omega = \omega_{\psi_r}\right)$

$$\psi^{e,i}_{dr,pu}(k) = K_1 \psi^{e,i}_{dr,pu}(k-1) + K_2 i^e_{ds,pu}(k) \quad \text{pu} \quad (36)$$

where $K_1 = \dfrac{\tau_r}{\tau_r + T}$, and $K_2 = \dfrac{T}{\tau_r + T}$.

Current model – rotor flux linkages in the stationary reference frame $(\omega = 0)$

$$\psi^{s,i}_{ds,pu}(k) = K_4 i^s_{ds,pu}(k) + K_3 \psi^{s,i}_{dr,pu}(k) \quad \text{pu} \quad (37)$$

$$\psi^{s,i}_{qs,pu}(k) = K_4 i^s_{qs,pu}(k) + K_3 \psi^{s,i}_{qr,pu}(k) \quad \text{pu} \quad (38)$$

where $K_3 = \dfrac{L_m}{L_r}$, and $K_4 = \dfrac{L_s L_r - L_m^2}{L_r L_m}$.

Voltage model – back emf's in the stationary reference frame $(\omega = 0)$

$$e^s_{ds,pu}(k) = u^s_{ds,pu}(k) - K_5 i^s_{ds,pu}(k) - u_{comp,ds,pu}(k) \quad \text{pu} \quad (39)$$

$$e^s_{qs,pu}(k) = u^s_{qs,pu}(k) - K_5 i^s_{qs,pu}(k) - u_{comp,qs,pu}(k) \quad \text{pu} \quad (40)$$

where $K_5 = \dfrac{I_b R_s}{V_b}$.

Voltage model – stator flux linkages in the stationary reference frame $(\omega = 0)$

$$\psi^{s,v}_{ds,pu}(k) = \psi^{s,v}_{ds,pu}(k-1) + K_6\left(\frac{e^{s}_{ds,pu}(k) + e^{s}_{ds,pu}(k-1)}{2}\right) \quad \text{pu} \quad (41)$$

$$\psi^{s,v}_{qs,pu}(k) = \psi^{s,v}_{qs,pu}(k-1) + K_6\left(\frac{e^{s}_{qs,pu}(k) + e^{s}_{qs,pu}(k-1)}{2}\right) \quad \text{pu} \quad (42)$$

where $K_6 = \frac{V_b T}{L_m I_b}$.

Voltage model – rotor flux linkages in the stationary reference frame $(\omega = 0)$

$$\psi^{s,v}_{dr,pu}(k) = -K_8 i^{s}_{ds,pu}(k) + K_7 \psi^{s,v}_{ds,pu}(k) \quad \text{pu}$$

(43)

$$\psi^{s,v}_{qr,pu}(k) = -K_8 i^{s}_{qs,pu}(k) + K_7 \psi^{s,v}_{qs,pu}(k) \quad \text{pu} \quad (44)$$

where $K_7 = \frac{L_r}{L_m}$, and $K_8 = \frac{L_s L_r - L_m^2}{L_m L_m}$.

Voltage model – rotor flux angle

$$\theta_{\psi_r,pu}(k) = \frac{1}{2\pi}\tan^{-1}\left(\frac{\psi^{s,v}_{qr,pu}(k)}{\psi^{s,v}_{dr,pu}(k)}\right) \quad \text{pu} \quad (45)$$

Notice that the rotor flux angle is computed by a look-up table of 0^o-45^o with 256 entries.

In fact, equations (36)-(44) are mainly employed to compute the estimated flux linkages in per-unit. The required parameters for this module are summarized as follows:

The machine parameters:
- stator resistance (R_s)
- 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 phase voltage (V_b)

The sampling period:
- sampling period (T)

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

Next, Table 1 shows the correspondence of notations between variables used here and variables used in the program (i.e. aci_fe.h). The software module requires that both input and output variables are in per unit values.

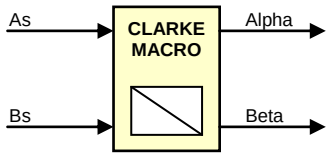
	Equation Variables	Program Variables
Inputs	u^{s}_{ds}	UDsS
	u^{s}_{qs}	UQsS
	i^{s}_{ds}	IDsS
	i^{s}_{qs}	IQsS
Outputs	$\psi^{s,v}_{dr}$	PsiDrS
	$\psi^{s,v}_{qr}$	PsiQrS
	θ_{ψ_r}	ThetaFlux
Others	$\psi^{e,i}_{dr}$	FluxDrE
	$\psi^{s,i}_{dr}$	FluxDrS
	$\psi^{s,i}_{qr}$	FluxQrS
	$\psi^{s,i}_{ds}$	FluxDsS
	$\psi^{s,i}_{qs}$	FluxQsS
	$\psi^{s,v}_{ds}$	PsiQsS
	$\psi^{s,v}_{qs}$	PsiQsS
	e^{s}_{ds}	EmfDsS
	e^{s}_{qs}	EmfQsS
	$u_{comp,ds}$	UCompDsS
	$u_{comp,qs}$	UCompQsS

Table 1: Correspondence of notations

References:

[1] C. Lascu, I. Boldea, and F. Blaabjerg, “A modified direct torque control for induction motor sensorless drive”, *IEEE Trans. Ind. Appl.*, vol. 36, no. 1, pp. 122-130, January/February 2000.

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



Availability C interface version

Module Properties **Type:** Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: clarke.h

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

C Interface

Object Definition

The structure of CLARKE object is defined by following structure definition

```
typedef struct {  _iq  As;          // Input: phase-a stator variable
                  _iq  Bs;          // Input: phase-b stator variable
                  _iq  Cs;          // Input: phase-c stator variable
                  _iq  Alpha;       // Output: stationary d-axis stator variable
                  _iq  Beta;        // Output: stationary q-axis stator variable
                } CLARKE;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	As	Phase 'a' component of the balanced three phase quantities	GLOBAL_Q	80000000-7FFFFFFF
	Bs	Phase 'b' component of the balanced three phase quantities	GLOBAL_Q	80000000-7FFFFFFF
	Cs	Phase 'c' component of the balanced three phase quantities	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Alpha	Direct axis(d) component of the transformed signal	GLOBAL_Q	80000000-7FFFFFFF
	Beta	Quadrature axis(q) component of the transformed signal	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

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

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

Module Usage

Instantiation
The following example instances two CLARKE objects
CLARKE clarke1, clarke2;

Initialization

To Instance pre-initialized objects

```
CLARKE clarke1 = CLARKE_DEFAULTS;
CLARKE clarke2 = CLARKE_DEFAULTS;
```

Invoking the computation macro

```
CLARKE_MACRO (clarke1);
CLARKE_MACRO (clarke2);
```

Example

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        clarke1.As = as1;               // Pass inputs to clarke1
        clarke1.Bs = bs1;               // Pass inputs to clarke1

        clarke2.As = as2;               // Pass inputs to clarke2
        clarke2.Bs = bs2;               // Pass inputs to clarke2

        CLARKE_MACRO (clarke1);     // Call compute macro for clarke1
        CLARKE_MACRO (clarke2);     // Call compute macro for clarke2

        ds1 = clarke1.Alpha;            // Access the outputs of clarke1
        qs1 = clarke1.Beta;             // Access the outputs of clarke1

        ds2 = clarke2.Alpha;            // Access the outputs of clarke2
        qs2 = clarke2.Beta;             // Access the outputs of clarke2
}
```

Technical Background

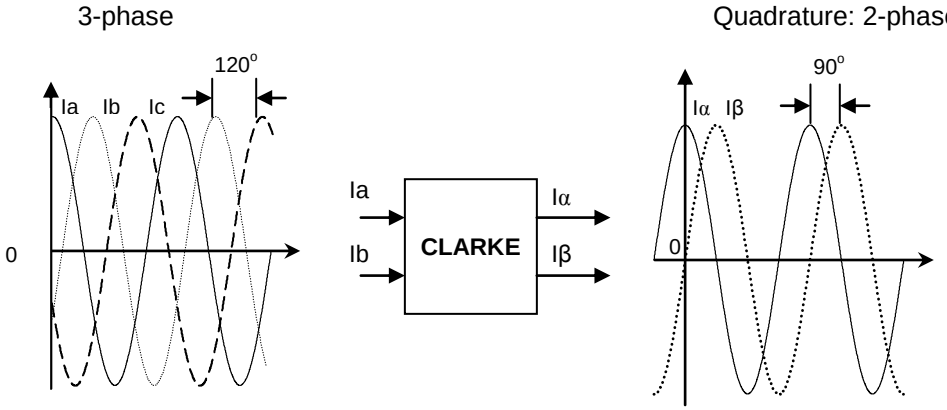
Here it is assumed that all three phases are balanced (i.e. Ia + Ib + Ic=0) and they have positive sequence (ABC) as follows:

$$Ia = I \times \cos(wt)$$
$$Ib = I \times \cos(wt - 2\pi/3)$$
$$Ic = I \times \cos(wt - 4\pi/3)$$

This macro implements the following equations:

$$\begin{cases} I\alpha = Ia \\ I\beta = (2Ib + Ia)/\sqrt{3} \end{cases}$$ which result in $$\begin{cases} I\alpha = I \times \cos(wt) \\ I\beta = I \times sin(wt) \end{cases}$$

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



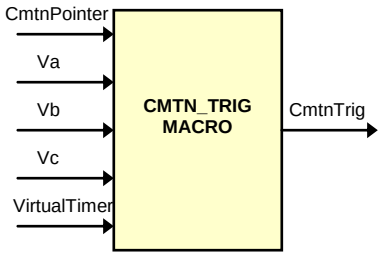
Next, Table 1 shows the correspondence of notations between variables used here and variables used in the program (i.e. clarke.h). The software module requires that both input and output variables are in per unit values.

	Equation Variables	Program Variables
Inputs	ia	As
	ib	Bs
	ic	Cs
Outputs	Iα	Alpha
	Iβ	Beta

Table 1: Correspondence of notations

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

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: com_trig.h

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

C Interface

Object Definition

The structure of CMTN object is defined by following structure definition

```
typedef  struct { Uint32 CmtnTrig;             // Output: Commutation trigger output (0 or 00007FFF)
                  _iq Va;                      // Input: Motor phase a voltages referenced to GND
                  _iq Vb;                      // Input: Motor phase b voltages referenced to GND
                  _iq Vc;                      // Input: Motor phase c voltages referenced to GND
                  _iq Neutral;                 // Variable: 3*Motor netural voltage
                  Uint32 RevPeriod;            // Variable: revolution time counter (Q0)
                  Uint32 ZcTrig;               // Variable: Zero-Crossing trig flag (0 or 00007FFF)
                  Uint32 CmtnPointer;          // Input: Commutation state pointer input (Q0)
                  _iq DebugBemf;               // Variable: 3*Back EMF = 3*(vx=vn), x=a,b,c
                  Uint32 NoiseWindowCounter; // Variable: Noise windows counter (Q0)
                  Uint32 Delay30DoneFlag;      // Variable: 30 Deg delay flag (0 or 0000000F)
                  Uint32 NewTimeStamp;     // Variable: Time stamp (Q0)
                  Uint32 OldTimeStamp;       // History: Previous time stamp (Q0)
                  Uint32 VirtualTimer;         // Input: Virtual timer (Q0)
                  Uint32 CmtnDelay;            // Variable: Time delay (Q0)
                  Uint32 DelayTaskPointer;  // Variable: Delay task pointer, see note below (0 or 1)
                  Uint32 NoiseWindowMax;  // Variable: Maximum noise windows counter (Q0)
                  Uint32 CmtnDelayCounter;   // Variable: Time delay counter (Q0)
                  Uint32 NWDelta;              // Variable: Noise windows delta (Q0)
                  Uint32 NWDelayThres;         // Variable: Noise windows dynamic threshold (Q0)
                } CMTN;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	CmtnPointer	Commutation state pointer input. This is used for Bemf zero crossing point calculation for the appropriate motor phase.	Q0	0 - 5
	Va	Motor phase-a voltages referenced to GND.	GLOBAL_Q	00000000-7FFFFFFF
	Vb	Motor phase-b voltages referenced to GND.	GLOBAL_Q	00000000-7FFFFFFF
	Vc	Motor phase-c voltages referenced to GND.	GLOBAL_Q	00000000-7FFFFFFF
	VirtualTimer	A virtual timer used for commutation delay angle calculation.	Q0	80000000-7FFFFFFF
Output	CmtnTrig	Commutation trigger output.	Q0	0 or 00007FFF
Internal	Neutral	3*Motor netural voltage	GLOBAL_Q	80000000-7FFFFFFF
	RevPeriod	revolution time counter	Q0	00000000-7FFFFFFF
	ZcTrig	Zero-Crossing trig flag	Q0	0 or 00007FFF
	DebugBemf	3*Back EMF	GLOBAL_Q	80000000-7FFFFFFF
	NoiseWindowCounter	Noise windows counter	Q0	80000000-7FFFFFFF
	Delay30DoneFlag	30 Deg delay flag	Q0	0 or 0000000F
	NewTimeStamp	Time stamp	Q0	00000000-7FFFFFFF
	OldTimeStamp	Previous time stamp	Q0	00000000-7FFFFFFF
	CmtnDelay	Time delay in terms of number of sampling time periods	Q0	00000000-7FFFFFFF
	DelayTaskPointer	Delay task pointer	Q0	0 or 1
	NoiseWindowMax	Maximum noise windows counter	Q0	80000000-7FFFFFFF
	CmtnDelayCounter	Time delay counter	Q0	80000000-7FFFFFFF
	NWDelta	Noise windows delta	Q0	80000000-7FFFFFFF
	NWDelayThres	Noise windows dynamic threshold	Q0	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

CMTN

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

CMTN_DEFAULTS

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

Module Usage

Instantiation
The following example instances two CMTN objects
CMTN cm_trig1, cm_trig2;

Initialization
To Instance pre-initialized objects
CMTN cm_trig1 = CMTN_DEFAULTS;
CMTN cm_trig2 = CMTN_DEFAULTS;

Invoking the computation macro
CMTN_TRIG_MACRO (cm_trig1);
CMTN_TRIG_MACRO (cm_trig2);

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        cm_trig1.CmtnPointer = input11;         // Pass inputs to cm_trig1
        cm_trig1.Va = input12;                  // Pass inputs to cm_trig1
        cm_trig1.Vb = input13;                  // Pass inputs to cm_trig1
        cm_trig1.Vc = input14;                  // Pass inputs to cm_trig1
        cm_trig1.VirtualTimer = input15;        // Pass inputs to cm_trig1

        cm_trig2.CmtnPointer = input21;         // Pass inputs to cm_trig2
        cm_trig2.Va = input22;                  // Pass inputs to cm_trig2
        cm_trig2.Vb = input23;                  // Pass inputs to cm_trig2
        cm_trig2.Vc = input24;                  // Pass inputs to cm_trig2
        cm_trig2.VirtualTimer = input25;        // Pass inputs to cm_trig2

        CMTN_TRIG_MACRO (cm_trig1);             // Call compute macro for cm_trig1
        CMTN_TRIG_MACRO (cm_trig2);             // Call compute macro for cm_trig2

        out1 = cm_trig1.CmtnTrig;               // Access the outputs of cm_trig1
        out2 = cm_trig2.CmtnTrig;               // Access the outputs of cm_trig2
}
```

Technical Background

Figure 1 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.

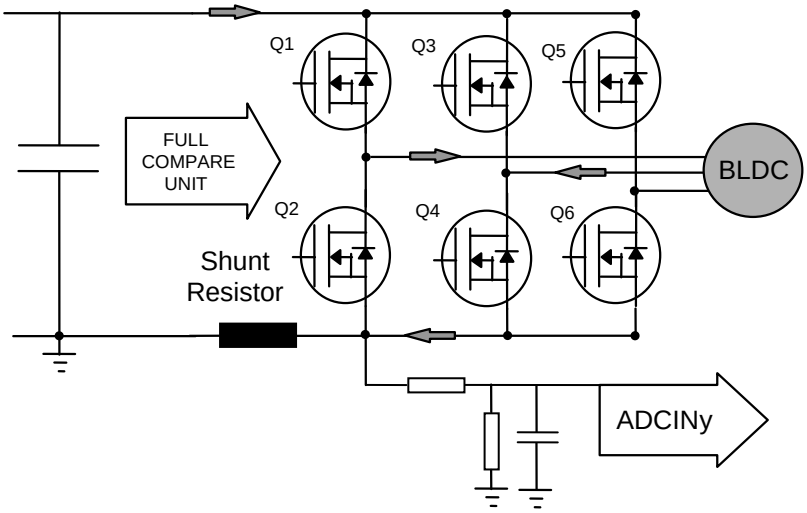


Figure 1: Three Phase Power Inverter for a BLDC Motor Drive

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.

The figure below shows the basic hardware necessary to perform these tasks.

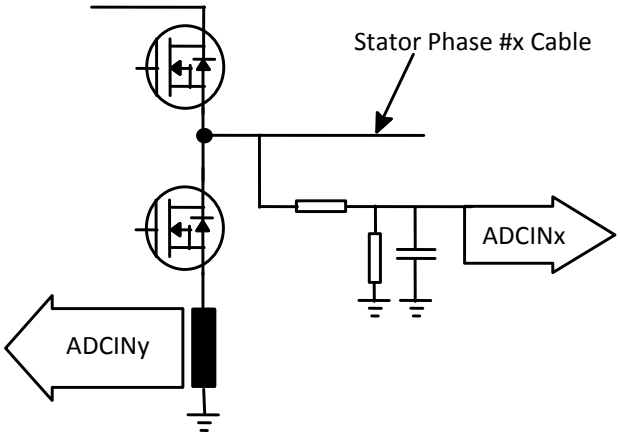


Figure 2: Basic Sensorless Additional Hardware

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 3 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 2.

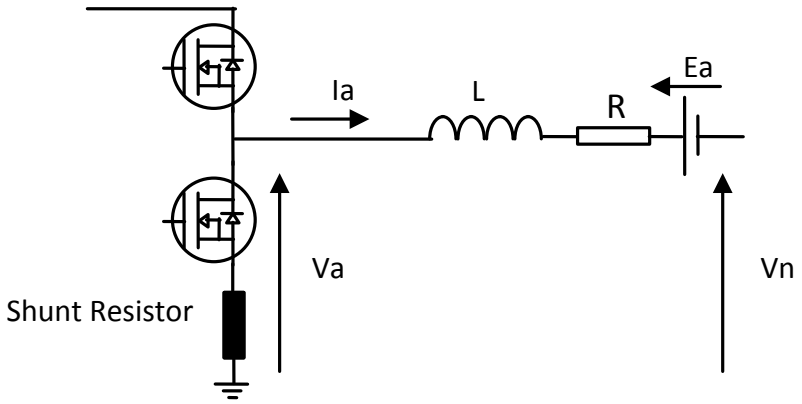


Figure 3: Stator Terminal Electrical Model

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

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

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

$$Vc = Ec + Vn$$

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 4. 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*.

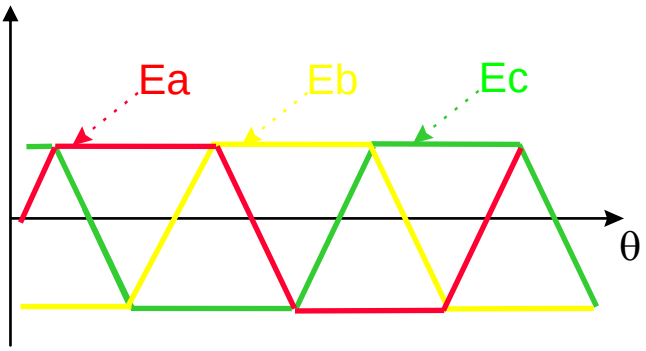


Figure 4: Instantaneous Bemf Wave-forms

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, e.g., 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 30º 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 30º 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 360º 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 30º. The corresponding time delay is represented in terms of the number of sampling time periods and is stored in the variable *CmtnDelay*. Therefore,

Time delay = *CmtnDelay* **.**Ts = T(α/360) = *VirtualTimer***.**Ts(α/360) = *VirtualTimer* **.** Ts/12

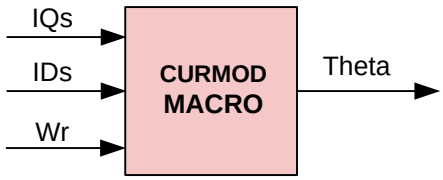
Where, Ts is the sampling time period and *VirtualTimer* is a timer that counts the number of sampling cycles during the previous revolution of the rotor.

The above equation is further simplified as,

CmtnDelay = *VirtualTimer* /12

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

Description This module takes as input both IQs and IDs, currents coming from the PARK transform, as well as the rotor mechanical speed and gives the rotor flux position.



Availability C interface version

Module Properties **Type:** Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: cur_mod.h

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

C Interface

Object Definition

The structure of CURMOD object is defined by following structure definition

```
typedef struct  { _iq  IDs;          // Input: Syn. rotating d-axis current
                  _iq  IQs;          // Input: Syn. rotating q-axis current
                  _iq  Wr;           // Input: Rotor electrically angular velocity
                  _iq  IMDs;         // Variable: Syn. rotating d-axis magnetizing current
                  _iq  Theta;        // Output: Rotor flux angle
                  _iq  Kr;           // Parameter: constant using in magnetizing current calc
                  _iq  Kt;           // Parameter: constant using in slip calculation
                  _iq  K;            // Parameter: constant using in rotor flux angle calculation
                } CURMOD;
```

Module Terminal Variables

Item	Name	Description	Format*	Range(Hex)
Inputs	IDs	Syn. rotating d-axis current	GLOBAL_Q	80000000-7FFFFFFF
	IQs	Syn. rotating d-axis current	GLOBAL_Q	80000000-7FFFFFFF
	Wr	Rotor electrically angular velocity	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Theta	Rotor flux angle	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)
CUR_MOD parameter	Kr	Kr = T/Tr	GLOBAL_Q	80000000-7FFFFFFF
	Kt	Kt = 1/(Tr*wb)	GLOBAL_Q	80000000-7FFFFFFF
	K	K = T*fb	GLOBAL_Q	80000000-7FFFFFFF
Internal	Wslip	Slip frequency	GLOBAL_Q	80000000-7FFFFFFF
	We	Synchronous frequency	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued betWeen 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

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

CURMOD_DEFAULTS
Structure symbolic constant to initialize CURMOD module. This provides the initial values to the terminal variables as Well as method pointers.

Module Usage

Instantiation
The following example instances two CURMOD objects
CURMOD cm1, cm2;

Initialization
To Instance pre-initialized objects
CURMOD cm1 = CURMOD_DEFAULTS;
CURMOD cm2 = CURMOD_DEFAULTS;

Invoking the computation macro
CURMOD_MACRO(cm1);
CURMOD_MACRO(cm2);

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

```
main()
{
        cm1.Kr = parem1_1;          // Pass parameters to cm1
        cm1.Kt = parem1_2;          // Pass parameters to cm1
        cm1.K = parem1_3;           // Pass parameters to cm1

        cm2.Kr = parem2_1;          // Pass parameters to cm2
        cm2.Kt = parem2_2;          // Pass parameters to cm2
        cm2.K = parem2_3;           // Pass parameters to cm2

}
void interrupt periodic_interrupt_isr()
{
        cm1.IDs = de1;              // Pass inputs to cm1
        cm1.IQs = qe1;              // Pass inputs to cm1
        cm1.Wr = Wr1;               // Pass inputs to cm1

        cm2.IDs = de2;              // Pass inputs to cm2
        cm2.IQs = qe2;              // Pass inputs to cm2
        cm2.Wr = Wr2;               // Pass inputs to cm2

        CURMOD_MACRO(cm1);          // Call compute macro for cm1
        CURMOD_MACRO(cm2);          // Call compute macro for cm2

        ang1 = cm1.Theta;           // Access the outputs of cm1
        ang2 = cm2.Theta;           // Access the outputs of cm2
}
```

Constant Computation Macro

Since the current model module requires three constants (Kr, Kt, and K) to be input basing on the machine parameters, base quantities, mechanical parameters, and sampling period. These four constants can be internally computed by the macro (cur_const.h). The followings show how to use the C constant computation macro.

Object Definition

The structure of CURMOD_CONST object is defined by following structure definition

```
typedef struct {    float32  Rr;     // Input: Rotor resistance (ohm)
                    float32  Lr;     // Input: Rotor inductance (H)
                    float32  fb;     // Input: Base electrical frequency (Hz)
                    float32  Ts;     // Input: Sampling period (sec)
                    float32  Kr;     // Output: constant using in magnetizing current calculation
                    float32  Kt;     // Output: constant using in slip calculation
                    float32  K;      // Output: constant using in rotor flux angle calculation
                } CURMOD_CONST;
```

Module Terminal Variables

Item	Name	Description	Format	Range(Hex)
Inputs	Rr	Rotor resistance (ohm)	Floating	N/A
	Lr	Rotor inductance (H)	Floating	N/A
	fb	Base electrical frequency (Hz)	Floating	N/A
	Ts	Sampling period (sec)	Floating	N/A
Outputs	Kr	constant using in current model calculation	Floating	N/A
	Kt	constant using in current model calculation	Floating	N/A
	K	constant using in current model calculation	Floating	N/A

Special Constants and Data types

CURMOD_CONST

The module definition is created as a data type. This makes it convenient to instance an interface to the current model constant computation module. To create multiple instances of the module simply declare variables of type CURMOD_CONST.

CURMOD_CONST_DEFAULTS

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

Module Usage

Instantiation

The following example instances two CURMOD_CONST objects

```
CURMOD_CONST cm1_const, cm2_const;
```

Initialization

To Instance pre-initialized objects

```
CURMOD_CONST cm1_const = CURMOD_CONST_DEFAULTS;
CURMOD_CONST cm2_const = CURMOD_CONST_DEFAULTS;
```

Invoking the computation macro

```
CURMOD_CONST_MACRO (cm1_const);
CURMOD_CONST_MACRO (cm2_const);
```

Example

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

```
main()
{

        cm1_const.Rr = Rr1;             // Pass floating-point inputs to cm1_const
        cm1_const.Lr = Lr1;             // Pass floating-point inputs to cm1_const
        cm1_const.fb = Fb1;             // Pass floating-point inputs to cm1_const
        cm1_const.Ts = Ts1;             // Pass floating-point inputs to cm1_const

        cm2_const.Rr = Rr2;             // Pass floating-point inputs to cm2_const
        cm2_const.Lr = Lr2;             // Pass floating-point inputs to cm2_const
        cm2_const.fb = Fb2;             // Pass floating-point inputs to cm2_const
        cm2_const.Ts = Ts2;             // Pass floating-point inputs to cm2_const

        CURMOD_CONST_MACRO (cm1_const); // Call compute macro for cm1_const
        CURMOD_CONST_MACRO (cm2_const); // Call compute macro for cm2_const

        cm1.Kr = _IQ(cm1_const.Kr);     // Access the floating-point outputs of cm1_const
        cm1.Kt = _IQ(cm1_const.Kt);     // Access the floating-point outputs of cm1_const
        cm1.K = _IQ(cm1_const.K);       // Access the floating-point outputs of cm1_const

        cm2.Kr = _IQ(cm2_const.Kr);     // Access the floating-point outputs of cm2_const
        cm2.Kt = _IQ(cm2_const.Kt);     // Access the floating-point outputs of cm2_const
        cm2.K = _IQ(cm2_const.K);       // Access the floating-point outputs of cm2_const

}
```

Technical Background

With the 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 the mechanical position sensor used with the asynchronous motor (QEP or tachometer). The current model module be added to the generic structure in the regulation block diagram to perform a current and speed closed loop for a three phases 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 as it is this system that outputs the rotor flux speed that will be integrated to get the rotor flux position.

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

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

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

In this equation system, T represents the Main loop control period. In a FOC control this usually corresponds to the Timer 1 underflow interrupt period.

Let the two above equations constants $\frac{T}{T_R}$ and $\frac{1}{T_R \omega_b}$ be renamed respectively K_t and K_R. These two constants need to be calculated according to the motor parameters and initialize into the cur_mod.h file.

Once the motor flux speed (fs) has been calculated, the necessary rotor flux position in per-unit (θ) is computed by the integration formula:

$$\theta = \theta_{k-1} + Kf_{s_k}$$

where $K = Tf_b$

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 to do so are the rotor resistance, the rotor inductance (which is the sum of the magnetizing inductance and the rotor leakage inductance ($L_R = L_H + L_{\sigma R}$)).

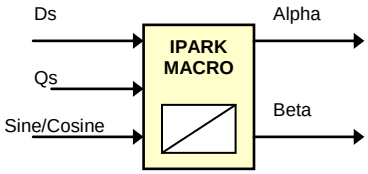
Next, Table 1 shows the correspondence of notations betWeen variables used here and variables used in the program (i.e. cur_mod.h). The software module requires that both input and output variables are in per unit values.

	Equation Variables	Program Variables
Inputs	i_{qS}	IQs
	i_{dS}	IDs
	n	Wr
Output	θ	Theta
Others	i_{mR}	IMDs

Table 1: Correspondence of notations

Description

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



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: ipark.h

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

C Interface

Object Definition

The structure of IPARK object is defined by following structure definition

```
typedef struct {  _iq  Alpha;      // Output: stationary d-axis stator variable
                  _iq  Beta;       // Output: stationary q-axis stator variable
                  _iq  Ds;         // Input: rotating d-axis stator variable
                  _iq  Qs;         // Input: rotating q-axis stator variable
                  _iq  Sine;       // Input: sine term
                  _iq  Cosine;     // Input: cosine term

                } IPARK;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	Ds	Direct axis(D) component of transformed signal in rotating reference frame	GLOBAL_Q	80000000-7FFFFFFF
	Qs	Quadrature axis(Q) component of transformed signal in rotating reference frame	GLOBAL_Q	80000000-7FFFFFFF
	Sine	Sine of the phase angle between stationary and rotating frame	GLOBAL_Q	80000000-7FFFFFFF
	Cosine	Cosine of the phase angle between stationary and rotating frame	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Alpha	Direct axis(d) component of the transformed signal	GLOBAL_Q	80000000-7FFFFFFF
	Beta	Quadrature axis(q) component of the transformed signal	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

IPARK

The module definition is created as a data type. This makes it convenient to instance an interface to the Inverse Park variable transformation. To create multiple instances of the module simply declare variables of type IPARK.

IPARK_DEFAULTS

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

Module Usage

Instantiation
The following example instances two IPARK objects
IPARK ipark1, ipark2;

Initialization
To Instance pre-initialized objects
IPARK ipark1 = IPARK_DEFAULTS;
IPARK ipark2 = IPARK_DEFAULTS;

Invoking the computation macro
IPARK_MACRO (ipark1);
IPARK_MACRO (ipark2);

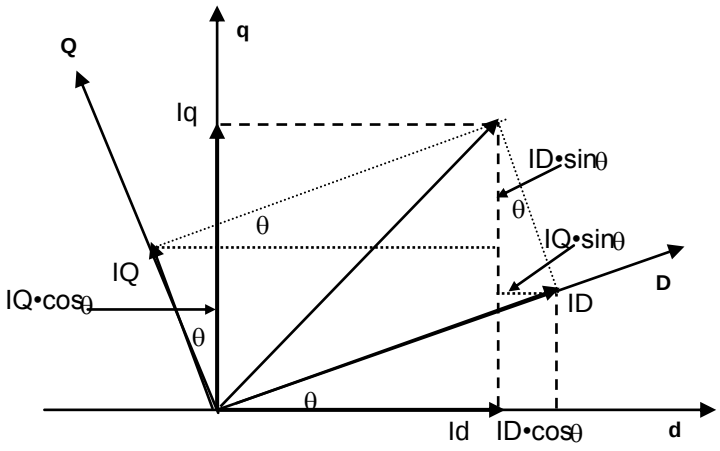
Example
The following pseudo code provides the information about the module usage.

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        ipark1.Ds = de1;                // Pass inputs to ipark1
        ipark1.Qs = qe1;                // Pass inputs to ipark1
        ipark1.Angle = ang1;            // Pass inputs to ipark1

        ipark2.Ds = de2;                // Pass inputs to ipark2
        ipark2.Qs = qe2;                // Pass inputs to ipark2
        ipark2.Angle = ang2;            // Pass inputs to ipark2

        IPARK_MACRO (ipark1);           // Call compute macro for ipark1
        IPARK_MACRO (ipark2);           // Call compute macro for ipark2

        ds1 = ipark1.Alpha;             // Access the outputs of ipark1
        qs1 = ipark1.Beta;              // Access the outputs of ipark1

        ds2 = ipark2.Alpha;             // Access the outputs of ipark2
        qs2 = ipark2.Beta;              // Access the outputs of ipark2

}
```

Technical Background

Implements the following equations:

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



Next, Table 1 shows the correspondence of notations between variables used here and variables used in the program (i.e. ipark.h). The software module requires that both input and output variables are in per unit values.

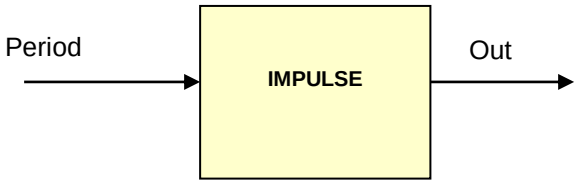
	Equation Variables	Program Variables
Inputs	ID	Ds
	IQ	Qs
	θ	Angle
	sin	Sine
	cos	Cosine
Outputs	id	Alpha
	iq	Beta

Table 1: Correspondence of notations

Description

This module implements a periodic impulse macro. The output variable *Out* is set to 0x00007FFF for 1 sampling period. The period of the output signal *Out* is specified by the input *Period.*

.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: impulse.h

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

C Interface

Object Definition

The structure of IMPULSE object is defined by following structure definition

```
typedef struct { Uint32  Period;      // Input: Period of output in # of sampling cycles (Q0)
                 Uint32  Out;         // Output: Impulse output (0x00000000 or 0x00007FFF)
                 Uint32  Counter;     // Variable: Impulse generator counter (Q0)
               } IMPULSE;
```

Item	Name	Description	Format*	Range(Hex)
Input	Period	Period of output in # of sampling period	Q0	00000000-7FFFFFFF
Output	Out	Impulse output	Q0	0 or 00007FFF
Internal	Counter	Impulse generator counter	Q0	00000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

IMPULSE

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

IMPULSE_DEFAULTS

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

Module Usage

Instantiation

The following example instances two IMPULSE objects
IMPULSE ig1, ig2;

Initialization

To Instance pre-initialized objects
IMPULSE ig1 = IMPULSE_DEFAULTS;
IMPULSE ig2 = IMPULSE_DEFAULTS;

Invoking the computation macro
IMPULSE_MACRO (ig1);
IMPULSE_MACRO (ig2);

Example

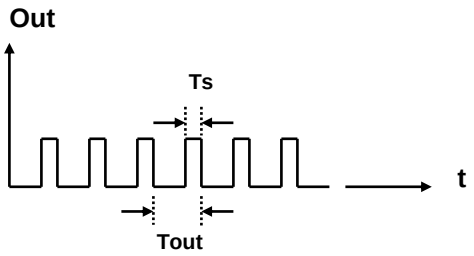
The following pseudo code provides the information about the module usage.

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        ig1.Period = input1;                    // Pass inputs to ig1
        ig2.Period = input2;                    // Pass inputs to ig2

        IMPULSE_MACRO (ig1);                    // Call compute macro for ig1
        IMPULSE_MACRO (ig2);                    // Call compute macro for ig2

        out1 = ig1.Out;                         // Access the outputs of ig1
        out2 = ig2.Out;                         // Access the outputs of ig2

}
```

Technical Background

Implements the following equation:

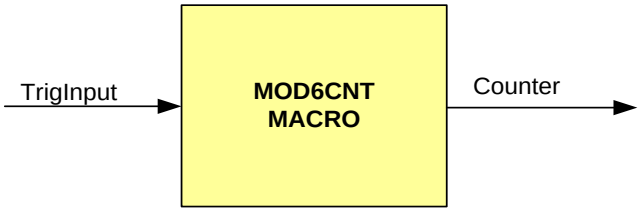
Out = 0x00007FFF, for t = n **.** Tout, n **=** 1, 2, 3, …….
= 0, otherwise
where,
Tout = Time period of output pulses = *Period* x Ts
Ts = Sampling time period



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 *Counter* changes to the next state every time it receives a trigger input through the input variable *TrigInput.*

.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: mod6_cnt.h

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

C Interface

Object Definition

The structure of MOD6CNT object is defined by following structure definition

```
typedef struct { Uint32  TrigInput;      // Input: Modulo 6 counter trigger 0x0000 or 0x7FFF)
                 Uint32  Counter;        // Output: Modulo 6 counter output  (0,1,2,3,4,5)
               } MOD6CNT;
```

Item	Name	Description	Format*	Range(Hex)
Input	TrigInput	Modulo 6 counter trigger	Q0	0 or 7FFF
Outputs	Counter	Modulo 6 counter output	Q0	0,1,2,3,4,5

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

MOD6CNT

The module definition is created as a data type. This makes it convenient to instance an interface to the modulo 6 counter. To create multiple instances of the module simply declare variables of type MOD6CNT.

MOD6CNT_DEFAULTS

Structure symbolic constant to initialize MOD6CNT module. This provides the initial values to the terminal variables as well as method pointers.

Module Usage

Instantiation

The following example instances two MOD6CNT objects
MOD6CNT mod1, mod2;

Initialization

To Instance pre-initialized objects
MOD6CNT mod1 = MOD6CNT_DEFAULTS;
MOD6CNT mod2 = MOD6CNT_DEFAULTS;

Invoking the computation macro

MOD6CNT_MACRO (mod1);
MOD6CNT_MACRO (mod2);

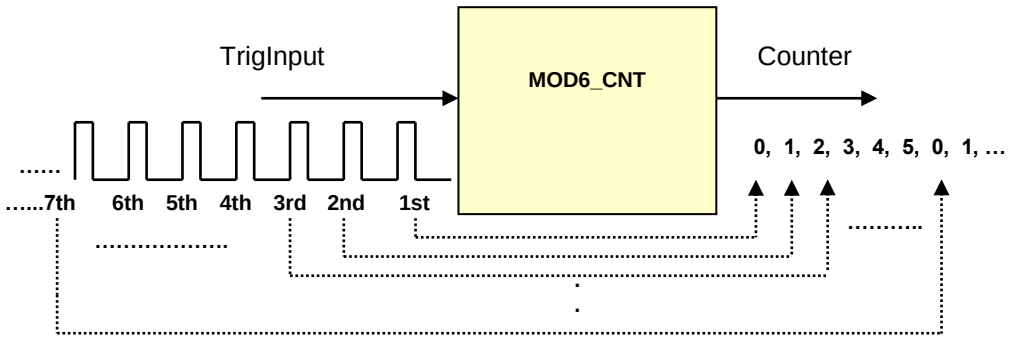
Example

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        mod1.TrigInput = input1;          // Pass inputs to mod1
        mod2.TrigInput = input2;          // Pass inputs to mod2

        MOD6CNT_MACRO (mod1);             // Call compute macro for mod1
        MOD6CNT_MACRO (mod2);             // Call compute macro for mod2

        out1 = mod1.Counter;              // Access the outputs of mod1
        out2 = mod2.Counter;              // Access the outputs of mod2

}
```

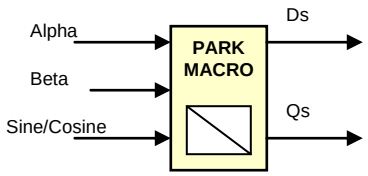
Technical Background

Counter = 0, when 1st trigger pulse occur (*TrigInput* is set to 0x7FFF for the 1st time)
= 1, when 2nd trigger pulse occur(*TrigInput* is set to 0x7FFF for the 2nd time)
= 2, when 3rd trigger pulse occur (*TrigInput* is set to 0x7FFF for the 3rd time)
= 3, when 4th trigger pulse occur (*TrigInput* is set to 0x7FFF for the 4th time)
= 5, when 5th trigger pulse occur (*TrigInput* is set to 0x7FFF for the 5th time)
= 5, when 6th trigger pulse occur (*TrigInput* is set to 0x7FFF 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

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: park.h

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

C Interface

Object Definition

The structure of PARK object is defined by following structure definition

```
typedef struct {  _iq  Alpha;      // Input: stationary d-axis stator variable
                  _iq  Beta;       // Input: stationary q-axis stator variable
                  _iq  Ds;         // Output: rotating d-axis stator variable
                  _iq  Qs;         // Output: rotating q-axis stator variable
                  _iq  Sine;       // Input: sine term
                  _iq  Cosine;     // Input: cosine term

                } PARK;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	Alpha	Direct axis(d) component of the transformed signal	GLOBAL_Q	80000000-7FFFFFFF
	Beta	Quadrature axis(q) component of the transformed signal	GLOBAL_Q	80000000-7FFFFFFF
	Sine	Sine of the phase angle between stationary and rotating frame	GLOBAL_Q	80000000-7FFFFFFF
	Cosine	Cosine of the phase angle between stationary and rotating frame	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Ds	Direct axis(D) component of transformed signal in rotating reference frame	GLOBAL_Q	80000000-7FFFFFFF
	Qs	Quadrature axis(Q) component of transformed signal in rotating reference frame	GLOBAL_Q	80000000-7FFFFFFF

* GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

PARK

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

PARK_DEFAULTS

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

Module Usage

Instantiation

The following example instances two PARK objects

```
PARK  park1, park2;
```

Initialization

To Instance pre-initialized objects

```
PARK park1 = PARK_DEFAULTS;
PARK park2 = PARK_DEFAULTS;
```

Invoking the computation macro

```
PARK_MACRO(park1);
PARK_MACRO(park2);
```

Example

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

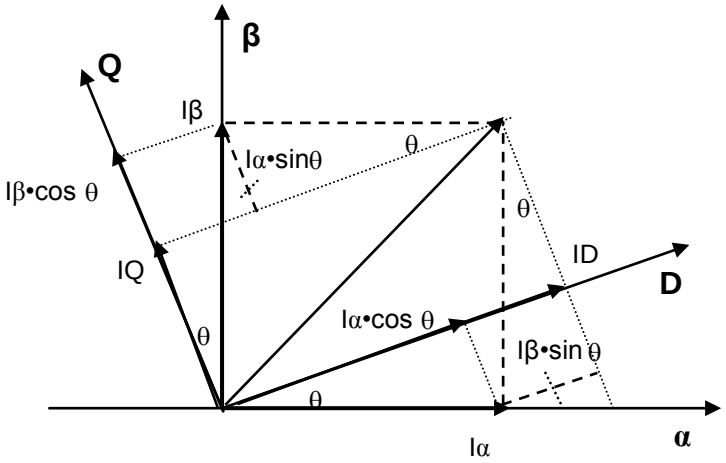
```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        park1.Alpha = ds1;          // Pass inputs to park1
        park1.Beta = qs1;           // Pass inputs to park1
        park1.Angle = ang1;         // Pass inputs to park1

        park2.Alpha = ds2;          // Pass inputs to park2
        park2.Beta = qs2;           // Pass inputs to park2
        park2.Angle = ang2;         // Pass inputs to park2

        PARK_MACRO(park1);          // Call compute macro for park1
        PARK_MACRO(park2);          // Call compute macro for park2

        de1 = park1.Ds;             // Access the outputs of park1
        qe1 = park1.Qs;             // Access the outputs of park1

        de2 = park2.Ds;             // Access the outputs of park2
        qe2 = park2.Qs;             // Access the outputs of park2

}
```

Technical Background

Implements the following equations:

$$\begin{cases} ID = I_\alpha \times \cos\theta + I_\beta \times \sin\theta \\ IQ = -I_\alpha \times \sin\theta + I_\beta \times \cos\theta \end{cases}$$

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:

$$\begin{cases} I_\alpha = I \times \cos(\omega t) \\ I_\beta = I \times \sin(\omega t) \end{cases}$$

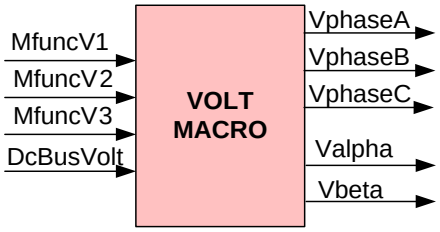
Next, Table 1 shows the correspondence of notations between variables used here and variables used in the program (i.e. park.h). The software module requires that both input and output variables are in per unit values.

	Equation Variables	Program Variables
Inputs	Iα	Alpha
	Iβ	Beta
	sin	Sine
	cos	Cosine
Outputs	ID	Ds
	IQ	Qs

Table 1: Correspondence of notations

Description

This software module calculates three phase voltages impressing to the 3-ph electric motor (i.e., induction or synchronous motor) by 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 in the inverter. In addition, this software module also includes the clarke transformation changing from three phase voltages into two stationary dq-axis phase voltages.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: volt_calc.h

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

C Interface

Object Definition

The structure of PHASEVOLTAGE object is defined by following structure definition

```
typedef struct { _iq  DcBusVolt;          // Input: DC-bus voltage
                 _iq  MfuncV1;            // Input: Modulation voltage phase A
                 _iq  MfuncV2;            // Input: Modulation voltage phase B
                 _iq  MfuncV3;            // Input: Modulation voltage phase C
                 Uint32  OutOfPhase;      // Parameter: Out of Phase adjustment (0 or 1)
                 _iq  VphaseA;            // Output: Phase voltage phase A
                 _iq  VphaseB;            // Output: Phase voltage phase B
                 _iq  VphaseC;            // Output: Phase voltage phase C
                 _iq  Valpha;             // Output: Stationary d-axis phase voltage
                 _iq  Vbeta;              // Output: Stationary q-axis phase voltage
               } PHASEVOLTAGE;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	DcBusVolt	DC-bus voltage	GLOBAL_Q	80000000-7FFFFFFF
	MfuncV1	Switching function of upper switching device 1	GLOBAL_Q	80000000-7FFFFFFF
	MfuncV2	Switching function of upper switching device 2	GLOBAL_Q	80000000-7FFFFFFF
	MfuncV3	Switching function of upper switching device 3	GLOBAL_Q	80000000-7FFFFFFF
Outputs	VphaseA	Line-neutral phase voltage A	GLOBAL_Q	80000000-7FFFFFFF
	VphaseA	Line-neutral phase voltage A	GLOBAL_Q	80000000-7FFFFFFF
	VphaseA	Line-neutral phase voltage A	GLOBAL_Q	80000000-7FFFFFFF
	Valpha	Stationary d-axis phase voltage	GLOBAL_Q	80000000-7FFFFFFF
	Vbeta	Stationary q-axis phase voltage	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

PHASEVOLTAGE

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

PHASEVOLTAGE_DEFAULTS

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

Module Usage

Instantiation
The following example instances two PHASEVOLTAGE objects
PHASEVOLTAGE volt1, volt2;

Initialization
To Instance pre-initialized objects
PHASEVOLTAGE volt1 = PHASEVOLTAGE_DEFAULTS;
PHASEVOLTAGE volt2 = PHASEVOLTAGE_DEFAULTS;

Invoking the computation macro
VOLT_MACRO(volt1);
VOLT_MACRO(volt2);

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        volt1.DcBusVolt = dc_volt1;          // Pass inputs to volt1
        volt1.MfuncV1 = M1_1;                // Pass inputs to volt1
        volt1.MfuncV2 = M2_1;                // Pass inputs to volt1
        volt1.MfuncV3 = M3_1;                // Pass inputs to volt1

        volt2.DcBusVolt = dc_volt2;          // Pass inputs to volt2
        volt2.MfuncV1 = M1_2;                // Pass inputs to volt2
        volt2.MfuncV2 = M2_2;                // Pass inputs to volt2
        volt2.MfuncV3 = M3_2;                // Pass inputs to volt2

        VOLT_MACRO(volt1);                    // Call compute macro for volt1
        VOLT_MACRO(volt2);                    // Call compute macro for volt2

        Vd1 = volt1.Valpha;                   // Access the outputs of volt1
        Vq1 = volt1.Vbeta;                    // Access the outputs of volt1

        Vd2 = volt2.Valpha;                   // Access the outputs of volt2
        Vq2 = volt2.Vbeta;                    // Access the outputs of volt2
}
```

Technical Background

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 as the Y connection without a neutral return path (or 3-ph, 3-wire system). The overall system can be shown in Figure 1.

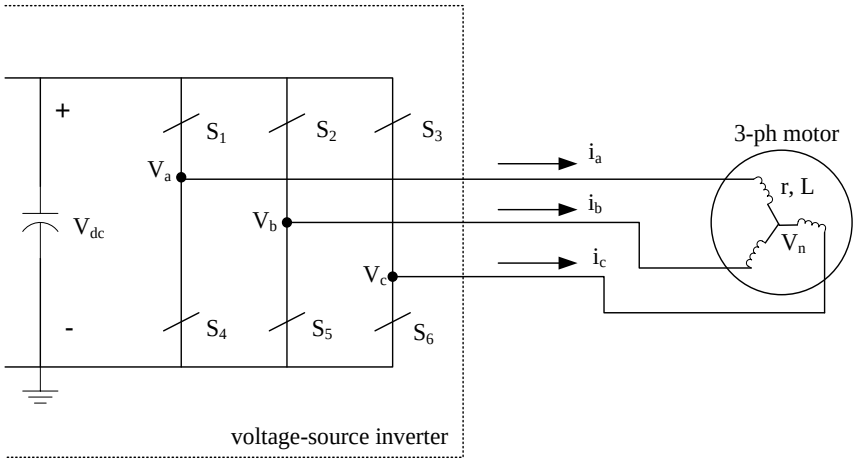


Figure 1: Voltage-source inverter with a 3-ph electric motor

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

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

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

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

Summing these three phase voltages, yields

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

Without a neutral return path, according to KCL, i.e., $i_a + i_b + i_c = 0$, and the back emfs are balanced and symmetrical due to the 3-ph winding structures, i.e., $e_a + e_b + e_c = 0$, so (4) becomes

$$V_{an} + V_{bn} + V_{cn} = 0 \quad (5)$$

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

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

Now three phase voltages can be calculated as

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

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

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

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 the following relation.

$$V_a = S_1 V_{dc} \quad (10)$$

$$V_b = S_2 V_{dc} \quad (11)$$

$$V_c = S_3 V_{dc} \quad (12)$$

where S_1, S_2, S_3 = either 0 or 1, and $S_4 = 1\text{-}S_1$, $S_5 = 1\text{-}S_2$, and $S_6 = 1\text{-}S_3$. (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 follows:

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

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

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

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 changing from three phase voltages (V_{an}, V_{bn}, and V_{cn}) to the stationary dq-axis phase voltages (V_{ds}^s, and V_{qs}^s) are applied by using the following relationship. Because of the balanced system (5), V_{cn} is not used in clarke transformation.

$$V_{ds}^s = V_{an} \quad (17)$$

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

Figure 2 depicts the abc-axis and stationary dq-axis components for the stator voltages of motor. Notice that the notation of the stationary dq-axis is sometimes used as the stationary $\alpha\beta$-axis, accordingly.

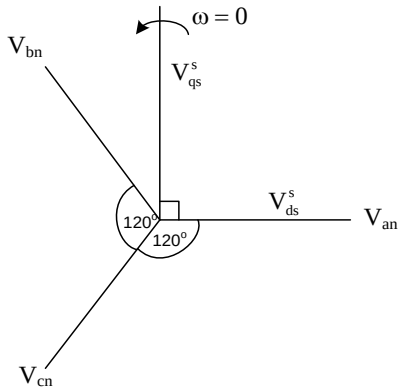


Figure 2: The abc-axis and stationary dq-axis components of the stator phase voltages

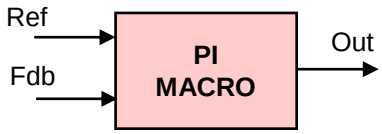
Next, Table 1 shows the correspondence of notations between variables used here and variables used in the program (i.e., volt_calc.h). The software module requires that both input and output variables are in per unit values.

	Equation Variables	Program Variables
Inputs	S_1	MfuncV1
	S_2	MfuncV2
	S_3	MfuncV3
	V_{dc}	DcBusVolt
Outputs	V_{an}	VphaseA
	V_{bn}	VphaseB
	V_{cn}	VphaseC
	V^s_{ds}	Valpha
	V^s_{qs}	Vbeta

Table 1: Correspondence of notations

Description

This module implements a simple 32-bit digital PI controller with anti-windup correction.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: pi.h

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

C Interface

Object Definition

The structure of PI object is defined by following structure definition

```
typedef struct {
        Ref     // Input: reference set-point
        Fbk     // Input: feedback
        Out     // Output: controller output
        Kp      // Parameter: proportional loop gain
        Ki      // Parameter: integral gain
        Umax    // Parameter: upper saturation limit
        Umin    // Parameter: lower saturation limit
        up      // Data: proportional term
        ui      // Data: integral term
        v1      // Data: pre-saturated controller output
        i1      // Data: integrator storage: ui(k-1)
        w1      // Data: saturation record: [u (k-1) - v(k-1)]
       } PI;
```

Module Terminal Variables/Macros

Item	Name	Description	Format*	Range(Hex)
Input	Ref	Reference input	GLOBAL_Q	80000000-7FFFFFFF
	Fbk	Feedback input	GLOBAL_Q	80000000-7FFFFFFF
	UMax	Maximum PI32 module output	GLOBAL_Q	80000000-7FFFFFFF
	UMin	Minimum PI32 module output	GLOBAL_Q	80000000-7FFFFFFF
Output	Out	PI Output (Saturated)	GLOBAL_Q	80000000-7FFFFFFF
PI parameter	Kp	Proportional gain	GLOBAL_Q	80000000-7FFFFFFF
	Ki	Integral gain	GLOBAL_Q	80000000-7FFFFFFF
Internal	up	Proportional term	GLOBAL_Q	80000000-7FFFFFFF
	ui	Integral term	GLOBAL_Q	80000000-7FFFFFFF
	v1	Pre-saturated controller output	GLOBAL_Q	80000000-7FFFFFFF
	i1	Integrator storage	GLOBAL_Q	80000000-7FFFFFFF
	w1	Saturation record	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

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

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

Module Usage

Instantiation
The following example instances two PI objects
```
PI  pi1, pi2;
```

Initialization
To Instance pre-initialized objects
```
PI pi1 = PI_DEFAULTS;
PI pi2 = PI_DEFAULTS;
```

Invoking the computation macro
```
PI_MACRO(pi1);
PI_MACRO(pi2);
```

Example

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

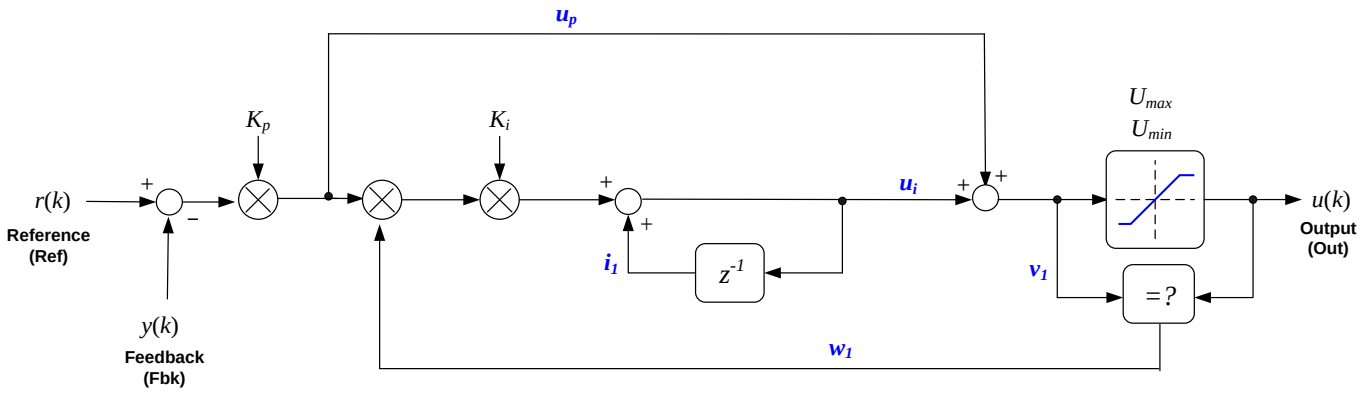
```
 /* Instance the PI module */
PI   pi1=PI_DEFAULTS;
PI   pi2=PI_DEFAULTS;

main()
{

        pi1.Kp = _IQ(0.5);          // Pass _iq parameters to pi1
        pi1.Ki  = _IQ(0.001);       // Pass _iq parameters to pi1
        pi1.Umax =_IQ(0.9);         // Pass _iq parameters to pi1
        pi1.Umin  =_IQ(-0.9);       // Pass _iq parameters to pi1

        pi2.Kp = _IQ(0.8);          // Pass _iq parameters to pi2
        pi2.Ki  = _IQ(0.0001);      // Pass _iq parameters to pi2
        pi1.Umax =_IQ(0.9);         // Pass _iq parameters to pi2
        pi1.Umin  =_IQ(-0.9);       // Pass _iq parameters to pi2


}

void interrupt periodic_interrupt_isr()
{
        pi1.Ref = input1_1;         // Pass _iq inputs to pi1
        pi1.Fdb = input1_2;         // Pass _iq inputs to pi1
        pi2.Ref = input2_1;         // Pass _iq inputs to pi2
        pi2.Fdb = input2_2;         // Pass _iq inputs to pi2

        PI_MACRO(pi1);              // Call compute macro for pi1
        PI_MACRO(pi2);              // Call compute macro for pi2

        output1 = pi1.Out;          // Access the output of pi1
        output2 = pi2.Out;          // Access the output of pi2
}
```

Technical Background

The PI_cntl module implements a basic summing junction and P+I control law with the following features:

- Programmable output saturation
- Independent reference weighting on proportional path
- Anti-windup integrator reset

The PI controller is a sub-set of the PID controller. All input, output and internal data is in I8Q24 fixed-point format. A block diagram of the internal controller structure is shown below.



a) Proportional path

The proportional path is a direct connection between the error term and a summing junction with the integral path. The error term is:

$$u_p(k) = e(k) = K_p[r(k) - y(k)] \quad \text{.....(1)}$$

b) Integral path

The integral path consists of a discrete integrator which is pre-multiplied by a term derived from the output module. The term w1 is either zero or one, and provides a means to disable the integrator path when output saturation occurs. This prevents the integral term from "winding up" and improves the response time on recovery from saturation. The integrator law used is based on a backwards approximation.

$$u_i(k) = u_i(k-1) + K_i e(k) \quad \text{.....(2)}$$

c) Output path

The output path contains a summing block to sum the proportional and integral controller terms. The result is then saturated according to user programmable upper and lower limits to give the controller output.

The pre-and post-saturated terms are compared to determine whether saturation has occurred, and if so, a zero or one result is produced which is used to disable the integral path (see above). The output path law is defined as follows.

$$v_1(k) = u_p(k) + u_i(k) \quad \text{.....(3)}$$

$$u(k) = \begin{cases} U_{\max} : v_1(k) > U_{\max} \\ U_{\min} : v_1(k) < U_{\min} \\ v_1(k) : U_{\min} < v_1(k) < U_{\max} \end{cases} \quad \text{.....(4)}$$

$$w_1(k) = \begin{cases} 0 : v_1(k) \neq u(k) \\ 1 : v_1(k) = u(k) \end{cases} \quad \text{.....(5)}$$

Tuning the P+I controller

Default values for the controller coefficients are defined in the macro header file which apply unity gain through the proportional path, and disable both integral and derivative paths. A suggested general technique for tuning the controller is now described.

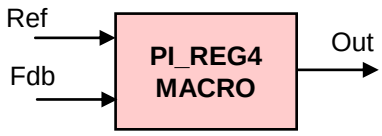
Step 1. Ensure integral is set to zero and proportional gain set to one.

Step 2. Gradually adjust proportional gain variable (K_p) while observing the step response to achieve optimum rise time and overshoot compromise.

Step 3. If necessary, gradually increase integral gain (K_i) to optimize the return of the steady state output to nominal. The controller will be very sensitive to this term and may become unstable so be sure to start with a very small number. Integral gain will result in an increase in overshoot and oscillation, so it may be necessary to slightly decrease the K_p term again to find the best balance. Note that if the integral gain is used then set to zero, a small residual term may persist in u_i.

Description

This module implements a simple 32-bit digital PI controller with anti-windup correction. Functionally, it is similar to PI module described above and uses the same object described above, the difference change in the path of P control such that K_p can be set to zero unlike the previous module. Refer to the previous section for object definitions and technical reference below for implementation details



Availability

C interface version

Module Properties

Type: Target Independent

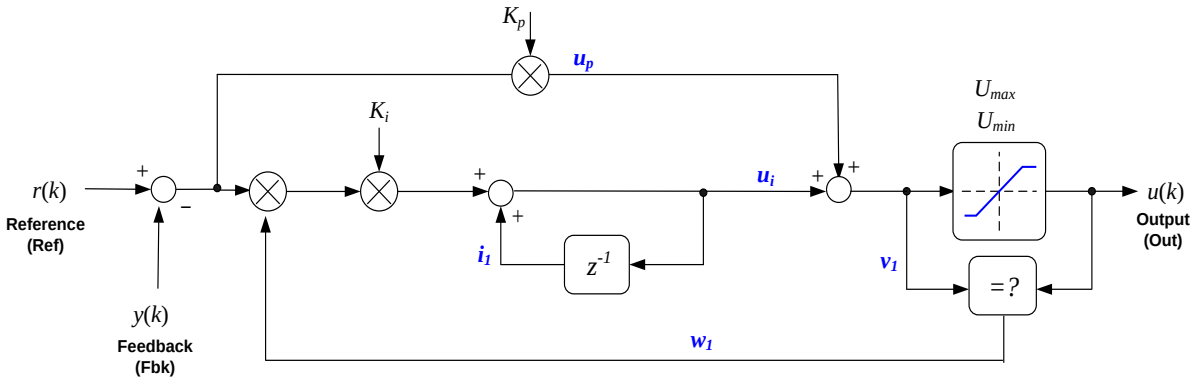
Target Devices: 28x Fixed or Floating Point

C Version File Names: pi_reg4.h

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

Technical Background

The PI_cntl module implementation resembles the previous one except that K_p is taken away from the forward path and is positioned in parallel to integral path as shown below.



a) Proportional path

The proportional path is a direct connection between the error term and a summing junction with the integral path. The error term is:

$$e(k) = r(k) - y(k) \quad \text{.....(2)}$$

$$u_p(k) = K_p e(k) \quad \text{.....(2)}$$

b) Integral path

The integral path consists of a discrete integrator which is pre-multiplied by a term derived from the output module. The term w1 is either zero or one, and provides a means to disable the integrator path when output saturation occurs. This prevents the integral term from "winding up" and improves the response time on recovery from saturation. The integrator law used is based on a backwards approximation.

$$u_i(k) = u_i(k-1) + K_i[r(k) - y(k)] \quad \text{.....(3)}$$

c) Output path

The output path contains a summing block to sum the proportional and integral controller terms. The result is then saturated according to user programmable upper and lower limits to give the controller output.

The pre-and post-saturated terms are compared to determine whether saturation has occurred, and if so, a zero or one result is produced which is used to disable the integral path (see above). The output path law is defined as follows.

$$v_1(k) = \lfloor u_p(k) + u_i(k) \rfloor \quad \text{.....(4)}$$

$$u(k) = \begin{cases} U_{\max} : v_1(k) > U_{\max} \\ U_{\min} : v_1(k) < U_{\min} \\ v_1(k) : U_{\min} < v_1(k) < U_{\max} \end{cases} \quad \text{.....(5)}$$

$$w_1(k) = \begin{cases} 0 : v_1(k) \neq u(k) \\ 1 : v_1(k) = u(k) \end{cases} \quad \text{.....(6)}$$

Tuning the P+I controller

Default values for the controller coefficients are defined in the macro header file which apply unity gain through the proportional path, and disable both integral and derivative paths. A suggested general technique for tuning the controller is now described.

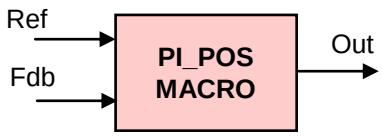
Step 1. Ensure integral is set to zero and proportional gain set to one.

Step 2. Gradually adjust proportional gain variable (K_p) while observing the step response to achieve optimum rise time and overshoot compromise.

Step 3. If necessary, gradually increase integral gain (K_i) to optimize the return of the steady state output to nominal. The controller will be very sensitive to this term and may become unstable so be sure to start with a very small number. Integral gain will result in an increase in overshoot and oscillation, so it may be necessary to slightly decrease the K_p term again to find the best balance. Note that if the integral gain is used then set to zero, a small residual term may persist in u_i.

Description

This module implements a generic, simple 32-bit digital PI controller with anti-windup correction, exactly same as in the previous section on PI controller, except for the difference in error handling. Refer to the previous section for implementation details of PI controller, and technical literature below for error handling.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: pi.h

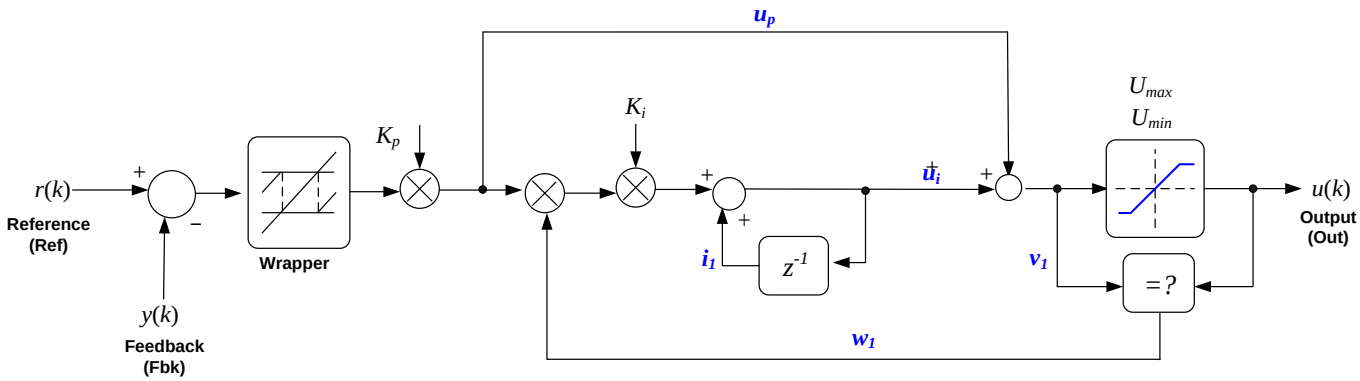
IQmath library files for C: IQmathLib.h, IQmath.lib

Technical Background

The PI_POS_cntl module implements a basic summing junction and P+I control law with the following features:

- Programmable output saturation
- Independent reference weighting on proportional path
- Anti-windup integrator reset
- Position error wrap around to fit within –π and +π

The PI controller is a sub-set of the PID controller. All input, output and internal data is in I8Q24 fixed-point format. A block diagram of the internal controller structure is shown below.



d) Position Error Wrapper

This block is the only difference between the original PI macro and the PI_POS macro. Since the reference and feedback for PI_POS macro are angles, it should be wrapped within - π and +π, otherwise, it will result in erratic behavior of the loop. Consider an error value of, say, 3π/2. This will pull the controller output in positive polarity. Actually this error should be treated as –π/2 which would have pulled it in negative polarity.

Description

This module implements a generic, simple 32-bit digital PI controller with anti-windup correction, exactly same as in the previous section on PI_POS controller but treated as in PI_REG4. Refer to the previous section for implementation details of PI_POS controller, and technical literature below for block diagram.

Availability

C interface version

Module Properties

Type: Target Independent

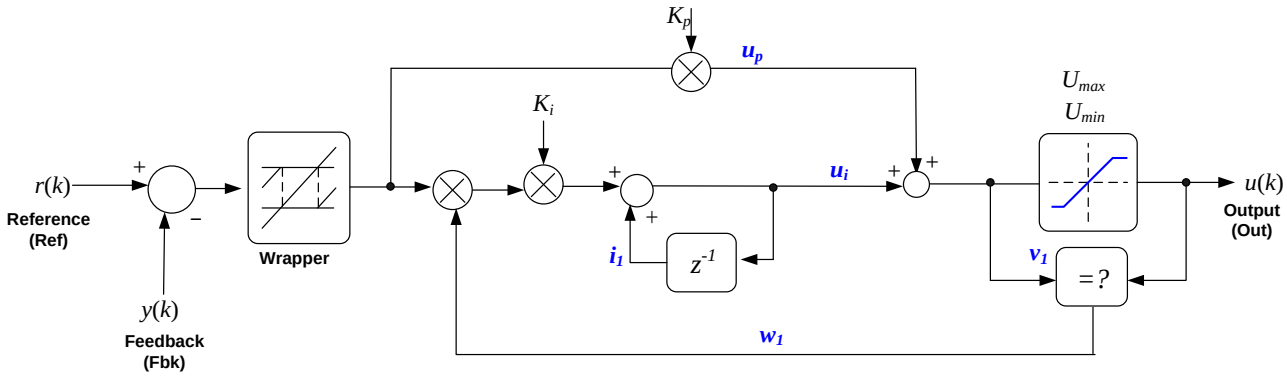
Target Devices: 28x Fixed or Floating Point

C Version File Names: pi_reg4.h

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

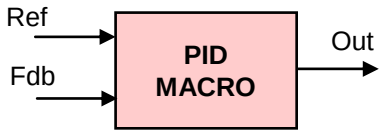
Technical Background

The block diagram of the internal controller structure is shown below. It is similar to the PI_POS module but for the K_p path which gives freedom to set K_p to zero.



Description

This module implements a 32-bit digital PID controller with anti-windup correction.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: pid_grando.h

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

C Interface

Object Definition

The structure of PI object is defined by following structure definitions

```
typedef struct {
                  _iq Ref          // Input: reference set-point
                  _iq Fbk          // Input: feedback
                  _iq Out          // Output: controller output
                  _iq c1           // Internal: derivative filter coefficient
                  _iq c2           // Internal: derivative filter coefficient
                  _iq Iae          // Output: performance index
                  _iq Err          // Internal: servo error

                 } PID_TERMINALS;

typedef struct {
                   _iq  Kr;        // Parameter: reference set-point weighting
                   _iq  Kp;        // Parameter: proportional loop gain
                   _iq  Ki;        // Parameter: integral gain
                   _iq  Kd;        // Parameter: derivative gain
                   _iq  Km;        // Parameter: derivative weighting
                   _iq  Umax;      // Parameter: upper saturation limit
                   _iq  Umin;      // Parameter: lower saturation limit
                   _iq Kiae        // Parameter: IAE scaling
                  } PID_PARAMETERS;

typedef struct {
                   _iq  up;        // Data: proportional term
                   _iq  ui;        // Data: integral term
                   _iq  ud;        // Data: derivative term
                   _iq  v1;        // Data: pre-saturated controller output
                   _iq  i1;        // Data: integrator storage: ui(k-1)
                   _iq  d1;        // Data: differentiator storage: ud(k-1)
                   _iq  d2;        // Data: differentiator storage: d2(k-1)
                   _iq  w1;        // Data: saturation record: [u(k-1) - v(k-1)]
                  } PID_DATA;


typedef struct {
                  PID_TERMINALS       term;
                  PID_PARAMETERS      param;
                  PID_DATA            data;
                 } PID_CONTROLLER;
```

Special Constants and Data types

PID

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

PID_DEFAULTS

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

Module Usage

Instantiation

The following example instances PID object

```
PID pid1;
```

Initialization

To Instance pre-initialized objects

```
PID pid1 = { PID_TERM_DEFAULTS, PID_PARAM_DEFAULTS, PID_DATA_DEFAULTS };
```

Invoking the computation macro

```
PID_MACRO(pid1);
```

Example

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

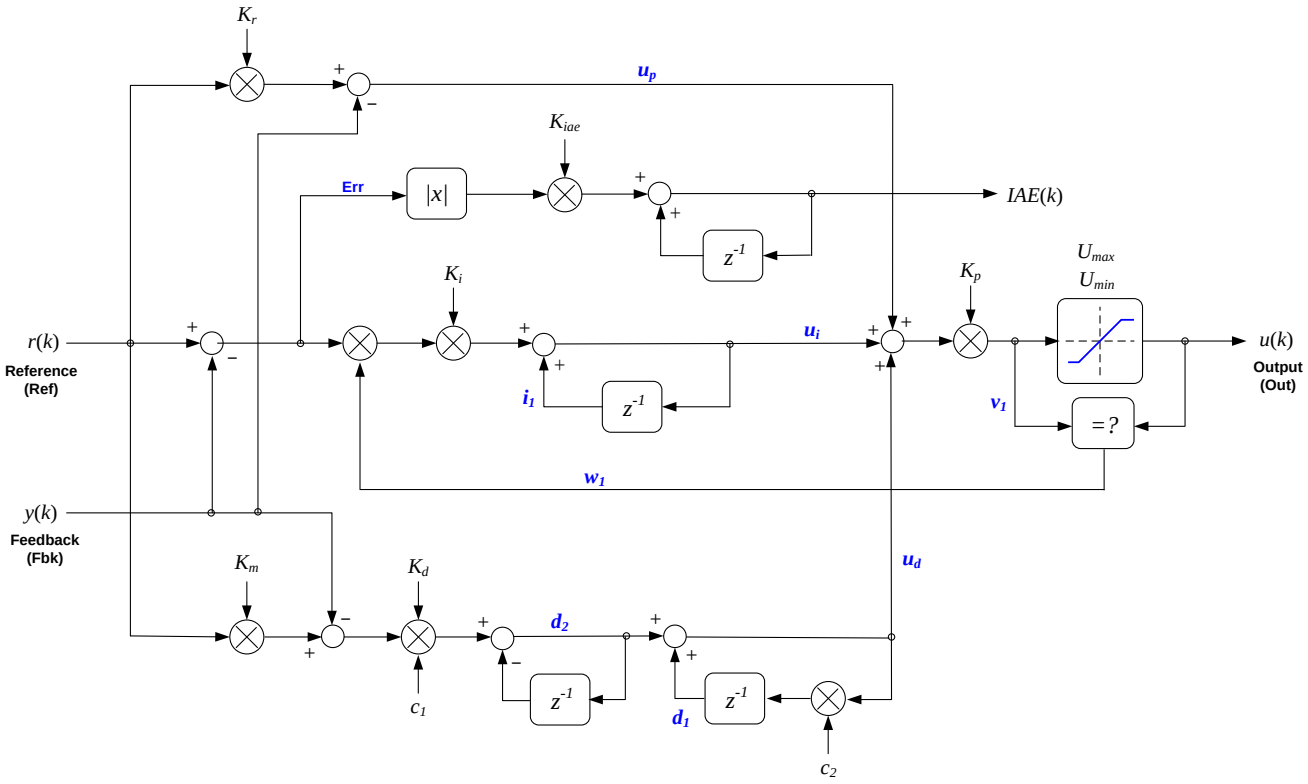
```
 /* Instance the PID module */

PID   pid1={ PID_TERM_DEFAULTS, PID_PARAM_DEFAULTS, PID_DATA_DEFAULTS };

main()
{

        pid1.param.Kp = _IQ(0.5);
        pid1.param.Ki  = _IQ(0.005);
        pid1.param.Kd = _IQ(0);
        pid1.param.Kr  = _IQ(1.0);
        pid1.param.Km =_IQ(1.0);
        pid1.param.Umax= _IQ(1.0);
        pid1.param.Umin= _IQ(-1.0);

}

void interrupt periodic_interrupt_isr()
{
        pid1.Ref = input1_1;            // Pass _iq inputs to pid1
        pid1.Fbk = input1_2;            // Pass _iq inputs to pid1

        PID_MACRO(pid1);                // Call compute macro for pid1

        output1 = pid1.Out;             // Access the output of pid1

}
```

Technical Background

The PID Grando module implements a basic summing junction and PID control law with the following features:

- Programmable output saturation
- Independent reference weighting on proportional path
- Independent reference weighting on derivative path
- Anti-windup integrator reset
- Programmable derivative filter
- Transient performance measurement

PID Grando is an example of a PID structure often called "standard" form, in which proportional gain is applied after the three controller paths have been summed. This contrasts with the "parallel" PID form, in which P, I, and D gains are applied in separate paths. All input, output and internal data is in I8Q24 fixed-point format. A block diagram of the internal controller structure is shown below.



Block diagram of the internal controller structure

a) Proportional path

The servo error is the difference between the reference input and the feedback input. Proportional gain is usually applied directly to servo error, however a feature of the Grando controller is that sensitivity of the proportional path to the reference input can be weighted differently to that of the feedback input. This provides an additional degree of freedom when tuning the controller. The proportional control law is:

$$u_p(k) = K_r r(k) - y(k) \quad \ldots\ldots (3)$$

Note that "proportional" gain is applied to the sum of all three terms and will be described in section d).

b) Integral path

The integral path consists of a discrete integrator which is pre-multiplied by a scalar gain (K_i) and a term derived from the output saturation module. The term w_1 is either zero or one, and provides a means to disable the integrator path when output saturation occurs. This prevents the integrator from "winding up" and improves the recovery time following saturation in the control loop. The integrator law used in Grando is based on a backwards approximation.

$$u_i(k) = u_i(k-1) + w_1(k) K_i [r(k) - y(k)] \quad \ldots\ldots (2)$$

c) Derivative path

The derivative term is a backwards approximation of the difference between the current and previous servo error. The input is the difference between the reference and feedback terms, and like the proportional term, the reference path can be weighted independently to provide an additional variable for tuning.

A first order digital filter is applied to the derivative term to reduce nose amplification at high frequencies. Filter cutoff frequency is determined by two coefficients (c_1 & c_2). The derivative law is shown below.

$$e(k) = K_m r(k) - y(k) \quad \ldots\ldots (3)$$

$$u_d(k) = K_d [c_2 u_i(k-1) + c_1 e(k) - c_1 e(k-1)] \quad \ldots\ldots (4)$$

Filter coefficients are based on the cut-off frequency (a) in Hz and sample period (T) in seconds as follows:

$$c_1 = a \quad \ldots\ldots (5)$$

$$c_2 = 1 - c_1 T \quad \ldots\ldots (6)$$

d) Output path

The output path contains a multiplying term (K_p) which acts on the sum of the three controller parts. The result is then saturated according to user programmable upper and lower limits to give the output term.

The pre-and post-saturated terms are compared to determine whether saturation has occurred, and if so, a zero value is assigned to the variable $w_1(k)$ which is used to disable the integral path (see above). The output path law is defined as follows.

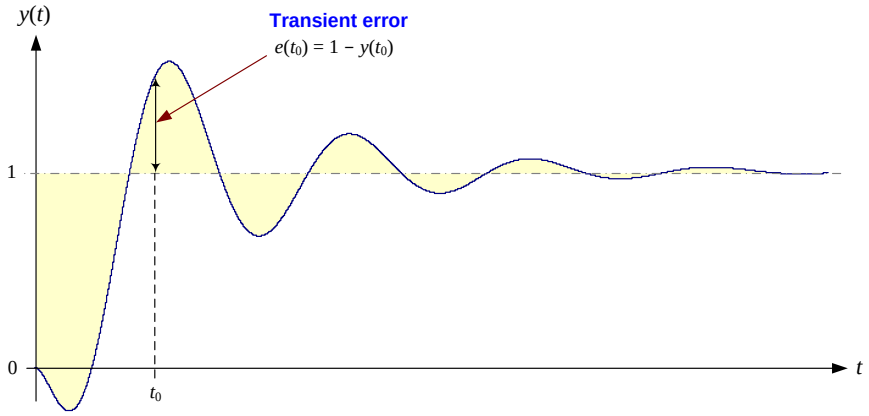
$$v_1(k) = K_p\left[u_p(k) + u_i(k) + u_d(k)\right] \quad \ldots\ldots (7)$$

$$u(k) = \begin{cases} U_{\max} : v_1(k) > U_{\max} \\ U_{\min} : v_1(k) < U_{\min} \\ v_1(k) : U_{\min} < v_1(k) < U_{\max} \end{cases} \quad \ldots\ldots (8)$$

$$w_1(k) = \begin{cases} 0 : v_1(k) \neq u(k) \\ 1 : v_1(k) = u(k) \end{cases} \quad \ldots\ldots (9)$$

e) Performance measurement

The Grando controller contains a single line of code which implements an IAE algorithm (Integral of Absolute Error). This integrates the absolute difference between the input reference and feedback (i.e. servo error), and can be used to tune transient performance by subjecting the loop to a step change of input reference and allowing the IAE term to integrate over a fixed time.



A small value of IAE indicates small absolute difference between the input reference and the measured output. Controller parameters may be adjusted iteratively to minimise measured IAE.

The IAE measurement is implemented as follows.

$$IAE(k) = IAE(k-1) + K_{iae}|r(k) - y(k)| \quad \text{..... (10)}$$

It is the task of the user code to reset and enable the IAE measurement (see below). The feature may be enabled and disabled by setting the K_{iae} term to one or zero respectively. The IAE output may be reset by setting the Iae term to zero. The K_{iae} term may be set to a value smaller than one if integrator overflow is an issue. The user may comment out the last line of the PID Grando macro to reduce cycle count if IAE is not required.

Tuning the controller

Default values for the controller coefficients are defined in the macro header file which apply unity gain through the proportional path, and disable IAE, integral and derivative paths. A suggested general technique for tuning the controller is now described.

Steps 1-4 are based on tuning a transient produced either by a step change in either load or reference set-point.

Step 1. Ensure integral and derivative gains are set to zero. Ensure also the reference weighting coefficients (K_r & K_m) are set to one.

Step 2. Gradually adjust proportional gain variable (K_p) while observing the step response to achieve optimum rise time and overshoot compromise.

Step 3. If necessary, gradually increase integral gain (K_i) to optimize the return of the steady state output to nominal. The controller will be very sensitive to this term and may become unstable so be sure to start with a very small number. Integral gain will result in an increase in overshoot and oscillation, so it may be necessary to slightly decrease the K_p term again to find the best balance. Note that if the integral gain is used then set to zero, a small residual term may persist in u_i.

Step 4. If the transient response exhibits excessive oscillation, this can sometimes be reduced by applying a small amount of derivative gain. To do this, first ensure the coefficients c_1 & c_2 are set to one and zero respectively. Next, slowly add a small amount of derivative gain (K_d).

Steps 5 & 6 only apply in the case of tuning a transient set-point. In the regulator case, or where the set-point is fixed and tuning is conducted against changing load conditions, they are not useful.

Step 5. Overshoot and oscillation following a set-point transient can sometimes be improved by lowering the reference weighting in the proportional path. To do this, gradually reduce the K_r term from its nominal unity value to optimize the transient. Note that this will change the loop sensitivity to the input reference, so the steady state condition will change unless integral gain is used.

Step 6. If derivative gain has been applied, transient response can often be improved by changing the reference weighting, in the same way as step 6 except that in the derivative case steady state should not be affected. Slowly reduce the K_m variable from it's nominal unity value to optimize overshoot and oscillation. Note that in many cases optimal performance is achieved with a reference weight of zero in the derivative path, meaning that the differential term acts on purely the feedback, with no contribution from the input reference. This can help to avoid derivative "kick" which occurs following a sudden change in input reference.

The derivative path introduces a term which has a frequency dependent gain. At higher frequencies, this can cause noise amplification in the loop which may degrade servo performance. If this is the case, it is possible to filter the derivative term using a first order digital filter in the derivative path. Steps 7 & 8 describe the derivative filter.

Step 7. Select a filter roll-off frequency in radians/second. Use this in conjunction with the system sample period (T) to calculate the filter coefficients c_1 & c_2 (see equations 5 & 6).

Step 8. Note that the c_1 coefficient will change the derivative path gain, so adjust the value of K_d to compensate for the filter gain. Repeat steps 5 & 6 to optimize derivative path gain.

Tuning with IAE

The IAE measurement feature may be useful in determining optimal controller settings based on transient response. The user code should reset and enable IAE just before applying a transient change as follows:

```
// reset and enable IAE
pid1.term.Iae = _IQ(0.0);
pid1.Param.Kiae = _IQ(1.0);
```

After a fixed time, the IAE integrator should be disabled and the measurement read as follows:

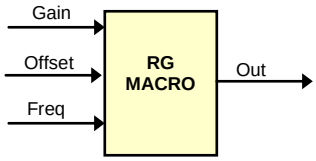
```
// disable IAE integrator
pid1.param.Kiae = _IQ(0.0);
IAE = pid1.term.Iae;
```

The user should ensure that the integration period is sufficiently long to allow transient effects to dissipate and should remain fixed between iterative tests for comparisons to be valid. Controller settings may be adjusted using steps 1 to 8 above to minimise measured IAE.

Saturation & Integrator Wind-up

The Grando controller includes a saturation block to limit the range of the control effort, $u(k)$. If the output saturates, the integrator is disabled to prevent a phenomenon known as "wind-up". In cases where saturation may occur in other parts of the control loop, user code should disable integral action by temporarily setting the integrator gain (K_i) to zero when saturation occurs, and restoring it once saturation has been cleared.

Description This module generates ramp output of adjustable gain, frequency and dc offset.



Availability C interface version

Module Properties **Type:** Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: rampgen.h

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

C Interface

Object Definition

The structure of RAMPGEN object is defined by following structure definition

```
typedef struct { _iq  Freq;          // Input: Ramp frequency
                 _iq  StepAngleMax;  // Parameter: Maximum step angle
                 _iq  Angle;         // Variable: Step angle
                 _iq  Gain;          // Input: Ramp gain
                 _iq  Out;           // Output: Ramp signal
                 _iq  Offset;        // Input: Ramp offset
                 } RAMPGEN;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	Freq	Ramp frequency	GLOBAL_Q	80000000-7FFFFFFF
	Gain	Ramp gain	GLOBAL_Q	80000000-7FFFFFFF
	Offset	Ramp offset	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Out	Ramp signal	GLOBAL_Q	80000000-7FFFFFFF
RAMPGEN parameter	StepAngleMax	sv_freq_max = fb*T	GLOBAL_Q	80000000-7FFFFFFF
Internal	Angle	Step angle	GLOBAL_Q	80000000-7FFFFFFF

* GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

RAMPGEN

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

RAMPGEN_DEFAULTS

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

Module Usage

Instantiation
The following example instances two RAMPGEN objects
RAMPGEN rg1, rg2;

Initialization
To Instance pre-initialized objects
RAMPGEN rg1 = RAMPGEN_DEFAULTS;
RAMPGEN rg2 = RAMPGEN_DEFAULTS;

Invoking the computation macro
RG_MACRO(rg1);
RG_MACRO(rg2);

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        rg1.Freq = freq1;               // Pass inputs to rg1
        rg1.Gain = gain1;               // Pass inputs to rg1
        rg1.Offset = offset1;           // Pass inputs to rg1

        rg2.Freq = freq2                // Pass inputs to rg2
        rg2.Gain = gain2;               // Pass inputs to rg2
        rg2.Offset = offset2;           // Pass inputs to rg2

        RC_MACRO(rg1);                  // Call compute macro for rg1
        RC_MACRO(rg2);                  // Call compute macro for rg1

        out1 = rg1.Out;                 // Access the outputs of rg1
        out2 = rg2.Out;                 // Access the outputs of rg1
}
```

Technical Background

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 28xx. One such variable, called *StepAngleMax* (a data memory location in 28xx) in this implementation, is used as a variable to determine the minimum period (1/frequency) of the ramp signal. Adding a fixed step value to the *Angle* variable causes the value in *Angle* to cycle at a constant rate.

$$Angle = Angle + StepAngleMax \times Freq$$

At the end limit the value in *Angle* simply wraps around and continues at the next modulo value given by the step size.

For a given step size, the frequency of the ramp output (in Hz) is given by:

$$f = \frac{\text{Step Angle} \times fs}{2^m}$$

where f_s = sampling loop frequency in Hz and *m* = # bits in the auto wrapper variable *Angle*.

From the above equation it is clear that a *StepAngle* value of 1 gives a frequency of 0.3052Hz when m=16 and f_S=20kHz. This defines the frequency setting resolution of the

For IQmath implementation, the maximum step size in per-unit, *StepAngleMax*, for a given base frequency, fb and a defined GLOBAL_Q number is therefore computed as follows:

$$\text{StepAngleMax} = f_b \times T_s \times 2^{GLOBAL\_Q}$$

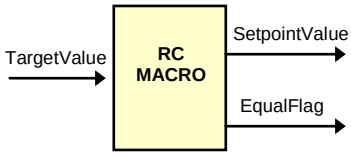
Equivalently, by using _IQ() function for converting from a floating-point number to a _iq number, the *StepAngleMax* can also be computed as

$$\text{StepAngleMax} = \_IQ(f_b \times T_s)$$

where Ts is the sampling period (sec).

Description

This module implements a ramp up and ramp down macro. The output flag variable EqualFlag is set to 7FFFFFFFh when the output variable SetpointValue equals the input variable TargetValue.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: rmp_cntl.h

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

C Interface

Object Definition

The structure of RMPCNTL object is defined by following structure definition

```
typedef struct { _iq  TargetValue;          // Input: Target input
                 Uint32  RampDelayMax;      // Parameter: Maximum delay rate (Q0)
                  _iq  RampLowLimit;        // Parameter: Minimum limit
                  _iq  RampHighLimit;       // Parameter: Maximum limit
                 Uint32 RampDelayCount;     // Variable: Incremental delay (Q0)
                  _iq  SetpointValue;       // Output: Target output
                 Uint32  EqualFlag;         // Output: Flag output (Q0)
                } RMPCNTL;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	TargetValue	Target input	GLOBAL_Q	80000000-7FFFFFFF
Outputs	SetpointValue	Target output	GLOBAL_Q	80000000-7FFFFFFF
	EqualFlag	Flag output	Q0	80000000-7FFFFFFF
RMP_CNTL parameter	RampDelayMax	Maximum delay rate	Q0	80000000-7FFFFFFF
	RampLowLimit	Minimum limit	GLOBAL_Q	80000000-7FFFFFFF
	RampHighLimit	Maximum limit	GLOBAL_Q	80000000-7FFFFFFF
Internal	RampDelayCount	Incremental delay	Q0	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

RMPCNTL

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

RMPCNTL_DEFAULTS

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

Module Usage

Instantiation
The following example instances two RMPCNTL objects
RMPCNTL rc1, rc2;

Initialization
To Instance pre-initialized objects
RMPCNTL rc1 = RMPCNTL_DEFAULTS;
RMPCNTL rc2 = RMPCNTL_DEFAULTS;

Invoking the computation macro
RC_MACRO(rc1);
RC_MACRO(rc2);

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        rc1.TargetValue = target1;          // Pass inputs to rc1
        rc2.TargetValue = target2;          // Pass inputs to rc2

        RC_MACRO(rc1);                      // Call compute macro for rc1
        RC_MACRO(rc2);                      // Call compute macro for rc2

        out1 = rc1.SetpointValue;           // Access the outputs of rc1
        out2 = rc2.SetpointValue;           // Access the outputs of rc2
}
```

Technical Background

This software module implements the following equations:

Case 1: When *TargetValue* > *SetpointValue*

SetpointValue = *SetpointValue*+ _IQ(0.0000305), for t = n **.** Td, n = 1, 2, 3…
and (*SetpointValue* + _IQ(0.0000305))< *RampHighLimit*
= *RampHighLimit* , for (*SetpointValue* + _IQ(0.0000305))> *RampHighLimit*

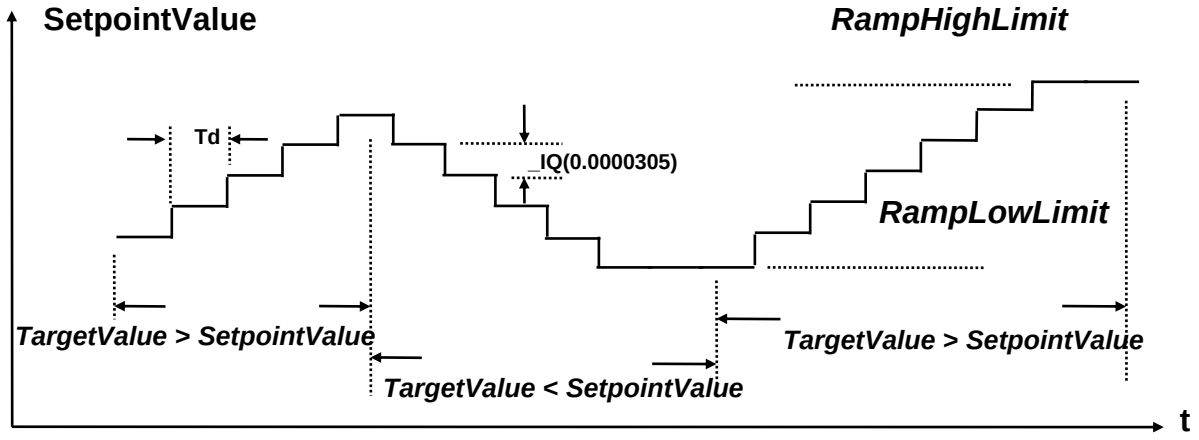
where, Td = *RampDelayMax* **.** Ts
Ts = Sampling time period

Case 2: When *TargetValue* < *SetpointValue*

SetpointValue = *SetpointValue* - _IQ(0.0000305), for t = n **.** Td, n = 1, 2, 3…..
and (*SetpointValue* - _IQ(0.0000305))> *RampLowLimit*
= *RampLowLimit* , for (*SetpointValue* - _IQ(0.0000305))<*RampLowLimit*

where, Td = *RampDelayMax* **.** Ts
Ts = Sampling time period

Note that *TargetValue* and *SetpointValue* variables are in _iq format.

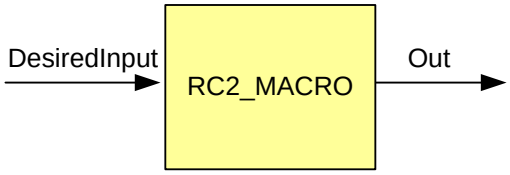


Example:
SetpointValue=0(initial value), TargetValue=1000(user specified),
RampDelayMax=500(user specified), sampling loop time period Ts=0.000025 Sec.
This means that the time delay for each ramp step is Td=500x0.000025=0.0125 Sec. Therefore, the total ramp time will be Tramp=1000x0.0125 Sec=12.5 Sec

Description

This module implements a ramp up and ramp down macro. The output variable *Out* follows the desired ramp value *DesiredInput.*

.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: rmp2cntl.h

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

C Interface

Object Definition

The structure of RMP2 object is defined by following structure definition

```
typedef struct { _iq15 DesiredInput;          // Input: Desired ramp input (Q15)
                 _iq15 Ramp2Max;              // Parameter: Maximum limit (Q15)
                 _iq15 Ramp2Min;              // Parameter: Minimum limit (Q15)
                 Uint32  Ramp2DelayCount;     // Variable: Incremental delay (Q0)
                 Uint32 Ramp2Delay;           // Parameter: Delay in # of sampling period (Q0)
                 _iq15 Out;                   // Output: Ramp2 output (Q15)
               } RMP2;
```

Item	Name	Description	Format*	Range(Hex)
Input	DesiredInput	Desired ramp input	Q15	80000000-7FFFFFFF
Output	Out	Ramp 2 output	Q15	80000000-7FFFFFFF
RMP2 parameter	Ramp2Max	Maximum limit	Q15	80000000-7FFFFFFF
	Ramp2Min	Minimum limit	Q15	80000000-7FFFFFFF
	Ramp2Delay	Delay in no. of sampling period	Q0	00000000-7FFFFFFF
Internal	Ramp2DelayCount	Incremental delay	Q0	00000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

RMP2

The module definition is created as a data type. This makes it convenient to instance an interface to the ramp2 control. To create multiple instances of the module simply declare variables of type RMP2.

RMP2_DEFAULTS

Structure symbolic constant to initialize RMP2 module. This provides the initial values to the terminal variables as well as method pointers.

Module Usage

Instantiation

The following example instances two RMP2 objects

```
RMP2  rmp1, rmp2;
```

Initialization
To Instance pre-initialized objects
RMP2 rmp1 = RMP2_DEFAULTS;
RMP2 rmp2 = RMP2_DEFAULTS;

Invoking the computation macro
RC2_MACRO(rmp1);
RC2_MACRO(rmp2);

Example

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        rmp1.DesiredInput = input1;          // Pass inputs to rmp1
        rmp2.DesiredInput = input2;          // Pass inputs to rmp2

        RC2_MACRO (rmp1);                    // Call compute macro for rmp1
        RC2_MACRO (rmp2);                    // Call compute macro for rmp2

        out1 = rmp1.Out;                     // Access the outputs of rmp1
        out2 = rmp2.Out;                     // Access the outputs of rmp2

}
```

Technical Background

Implements the following equations:

Case 1: When *DesiredInput* > *Out*.

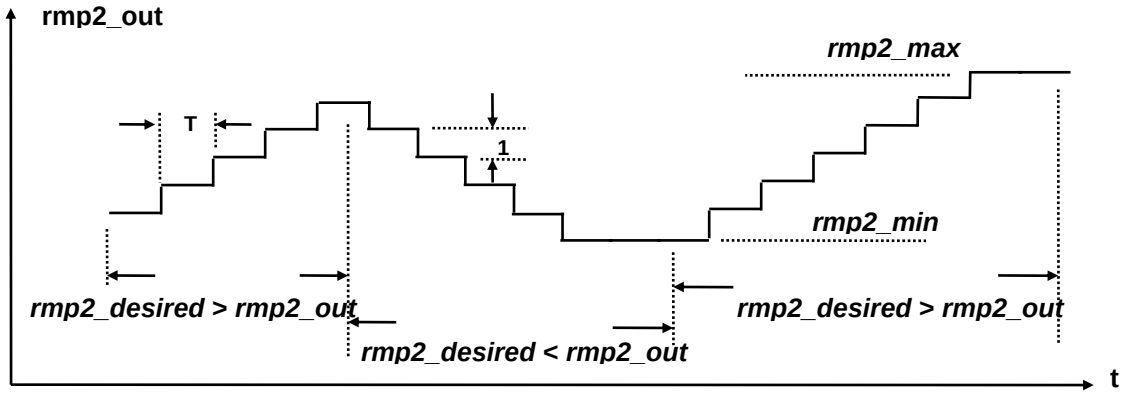
Out= *Out*+ 1, for t = n **.** Td, n = 1, 2, 3, ….. and (*Out*+ 1) < *Ramp2Max*
= *Ramp2Max*, for (*Out*+ 1) > *Ramp2Max*

where, Td = *Ramp2Delay* **.** Ts
Ts = Sampling time period

Case 2: When *DesiredInput* < *Out*.

Out= *Out*- 1, for t = n **.** Td, n = 1, 2, 3, ….. and (*Out*- 1) > *Ramp2Min*
= *Ramp2Min*, for (*Out*- 1) < *Ramp2Min*

where, Td = *Ramp2Delay* **.** Ts
Ts = Sampling time period

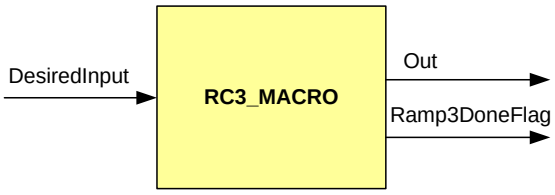


Example:
Out=0(initial value), DesiredInput=1000(user specified),
Ramp2Delay=500(user specified), sampling loop time period Ts=0.000025 Sec.
This means that the time delay for each ramp step is Td=500x0.000025=0.0125 Sec. Therefore, the total ramp time will be Tramp=1000x0.0125 Sec=12.5 Sec

Description

This module implements a ramp down macro. The output flag variable *Ramp3DoneFlag* is set to 0x7FFFFFFF when the output variable *Out* equals the input variable *DesiredInput.*

.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: rmp3cntl.h

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

C Interface

Object Definition

The structure of RMP3 object is defined by following structure definition

```
typedef struct { Uint32 DesiredInput;          // Input: Desired ramp input (Q0)
                 Uint32 Ramp3Delay;            // Parameter: Delay in # of sampling period (Q0)
                 Uint32 Ramp3DelayCount;       // Variable: Counter for rmp3 delay (Q0)
                 int32 Out;                    // Output: Ramp3 output (Q0)
                 int32 Ramp3Min;               // Parameter: Minimum ramp output (Q0)
                 Uint32 Ramp3DoneFlag;         // Output: Flag output (Q0) ction
               } RMP3;
```

Item	Name	Description	Format*	Range(Hex)
Input	DesiredInput	Desired ramp input	Q0	80000000-7FFFFFFF
Outputs	Out	Ramp 3 output	Q0	80000000-7FFFFFFF
	Ramp3DoneFlag	Flag output	Q0	0 or 7FFFFFFF
RMP3 parameter	Ramp3Min	Minimum limit	Q0	80000000-7FFFFFFF
	Ramp3Delay	Delay in no. of sampling period	Q0	00000000-7FFFFFFF
Internal	Ramp3DelayCount	Counter for rmp3 delay	Q0	00000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

RMP3

The module definition is created as a data type. This makes it convenient to instance an interface to the ramp3 control. To create multiple instances of the module simply declare variables of type RMP3.

RMP3_DEFAULTS

Structure symbolic constant to initialize RMP3 module. This provides the initial values to the terminal variables as well as method pointers.

Module Usage

Instantiation

The following example instances two RMP3 objects

```
RMP3  rmp1, rmp2;
```

Initialization
To Instance pre-initialized objects
RMP3 rmp1 = RMP3_DEFAULTS;
RMP3 rmp2 = RMP3_DEFAULTS;

Invoking the computation macro
RC3_MACRO(rmp1);
RC3_MACRO(rmp2);

Example

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        rmp1.DesiredInput = input1;          // Pass inputs to rmp1
        rmp2.DesiredInput = input2;          // Pass inputs to rmp2

        RC3_MACRO (rmp1);                    // Call compute macro for rmp1
        RC3_MACRO (rmp2);                    // Call compute macro for rmp2

        out1 = rmp1.Out;                     // Access the outputs of rmp1
        out2 = rmp2.Out;                     // Access the outputs of rmp2

}
```

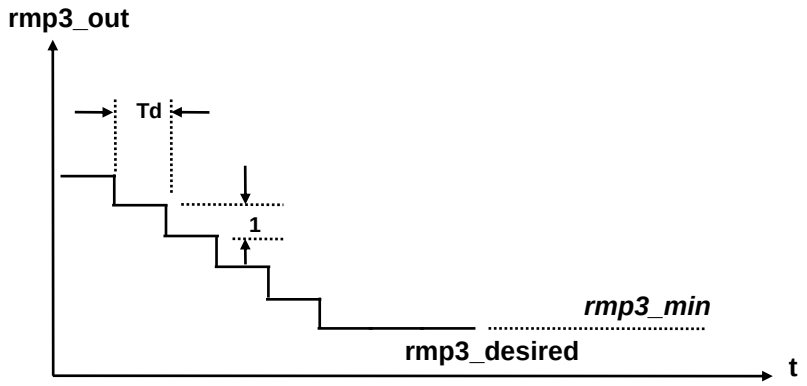
Technical Background

Implements the following equations:

Out = *Out* - 1, for t = n **.** Td, n = 1, 2, 3, ….. and (*Out* - 1) > *Ramp3Min*
= *Ramp3Min*, for (*Out* - 1) < *Ramp3Min*

Ramp3DoneFlag = 7FFFh, when *Out* = *DesiredInput* or *Ramp3Min*

where, Td = *Ramp3Delay* **.** Ts
Ts = Sampling time period



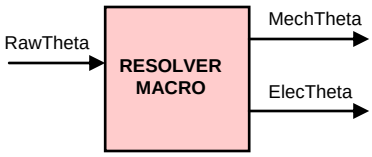
Example:
Out=500(initial value), DesiredInput=20(user specified),
Ramp3Delay=100(user specified), sampling loop time period Ts=0.000025 Sec.
This means that the time delay for each ramp step is Td=100x0.000025=0.0025 Sec. Therefore, the total ramp down time will be Tramp=(500-20)x0.0025 Sec=1.2 Sec

RESOLVER

Mechanical and Electrical Angle Calculation Based on Shaft Angle from Resolver

Description

This module calculates the mechanical and electrical angle of the motor based on raw rotor position measurement from a resolver position sensor.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: resolver.h

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

C Interface

Object Definition

The structure of RESOLVER object is defined by following structure definition

```
typedef struct {
        _iq    ElecTheta;          // Output: Motor Electrical angle (Q24)
        _iq    MechTheta;          // Output: Motor Mechanical Angle (Q24)
        int32 RawTheta;            // Input: Raw position data from resolver (Q0)
        int32 Speed;               // Input: Speed data from resolver (Q4)
        Uint16 StepsPerTurn;   // Parameter: Number of discrete positions (Q0)
        Uint32 MechScaler;     // Parameter: 0.9999/total count (Q30)
        Uint16 PolePairs;      // Parameter: Number of pole pairs (Q0)
        int32 InitTheta;       // Parameter: Raw angular offset between resolver and phase A (Q0)
  }  RESOLVER;                 // Data type created
```

Item	Name	Description	Format*	Range(Hex)
Inputs	RawTheta	Raw resolver angle	Q0	00000000-0000FFFF (0 – 360 degree)
	Speed	Resolver Speed	Q4	80000000-7FFFFFFF
Outputs	MechTheta	Mechanical angle in per-unit	GLOBAL_Q	80000000-7FFFFFFF
	ElecTheta	Speed in rpm	GLOBAL_Q	80000000-7FFFFFFF
RESOLVER parameter	StepsPerTurn	Number of resolver steps per turn	Q0	80000000-7FFFFFFF
	MechScaler	Mech angle per resolver step	Q30	80000000-7FFFFFFF
	PolePairs	Number of pole pairs of motor	Q0	80000000-7FFFFFFF
	InitTheta	Resolver output angle at end of initial alignment	Q0	00000000-7FFFFFFF (0 – 360 degree)

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

RESOLVER

The module definition is created as a data type. This makes it convenient to instance an interface to resolver for angle estimation. To create multiple instances of the module simply declare variables of type RESOLVER.

RESOLVER_DEFAULTS

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

Module Usage

Instantiation
The following example instances two RESOLVER objects
RESOLVER resolver1, resolver2;

Initialization
To Instance pre-initialized objects
RESOLVER resolver1 = RESOLVER _DEFAULTS;
RESOLVER resolver2 = RESOLVER _DEFAULTS;

Invoking the computation macro
RESOLVER_MACRO (resolver1);
RESOLVER_MACRO (resolver2);

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

```
main()
{


}


void interrupt periodic_interrupt_isr()
{
        resolver1.RawTheta = theta1;        // Pass inputs to resolver1
        resolver2.RawTheta = theta2;        // Pass inputs to resolver2

        RESOLVER_MACRO (resolver1);         // Call compute macro for resolver1
        RESOLVER_MACRO (resolver2);         // Call compute macro for resolver2

        mechPos1 = resolver1.MechTheta;     // Access the outputs of resolver1
        elecPos2   = resolver2.ElecTheta;   // Access the outputs of resolver2
}
```

Technical Background

Typical waveforms of the resolver position angle, θ_e, and stator phase A angle are given in Figure 1 for a single pole pair motor. Before starting the motor, it is important to know the offset between these two angles. So a rotor alignment is performed such that phase A angle is zero. The resolver output at this position is the mechanical (electrical) angular offset between these two angles. This offset is termed 'InitTheta' which must have already been figured out before calling this routine. Stator A position can then be derived by subtracting the offset from the resolver output. In multi pole pair motors, electrical angle can be computed by multiplying the mechanical angle with pole pairs. {In multi pole pair motors, alignment with stator phase A may position the rotor at different mechanical positions based on the nearest pair of poles}

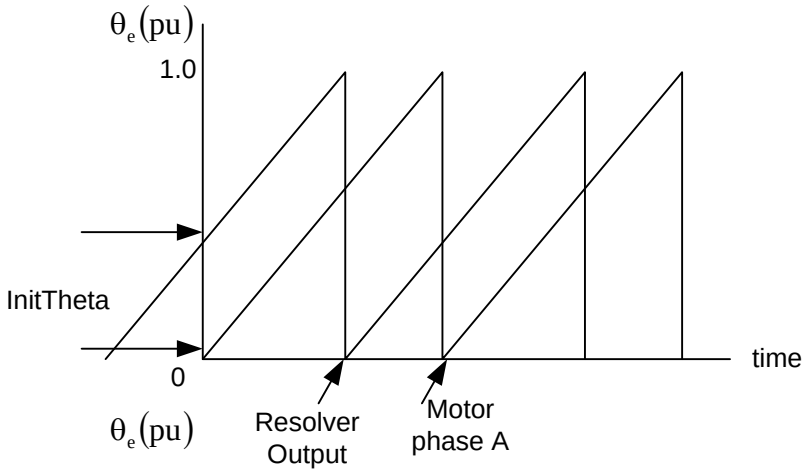
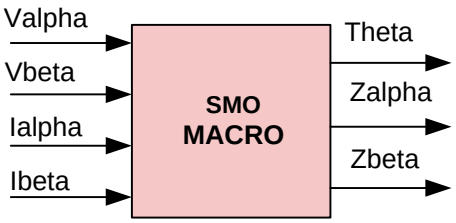


Figure 1: The waveforms of resolver position and phase A position

Description

This software module implements a rotor position estimation algorithm for Permanent-Magnet Synchronous Motor (PMSM) based on Sliding-Mode Observer (SMO).



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: smopos.h

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

C Interface

Object Definition

The structure of SMOPOS object is defined by following structure definition

```
typedef struct {  _iq  Valpha;       // Input: Stationary alpha-axis stator voltage
                  _iq  Ealpha;       // Variable: Stationary alpha-axis back EMF
                  _iq  Zalpha;       // Output: Stationary alpha-axis sliding control
                  _iq  Gsmopos;   // Parameter: Motor dependent control gain
                  _iq  EstIalpha;    // Variable: Estimated stationary alpha-axis stator current
                  _iq  Fsmopos;    // Parameter: Motor dependent plant matrix
                  _iq  Vbeta;        // Input: Stationary beta-axis stator voltage
                  _iq  Ebeta;        // Variable: Stationary beta-axis back EMF
                  _iq  Zbeta;        // Output: Stationary beta-axis sliding control
                  _iq  EstIbeta;     // Variable: Estimated stationary beta-axis stator current
                  _iq  Ialpha;       // Input: Stationary alpha-axis stator current
                  _iq  IalphaError;  // Variable: Stationary alpha-axis current error
                  _iq  Kslide;       // Parameter: Sliding control gain
                  _iq  Ibeta;        // Input: Stationary beta-axis stator current
                  _iq  IbetaError;   // Variable: Stationary beta-axis current error
                  _iq  Kslf;         // Parameter: Sliding control filter gain
                  _iq  Theta;        // Output: Compensated rotor angle
                } SMOPOS;
```

Module Terminal Variables

Item	Name	Description	Format*	Range(Hex)
Inputs	Valpha	stationary d-axis stator voltage	GLOBAL_Q	80000000-7FFFFFFF
	Vbeta	stationary q-axis stator voltage	GLOBAL_Q	80000000-7FFFFFFF
	Ialpha	stationary d-axis stator current	GLOBAL_Q	80000000-7FFFFFFF
	Ibeta	stationary q-axis stator current	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Theta	rotor position angle	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)
	Zalfa	stationary d-axis sliding control	GLOBAL_Q	80000000-7FFFFFFF
	Zbeta	stationary q-axis sliding control	GLOBAL_Q	80000000-7FFFFFFF
SMOPOS parameter	Fsmopos	Fsmopos = exp(-Rs*T/Ls)	GLOBAL_Q	80000000-7FFFFFFF
	Gsmopos	Gsmopos = (Vb/Ib)*(1- exp(-Rs*T/Ls))/Rs	GLOBAL_Q	80000000-7FFFFFFF
	Kslide	sliding mode control gain	GLOBAL_Q	80000000-7FFFFFFF
	Kslf	sliding control filter gain	GLOBAL_Q	80000000-7FFFFFFF
Internal	Ealpha	stationary d-axis back EMF	GLOBAL_Q	80000000-7FFFFFFF
	Ebeta	stationary q-axis back EMF	GLOBAL_Q	80000000-7FFFFFFF
	EstIalpha	stationary d-axis estimated current	GLOBAL_Q	80000000-7FFFFFFF
	EstIbeta	stationary q-axis estimated current	GLOBAL_Q	80000000-7FFFFFFF
	IalphaError	stationary d-axis current error	GLOBAL_Q	80000000-7FFFFFFF
	IbetaError	stationary q-axis current error	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

SMOPOS
The module definition is created as a data type. This makes it convenient to instance an interface to the sliding-mode rotor position observer of Permanent-Magnet Synchronous Motor module. To create multiple instances of the module simply declare variables of type SMOPOS.

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

Module Usage

Instantiation
The following example instances two SMOPOS objects
SMOPOS smo1, smo2;

Initialization
To Instance pre-initialized objects
SMOPOS fe1 = SMOPOS_DEFAULTS;
SMOPOS fe2 = SMOPOS_DEFAULTS;

Invoking the computation macro
SMO_MACRO (smo1);
SMO_MACRO (smo2);

Example

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

```
main()
{
        smo1.Fsmopos = parem1_1;        // Pass parameters to smo1
        smo1.Gsmopos = parem1_2;        // Pass parameters to smo1
        smo1.Kslide = parem1_3;         // Pass parameters to smo1
        smo1.Kslf = parem1_4;           // Pass parameters to smo1
        smo2.Fsmopos = parem2_1;        // Pass parameters to smo2
        smo2.Gsmopos = parem2_2;        // Pass parameters to smo2
        smo2.Kslide = parem2_3;         // Pass parameters to smo2
        smo2.Kslf = parem2_4;           // Pass parameters to smo2

}
```

```
void interrupt periodic_interrupt_isr()
{
        smo1.Valpha = voltage_dq1.d;        // Pass inputs to smo1
        smo1.Vbeta = voltage_dq1.q;         // Pass inputs to smo1
        smo1.Ialpha =current_dq1.d;         // Pass inputs to smo1
        smo1.Ibeta =current_dq1.q;          // Pass inputs to smo1

        smo2.Valpha = voltage_dq2.d;        // Pass inputs to smo2
        smo2.Vbeta = voltage_dq2.q;         // Pass inputs to smo2
        smo2.Ialpha =current_dq2.d;         // Pass inputs to smo2
        smo2.Ibeta =current_dq2.q;          // Pass inputs to smo2

        SMO_MACRO(smopos1)                  // Call compute macro for smopos1
        SMO_MACRO(smopos2);                 // Call compute macro for smopos2

        angle1 = smopos1.Theta;             // Access the outputs of smopos1
        angle2 = smopos2.Theta;             // Access the outputs of smopos2

}
```

Constant Computation Macro

Since the sliding-mode rotor position observer of Permanent-Magnet Synchronous Motor module requires two constants (Fsmopos and Gsmopos) to be input basing on the machine parameters, base quantities, mechanical parameters, and sampling period. These two constants can be internally computed by the macro (smopos_const.h). The followings show how to use the C constant computation macro.

Object Definition

The structure of SMOPOS_CONST object is defined by following structure definition

```
typedef struct  { float32  Rs;        // Input: Stator resistance (ohm)
                  float32  Ls;        // Input: Stator inductance (H)
                  float32  Ib;        // Input: Base phase current (amp)
                  float32  Vb;        // Input: Base phase voltage (volt)
                  float32  Ts;        // Input: Sampling period in sec
                  float32  Fsmopos;   // Output: constant using in observed current cal.
                  float32  Gsmopos;   // Output: constant using in observed current cal.
                } SMOPOS_CONST;
```

Module Terminal Variables

Item	Name	Description	Format	Range(Hex)
Inputs	Rs	Stator resistance (ohm)	Floating	N/A
	Ls	Stator inductance (H)	Floating	N/A
	Ib	Base phase current (amp)	Floating	N/A
	Vb	Base phase voltage (volt)	Floating	N/A
	Ts	Sampling period (sec)	Floating	N/A
Outputs	Fsmopos	constant using in observed current calculation	Floating	N/A
	Gsmopos	constant using in observed current calculation	Floating	N/A

Special Constants and Data types

SMOPOS_CONST

The module definition is created as a data type. This makes it convenient to instance an interface to the sliding-mode rotor position observer of Permanent-Magnet Synchronous Motor constant computation module. To create multiple instances of the module simply declare variables of type SMOPOS_CONST.

SMOPOS_CONST_DEFAULTS

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

Module Usage

Instantiation
The following example instances two SMOPOS_CONST objects
SMOPOS_CONST smopos1_const, smopos2_const;

Initialization
To Instance pre-initialized objects
SMOPOS_CONST smopos1_const = SMOPOS_CONST_DEFAULTS;
SMOPOS_CONST smopos2_const = SMOPOS_CONST_DEFAULTS;

Invoking the computation macro
SMO_CONST_MACRO (smopos1_const);
SMO_CONST_MACRO (smopos2_const);

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

```
main()
{
        smopos1_const.Rs = Rs1;      // Pass floating-point inputs to smopos1_const
        smopos1_const.Ls = Ls1;      // Pass floating-point inputs to smopos1_const
        smopos1_const.Ib = Ib1;      // Pass floating-point inputs to smopos1_const
        smopos1_const.Vb = Vb1;      // Pass floating-point inputs to smopos1_const
        smopos1_const.Ts = Ts1;      // Pass floating-point inputs to smopos1_const

        smopos2_const.Rs = Rs2;      // Pass floating-point inputs to smopos2_const
        smopos2_const.Ls = Ls2;      // Pass floating-point inputs to smopos2_const
        smopos2_const.Ib = Ib2;      // Pass floating-point inputs to smopos2_const
        smopos2_const.Vb = Vb2;      // Pass floating-point inputs to smopos2_const
        smopos2_const.Ts = Ts2;      // Pass floating-point inputs to smopos2_const

        SMO_CONST_MACRO (smopos1_const); // Call compute macro for smopos1_const
        SMO_CONST_MACRO (smopos2_const); // Call compute macro for smopos2_const

        // Access the outputs of smopos1_const
        smopos1.Fsmopos = _IQ(smopos1_const.Fsmopos);
        smopos1.Gsmopos = _IQ(smopos1_const.Gsmopos);
        // Access the outputs of smopos2_const
        smopos2.Fsmopos = _IQ(smopos2_const.Fsmopos);
        smopos2.Gsmopos = _IQ(smopos2_const.Gsmopos);
}
```

Technical Background

Figure 1 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 1 derives the position of rotor flux from encoder feedback. However, the encoder increases system cost and complexity.

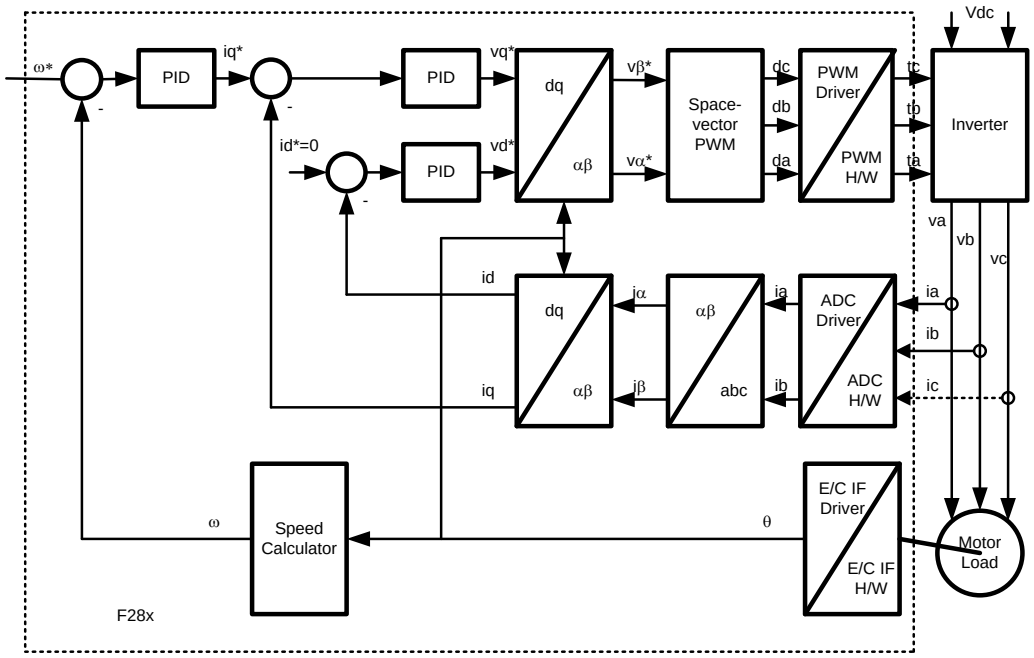


Figure 1 Field Oriented Control of PMSM

Therefore for cost sensitive applications, it is ideal if the rotor flux position information can be derived from measurement of voltages and currents. Figure 2 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.

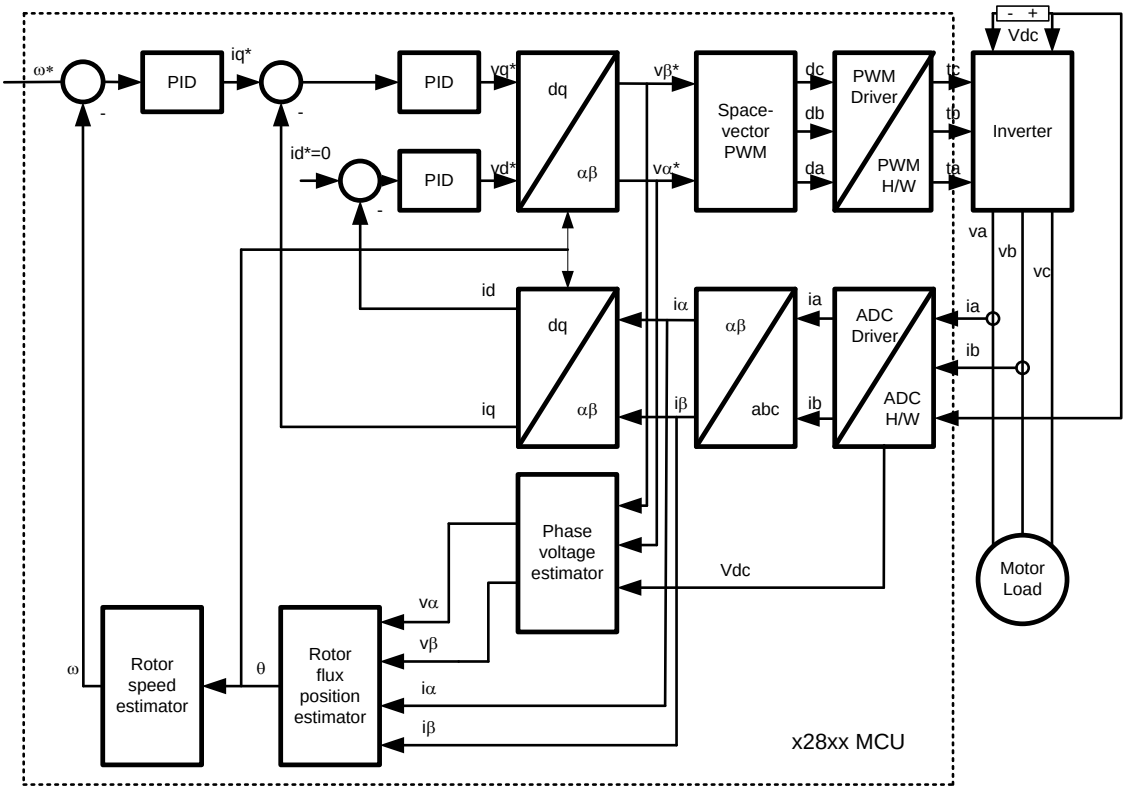


Figure 2 Sensorless Field Oriented Control of PMSM

This software module implements a rotor flux position estimator based on a sliding mode current observer. As shown in Figure 3, the inputs to the estimator are motor phase currents and voltages expressed in α-β coordinate frame.

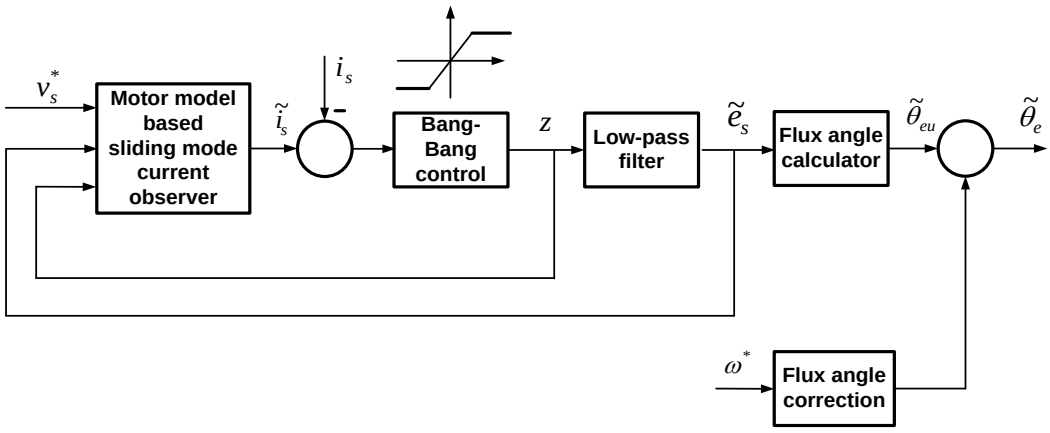


Figure 3 Sliding Mode Observer Based Rotor Flux Position Estimator

Figure 4 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.

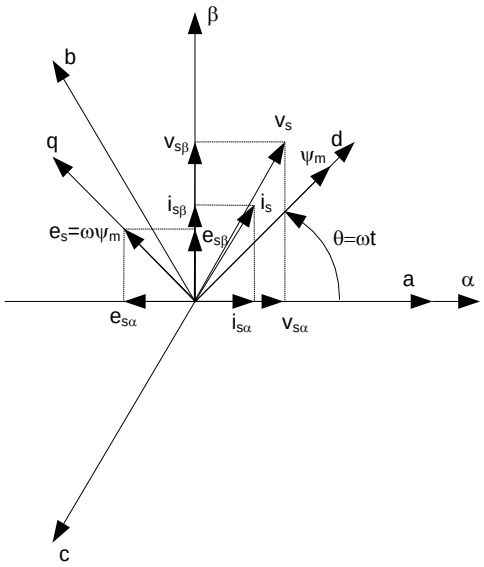


Figure 4 PMSM Coordinate Frames and Vectors

Equation 1 is the mathematical model of PMSM in α-β coordinate frame.

$$\frac{d}{dt} i_s = A i_s + B(v_s - e_s)$$

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. Next the mathematical equations for the blocks in Figure 3 are discussed.

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.

$$\frac{d}{dt} \tilde{i}_s = A \tilde{i}_s + B(v_s^* - \tilde{e}_s - z)$$

$$z = k\, sign(\tilde{i}_s - i_s)$$

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, 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.

$$\tilde{i}_s(n+1) = F\,\tilde{i}_s(n) + G(v_s^*(n) - \tilde{e}_s(n) - z(n))$$

$$z(n) = k\,sign(\tilde{i}_s(n) - i_s(n))$$

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.

$$\frac{d}{dt}\tilde{e}_s = -\omega_0 \tilde{e}_s + \omega_0 z$$

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.

$$\tilde{e}_s(n+1) = \tilde{e}_s(n) + 2\pi f_0 (z(n) - \tilde{e}_s(n))$$

3. Rotor Flux Position Calculation

Estimated rotor flux angle is obtained based on Equation 8 for back emf.

$$e_s = \frac{3}{2} k_e \omega \begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix}$$

Therefore given the estimated back emf, estimated rotor position can be calculated based on Equation 9.

$$\tilde{\theta}_{eu} = \arctan(-\tilde{e}_{s\alpha}, \tilde{e}_{s\beta})$$

Next, Table 1 shows the correspondence of notations between variables used here and variables used in the program (i.e. smopos.h). The software module requires that both input and output variables are in per unit values.

	Equation Variables	Program Variables
Inputs	$v_{s\alpha}$*	Valpha
	$v_{s\beta}$*	Vbeta
	$i_{s\alpha}$	Ialpha
	$i_{s\beta}$	Ibeta
Outputs	$\tilde{\theta}_e$	Theta
	z_α	Zalpha
	z_β	Zbeta
Others	$\tilde{i}_{s\alpha}$	EstIalpha
	$\tilde{i}_{s\beta}$	EstIbeta
	$\tilde{e}_{s\alpha}$	Ealpha
	$\tilde{e}_{s\beta}$	Ebeta
	$e^{-\frac{R}{L}T_s}$	Fsmopos
	$\frac{1}{R}(1-e^{-\frac{R}{L}T_s})$	Gsmopos
	k	Kslide
	$2\pi f_0$	Kslf

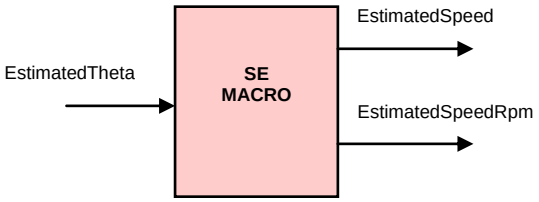
Table 1: Correspondence of notations

SPEED_EST

Speed Calculator Based on Rotor Angle Without Direction Information

Description

This module calculates the motor speed based on the estimated rotor position when the rotational direction information is not available.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: speed_est.h

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

C Interface

Object Definition

The structure of SPEED_ESTIMATION object is defined by following structure definition

```
typedef struct {_iq EstimatedTheta;          // Input: Electrical angle
                _iq OldEstimatedTheta;       // History: Electrical angle at previous step
                _iq EstimatedSpeed;          // Output: Estimated speed in per-unit
                Uint32 BaseRpm;              // Parameter: Base speed in rpm (Q0)
                _iq21 K1;                    // Parameter: Constant for differentiator (Q21)
                _iq K2;                      // Parameter: Constant for low-pass filter
                _iq K3;                      // Parameter: Constant for low-pass filter
                int32 EstimatedSpeedRpm;     // Output : Estimated speed in rpm  (Q0)
               } SPEED_ESTIMATION;           // Data type created
;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	EstimatedTheta	Electrical angle	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)
Outputs	EstimatedSpeed	Computed speed in per-unit	GLOBAL_Q	80000000-7FFFFFFF
	EstimatedSpeedRpm	Speed in rpm	Q0	80000000-7FFFFFFF
ESTIMATED SPEED parameter	K1	K1 = 1/(fb*T)	Q21	80000000-7FFFFFFF
	K2	K2 = 1/(1+T*2*pi*fc)	GLOBAL_Q	80000000-7FFFFFFF
	K3	K3 = T*2*pi*fc/(1+T*2*pi*fc)	GLOBAL_Q	80000000-7FFFFFFF
	BaseRpm	BaseRpm = 120fb/p	Q0	80000000-7FFFFFFF
Internal	OldEstimatedTheta	Electrical angle in previous step	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

SPEED_ESTIMATION

The module definition is created as a data type. This makes it convenient to instance an interface to speed calculation based on measured rotor angle. To create multiple instances of the module simply declare variables of type SPEED_ESTIMATION.

SPEED_ESTIMATION_DEFAULTS

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

Module Usage

Instantiation

The following example instances two SPEED_ESTIMATION objects
SPEED_ESTIMATION speed1, speed2;

Initialization

To Instance pre-initialized objects
SPEED_ESTIMATION speed1 = SPEED_ESTIMATION_DEFAULTS;
SPEED_ESTIMATION speed2 = SPEED_ESTIMATION_DEFAULTS;

Invoking the computation macro

SE_MACRO(speed1);
SE_MACRO(speed2);

Example

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

```
main()
{

}


void interrupt periodic_interrupt_isr()
{
        speed1.EstimatedTheta = theta1;          // Pass inputs to speed1
        speed2.EstimatedTheta = theta2;          // Pass inputs to speed2

        SE_MACRO (speed1);                       // Call compute macro for speed1
        SE_MACRO (speed2);                       // Call compute macro for speed2

        measured_spd1 = speed1.EstimatedSpeed;   // Access the outputs of speed1
        measured_spd2 = speed2.EstimatedSpeed;   // Access the outputs of speed2
}
```

Technical Background

The typical waveforms of the electrical rotor position angle, θ_e, in both directions can be seen in Figure 1. Assuming the direction of rotation is not available. To take care the discontinuity of angle from 360^o to 0^o (CCW) or from 0^o to 360^o (CW), the differentiator is simply operated only within the differentiable range as seen in this Figure. This differentiable range does not significantly lose the information to compute the estimated speed.

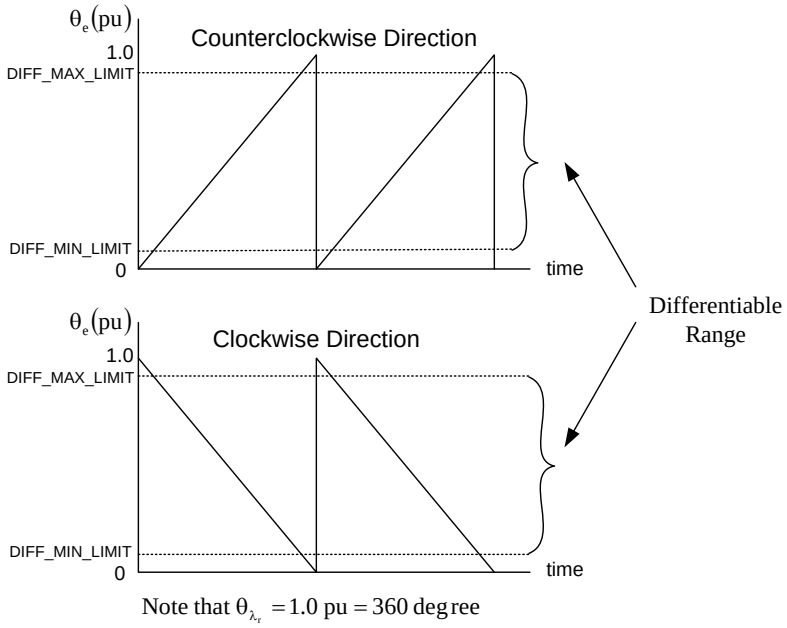


Figure 1: The waveforms of rotor position in both directions

The differentiator equation of rotor position can be expressed as follows.

$$\omega_e(k) = K_1\left(\theta_e(k) - \theta_e(k-1)\right) \quad (1)$$

where $K_1 = \dfrac{1}{f_b T}$, f_b is base frequency (Hz) and T is sampling period (sec).

In addition, the rotor speed is necessary to be filtered out by the low-pass filter in order to reduce the amplifying noise generated by the pure differentiator. The simple 1st-order low-pass filter is used, then the actual rotor speed to be used is the output of the low-pass filter, $\hat{\omega}_e$, seen in following equation. The continuous-time equation of 1st-order low-pass filter is as

$$\frac{d\hat{\omega}_e}{dt} = \frac{1}{\tau_c}\left(\omega_e - \hat{\omega}_e\right) \quad (2)$$

where $\tau_c = \frac{1}{2\pi f_c}$ is the low-pass filter time constant (sec), and f_c is the cut-off frequency (Hz). Using backward approximation, then (2) finally becomes

$$\hat{\omega}_e(k) = K_2\hat{\omega}_e(k-1) + K_3\omega_e(k) \quad (3)$$

where $K_2 = \frac{\tau_c}{\tau_c + T}$, and $K_3 = \frac{T}{\tau_c + T}$.

Next, Table 1 shows the correspondence of notations between variables used here and variables used in the program (i.e. speed_est.h). The software module requires that both input and output variables are in per unit values.

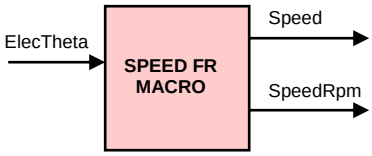
	Equation Variables	Program Variables
Input	θ_e	EstimatedTheta
Output	$\hat{\omega}_e$	EstimatedSpeed
Others	K1	K1
	K2	K2
	K3	K3

Table 1: Correspondence of notations

SPEED_FR *Speed Calculator Based on Rotor Angle from QEP sensor*

Description

This module calculates the motor speed based on a rotor position measurement from QEP sensor.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: speed_fr.h

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

C Interface

Object Definition

The structure of SPEED_MEAS_QEP object is defined by following structure definition

```
typedef struct {  _iq ElecTheta;          // Input: Electrical angle
                  Uint32 DirectionQep;    // Variable: Direction of rotation (Q0)
                  _iq OldElecTheta;       // History: Electrical angle at previous step
                  _iq Speed;              // Output: Speed in per-unit
                  Uint32 BaseRpm;         // Parameter: Base speed in rpm (Q0)
                  _iq21 K1;               // Parameter: Constant for differentiator (Q21)
                  _iq K2;                 // Parameter: Constant for low-pass filter
                  _iq K3;                 // Parameter: Constant for low-pass filter
                  int32 SpeedRpm;         // Output : Speed in rpm  (Q0)
               } SPEED_MEAS_QEP;          // Data type created
```

Item	Name	Description	Format*	Range(Hex)
Inputs	ElecTheta	Electrical angle	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)
Outputs	Speed	Computed speed in per-unit	GLOBAL_Q	80000000-7FFFFFFF
	SpeedRpm	Speed in rpm	Q0	80000000-7FFFFFFF
SPEED_QEP parameter	K1	K1 = 1/(fb*T)	Q21	80000000-7FFFFFFF
	K2	K2 = 1/(1+T*2*pi*fc)	GLOBAL_Q	80000000-7FFFFFFF
	K3	K3 = T*2*pi*fc/(1+T*2*pi*fc)	GLOBAL_Q	80000000-7FFFFFFF
	BaseRpm	BaseRpm = 120fb/p	Q0	80000000-7FFFFFFF
Internal	OldElecTheta	Electrical angle in previous step	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

SPEED_MEAS_QEP

The module definition is created as a data type. This makes it convenient to instance an interface to speed calculation based on measured rotor angle. To create multiple instances of the module simply declare variables of type SPEED_MEAS_QEP.

SPEED_MEAS_QEP_DEFAULTS

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

Module Usage

Instantiation
The following example instances two SPEED_MEAS_QEP objects
SPEED_MEAS_QEP speed1, speed2;

Initialization
To Instance pre-initialized objects
SPEED_MEAS_QEP speed1 = SPEED_MEAS_QEP_DEFAULTS;
SPEED_MEAS_QEP speed2 = SPEED_MEAS_QEP_DEFAULTS;

Invoking the computation macro
SPEED_FR_MACRO (speed1);
SPEED_FR_MACRO (speed2);

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

```
main()
{


}


void interrupt periodic_interrupt_isr()
{
        speed1.ElecTheta = theta1;          // Pass inputs to speed1
        speed2.ElecTheta = theta2;          // Pass inputs to speed2

        SPEED_FR_MACRO (speed1);            // Call compute macro for speed1
        SPEED_FR_MACRO (speed2);            // Call compute macro for speed2

        measured_spd1 = speed1.Speed;       // Access the outputs of speed1
        measured_spd2 = speed2.Speed;       // Access the outputs of speed2
}
```

Technical Background

The typical waveforms of the electrical rotor position angle, θ_e, in both directions can be seen in Figure 1. Assuming the direction of rotation is not available. Speed is estimated based on differentiation of angular values between successive iterations. To take care of the discontinuity of angle from 360° to 0° (CCW) or from 0° to 360° (CW), an error roll over to fit the difference numerically within -180° and +180° is performed.

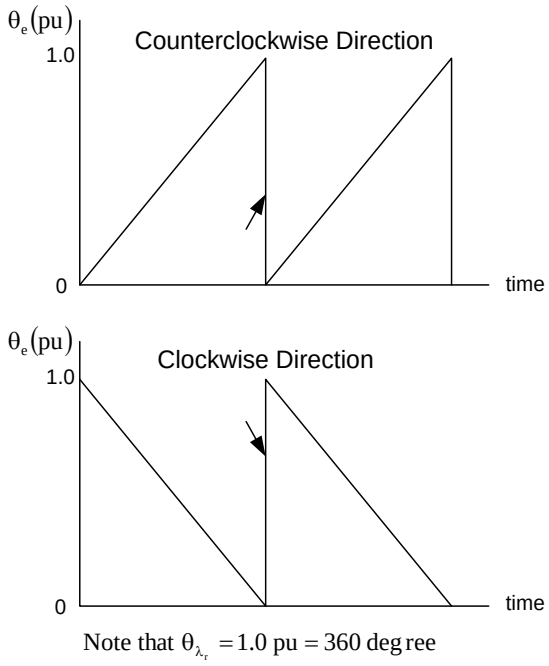


Figure 1: The waveforms of rotor position in both directions

The differentiator equation of rotor position can be expressed as follows.

$$\omega_e(k) = K_1\left(\theta_e(k) - \theta_e(k-1)\right) \quad (1)$$

where $K_1 = \frac{1}{f_b T}$, f_b is base frequency (Hz) and T is sampling period (sec).

In addition, the rotor speed is necessary to be filtered out by the low-pass filter in order to reduce the amplifying noise generated by the pure differentiator. The simple 1st-order low-pass filter is used, then the actual rotor speed to be used is the output of the low-pass filter, $\hat{\omega}_e$, seen in following equation. The continuous-time equation of 1st-order low-pass filter is as

$$\frac{d\hat{\omega}_e}{dt} = \frac{1}{\tau_c}\left(\omega_e - \hat{\omega}_e\right) \quad (2)$$

where $\tau_c = \frac{1}{2\pi f_c}$ is the low-pass filter time constant (sec), and f_c is the cut-off frequency (Hz). Using backward approximation, then (2) finally becomes

$$\hat{\omega}_e(k) = K_2\hat{\omega}_e(k-1) + K_3\omega_e(k) \quad (3)$$

where $K_2 = \frac{\tau_c}{\tau_c + T}$, and $K_3 = \frac{T}{\tau_c + T}$.

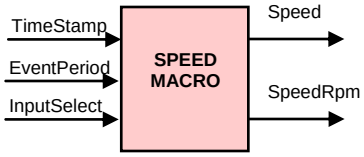
Next, Table 1 shows the correspondence of notations between variables used here and variables used in the program (i.e. speed_fr.h). The software module requires that both input and output variables are in per unit values.

	Equation Variables	Program Variables
Input	θ_e	ElecTheta
Output	$\hat{\omega}_e$	Speed
Others	K1	K1
	K2	K2
	K3	K3

Table 1: Correspondence of notations

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

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: speed_pr.h

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

C Interface

Object Definition

The structure of SPEED_MEAS_CAP object is defined by following structure definition

```
typedef struct {  Uint32 NewTimeStamp; // Variable: New 'Timestamp' for a capture event (Q0)
                  Uint32 OldTimeStamp;  // Variable : Old 'Timestamp' for a capture event (Q0)
                  Uint32 TimeStamp;     // Input: Current 'Timestamp' for a capture event (Q0)
                  Uint32 SpeedScaler;   // Parameter:  Scaler converting 1/N cycles (Q0)
                   int32 EventPeriod;   // Input/Variable :  Event Period (Q0)
                   int16 InputSelect;   // Input : Input selection (1 or 0)
                  _iq Speed;            // Output:  speed in per-unit
                  Uint32 BaseRpm;       // Parameter : Scaler converting to rpm (Q0)
                  int32 SpeedRpm;       // Output : speed in r.p.m. (Q0)
                } SPEED_MEAS_CAP;    // Data type created
;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	TimeStamp	Current 'Timestamp' for a capture event	Q0	80000000-7FFFFFFF
	InputSelect	TimeStamp (InputSelect=0) and EventPeriod (InputSelect=1)	Q0	0 or 1
	EventPeriod	Event period between time stamp	Q0	80000000-7FFFFFFF
Outputs	Speed	Computed speed in per-unit	GLOBAL_Q	80000000-7FFFFFFF
	SpeedRpm	Speed in rpm	Q0	80000000-7FFFFFFF
SPEED_CAP parameter	SpeedScaler	SpeedScaler = 60*CLK_freq/(128*N*rmp_max), N = number of sprocket teeth	Q0	80000000-7FFFFFFF
	BaseRpm	rmp_max = 120fb/p	Q0	80000000-7FFFFFFF
Internal	NewTimeStamp	New 'Timestamp' for a capture event	Q0	80000000-7FFFFFFF
	OldTimeStamp	Old 'Timestamp' for a capture event	Q0	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

SPEED_MEAS_CAP

The module definition is created as a data type. This makes it convenient to instance an interface to speed calculation based on period. To create multiple instances of the module simply declare variables of type SPEED_MEAS_CAP.

SPEED_MEAS_CAP_DEFAULTS

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

Module Usage

Instantiation

The following example instances two SPEED_MEAS_CAP objects

```
SPEED_MEAS_CAP speed1, speed2;
```

Initialization

To Instance pre-initialized objects

```
SPEED_MEAS_CAP speed1 = SPEED_MEAS_CAP_DEFAULTS;
SPEED_MEAS_CAP speed2 = SPEED_MEAS_CAP_DEFAULTS;
```

Invoking the computation macro

```
SPEED_PR_MACRO(speed1);
SPEED_PR_MACRO(speed2);
```

Example

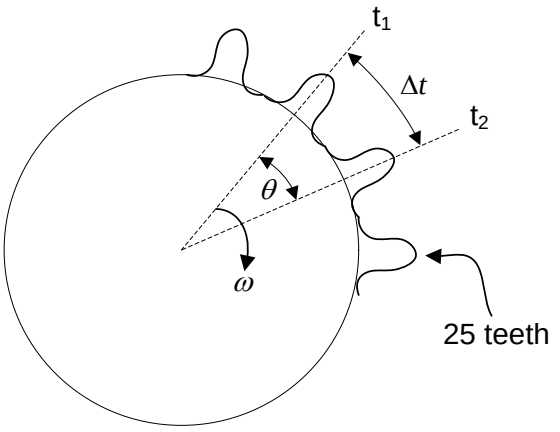
The following pseudo code provides the information about the module usage.

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        speed1.TimeStamp = TimeStamp1;      // Pass inputs to speed1
        speed2.TimeStamp = TimeStamp2;      // Pass inputs to speed2

        SPEED_PR_MACRO(speed2);             // Call compute macro for speed1
        SPEED_PR_MACRO(speed2);             // Call compute macro for speed2

        measured_spd1 = speed1.Speed;       // Access the outputs of speed1
        measured_spd2 = speed2.Speed;       // Access the outputs of speed2
}
```

Technical Background

A low cost shaft sprocket with n teeth and a Hall effect gear tooth sensor is used to measure the motor speed. Fig. 1 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 281x 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 *TimeStamp*.

In this module, every time a new input *TimeStamp* becomes available it is compared with the previous *TimeStamp*. 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.



$$\Delta t = t_2 - t_1 \quad \text{sec}$$

$$\theta = \frac{360}{25} = 14.4^0$$

$$\omega = \frac{2\pi}{\sum_{n=1}^{25} \Delta t_n} \quad \text{rad/sec}$$

Fig. 1 Speed measurement with a sprocket

From the two consecutive *TimeStamp* values the difference between the captured values are calculated as,

Δ = TimeStamp(new) – TimeStamp(old)

Then the time period in sec is given by,

$$\Delta t = t_2 - t_1 = \mathrm{K_P} \times \mathrm{T}_{CLK} \times \Delta$$

where,

K_P = Prescaler value for the Capture unit time base

T_{CLK} = CPU clock period in sec

From Fig. 1, 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,

$$\omega = \frac{\theta}{\Delta t} = \frac{2\pi}{n\Delta t} = \frac{2\pi}{n \times \mathrm{K}_P \times \mathrm{T}_{CLK} \times \Delta}$$

$$\Rightarrow \omega_N = \frac{\omega}{\omega_{\max}} = \frac{\omega}{2\pi\left(\frac{1}{n \times \mathrm{K}_P \times \mathrm{T}_{CLK}}\right)} = \frac{1}{\Delta}$$

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,

$$\omega_{\max 1} = \frac{23438 \times 2\pi}{60} \approx 2\pi(390)$$

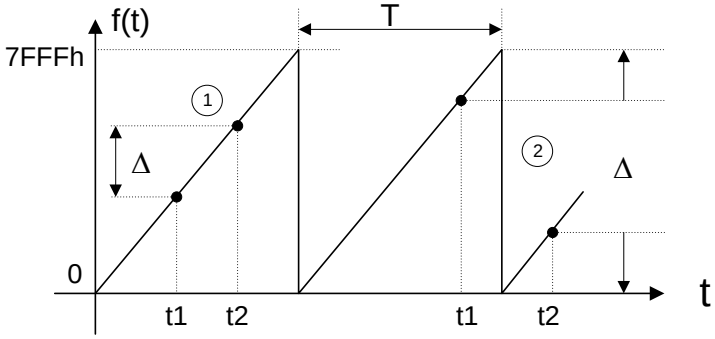
Therefore, the scaled normalized speed in per-unit is calculated as,

$$\omega_{N1} = \frac{\omega}{2\pi(390)} \approx \frac{64}{\Delta} = 64 \times \omega_N = SpeedScaler \times \omega_N$$

This shows that in this case the scaling factor is 64. The speed, in rpm, is calculated as,

$$N_1 = 23438 \times \omega_{N1} = 23438 \times \frac{64}{\Delta} = BaseRpm \times \omega_{N1}$$

The capture unit 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 Fig. 2.



Case ① $f(t_2) \geq f(t_1)$

$\Delta = f(t_2) - f(t_1)$

Case ② $f(t_2) \leq f(t_1)$

$\Delta = 1 + f(t_2) - f(t_1)$

Note: only true if $t_2 - t_1 < T$

Fig. 2 Calculation of speed

In Fig. 2, 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 = 7FFFh. 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.

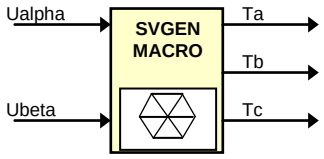
Once a "robust" period measurement is extracted from the averaging algorithm, the speed is calculated using the appropriate equations explained before

Finally, for a base speed in rpm (BaseRpm), the *SpeedScaler* is basically $1/\omega_N$ (with $\omega = \omega_{max1}$) for the number of teeth in the sprocket, prescaler value for the capture unit time base, and CPU clock period (sec) is therefore computed as follows:

$$\text{SpeedScaler} = \frac{60}{T_{CLK} \times K_p \times n \times \text{BaseRpm}}$$

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, Ualpha and Ubeta.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: aci_se.h

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

C Interface

Object Definition

The structure of SVGENDQ object is defined by following structure definition

typedef struct

```
{ _iq  Ualpha;          // Input: reference alpha-axis phase voltage
  _iq  Ubeta;           // Input: reference beta-axis phase voltage
  _iq  Ta;              // Output: reference phase-a switching function
  _iq  Tb;              // Output: reference phase-b switching function
  _iq  Tc;              // Output: reference phase-c switching function
} SVGENDQ;
```

Module Terminal Variables/Macros

Item	Name	Description	Format*	Range(Hex)
Inputs	Ualpha	Component of reference stator voltage vector on direct axis stationary reference frame.	GLOBAL_Q	80000000-7FFFFFFF
	Ubeta	Component of reference stator voltage vector on quadrature axis stationary reference frame.	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Ta	Duty ratio of PWM1 (CMPR1 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
	Tb	Duty ratio of PWM3 (CMPR2 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
	Tc	Duty ratio of PWM5 (CMPR3 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
Variable	tmp1,2,3	Internal variable	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

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

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

Module Usage

Instantiation
The following example instances two SVGENDQ objects
SVGENDQ svgen1, svgen2;

Initialization
To Instance pre-initialized objects
SVGENDQ svgen1 = SVGENDQ_DEFAULTS;
SVGENDQ svgen2 = SVGENDQ_DEFAULTS;

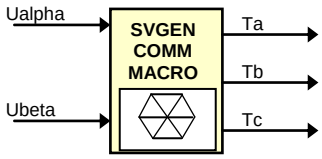
Invoking the computation macro
SVGEN_MACRO (svgen1);
SVGEN_MACRO (svgen2);

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        svgen1.Ualpha = Ualpha1;          // Pass inputs to svgen1
        svgen1.Ubeta = Ubeta1;            // Pass inputs to svgen1

        svgen2.Ualpha = Ualpha2;          // Pass inputs to svgen2
        svgen2.Ubeta = Ubeta2;            // Pass inputs to svgen2

        SVGEN_MACRO (svgen1);             // Call compute macro for svgen_dq1
        SVGEN_MACRO (svgen2);             // Call compute macro for svgen2

        Ta1 = svgen_dq1.Ta;               // Access the outputs of svgen_dq1
        Tb1 = svgen_dq1.Tb;               // Access the outputs of svgen_dq1
        Tc1 = svgen_dq1.Tc;               // Access the outputs of svgen_dq1

        Ta2 = svgen2.Ta;                  // Access the outputs of svgen2
        Tb2 = svgen2.Tb;                  // Access the outputs of svgen2
        Tc2 = svgen2.Tc;                  // Access the outputs of svgen2
}
```

Description

This module calculates the appropriate duty ratios needed to generate a given stator reference voltage using common mode voltage. The stator reference voltage is described by it's (α,β) components, Ualpha and Ubeta. Note that the input range for this particular macro is ± 2/sqrt(3).



Availability

This IQ module is available in one interface format:

- The C interface version

Module Properties

Type: Target Independent, Application Independent

Target Devices: 28x Fixed and Floating Point devices

C Version File Names: svgen_comm.h

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

C Interface

Object Definition

The structure of SVGENCOMM object is defined by following structure definition

```
typedef struct {  _iq  Ualpha;     // Input: reference alpha-axis phase voltage
                  _iq  Ubeta;      // Input: reference beta-axis phase voltage
                  _iq  Ta; // Output: reference phase-a switching function
                  _iq  Tb; // Output: reference phase-b switching function
                  _iq  Tc; // Output: reference phase-c switching function
                  _iq  Va; // Variable: reference phase-a voltage
                  _iq  Vb; // Variable: reference phase-a voltage
                  _iq  Vc; // Variable: reference phase-a voltage
                  _iq  Vmax;     //Variable: max phase
                  _iq  Vmin;     //Variable: min phase
                  _iq  Vcomm;  //Variable: common mode voltage
} SVGENCOMM;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	Ualpha	Component of reference stator voltage vector on direct axis stationary reference frame.	GLOBAL_Q	80000000-7FFFFFFF
	Ubeta	Component of reference stator voltage vector on quadrature axis stationary reference frame.	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Ta	Duty ratio of PWM1 (CMPR1 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
	Tb	Duty ratio of PWM3 (CMPR2 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
	Tc	Duty ratio of PWM5 (CMPR3 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
Internal	Va	Reference phase-a voltage	GLOBAL_Q	80000000-7FFFFFFF
	Vb	Reference phase-b voltage	GLOBAL_Q	80000000-7FFFFFFF
	Vc	Reference phase-c voltage	GLOBAL_Q	80000000-7FFFFFFF
	Vmax	Maximum of phase voltages	GLOBAL_Q	80000000-7FFFFFFF
	Vmin	Minimum of phase voltages	GLOBAL_Q	80000000-7FFFFFFF
	Vcomm	Common mode voltage	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

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

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

Module Usage

Instantiation
The following example instances two SVGEN_COMM objects
SVGEN_COMM svgen_comm1, svgen_comm2;

Initialization
To Instance pre-initialized objects
SVGEN_COMM svgen_comm1 = SVGEN_COMM_DEFAULTS;
SVGEN_COMM svgen_comm2 = SVGEN_COMM_DEFAULTS;

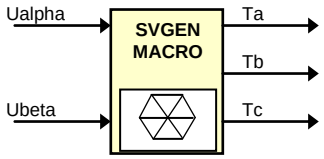
Invoking the computation macro
SVGEN_COMM_MACRO (svgen_comm1);
SVGEN_COMM_MACRO (svgen_comm2);

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

```
main()
{

}
void interrupt periodic_interrupt_isr()
{
        svgen_comm.Ualpha = Ualpha1;            // Pass inputs to svgen_comm
        svgen_comm.Ubeta = Ubeta1;              // Pass inputs to svgen_comm

        svgen_comm2.Ualpha = Ualpha2;           // Pass inputs to svgen_comm2
        svgen_comm2.Ubeta = Ubeta2;             // Pass inputs to svgen_comm2

        SVGEN_MACRO (svgen_comm1);              // Call compute macro for svgen_comm1
        SVGEN_MACRO (svgen_comm2);              // Call compute macro for svgen_comm2

        Ta1 = svgen_comm1.Ta;                   // Access the outputs of svgen_comm1
        Tb1 = svgen_comm1.Tb;                   // Access the outputs of svgen_comm1
        Tc1 = svgen_comm1.Tc;                   // Access the outputs of svgen_comm1

        Ta2 = svgen_comm2.Ta;                   // Access the outputs of svgen_comm2
        Tb2 = svgen_comm2.Tb;                   // Access the outputs of svgen_comm2
        Tc2 = svgen_comm2.Tc;                   // Access the outputs of svgen_comm2
}
```

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, Ualpha and Ubeta. Different than the regular SVGEN, this modulation technique keeps one of the three switches off during the entire 120^0 to minimize switching losses. This technique is also known as DPWMmin in the literature.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: aci_se.h

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

Object Definition

The structure of SVGENDPWM object is defined by following structure definition

typedef struct

```
{ _iq  Ualpha;          // Input: reference alpha-axis phase voltage
  _iq  Ubeta;           // Input: reference beta-axis phase voltage
  _iq  Ta;              // Output: reference phase-a switching function
  _iq  Tb;              // Output: reference phase-b switching function
  _iq  Tc;              // Output: reference phase-c switching function
} SVGENDPWM;
```

Module Terminal Variables/Macros

Item	Name	Description	Format*	Range(Hex)
Inputs	Ualpha	Component of reference stator voltage vector on direct axis stationary reference frame.	GLOBAL_Q	80000000-7FFFFFFF
	Ubeta	Component of reference stator voltage vector on quadrature axis stationary reference frame.	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Ta	Duty ratio of PWM1 (CMPR1 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
	Tb	Duty ratio of PWM3 (CMPR2 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
	Tc	Duty ratio of PWM5 (CMPR3 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
Variable	tmp1,2,3	Internal variable	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

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

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

Module Usage

Instantiation
The following example instances two SVGENDPWM objects
SVGENDPWM svgendpwm1, svgendpwm2;

Initialization
To Instance pre-initialized objects
SVGENDPWM svgendpwm1 = SVGENDPWM_DEFAULTS;
SVGENDPWM svgendpwm2 = SVGENDPWM_DEFAULTS;

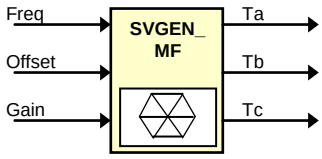
Invoking the computation macro
SVGENDPWM_MACRO (svgendpwm1);
SVGENDPWM_MACRO (svgendpwm2);

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        svgendpwm1.Ualpha = Ualpha1;          // Pass inputs to svgendpwm1
        svgendpwm1.Ubeta = Ubeta1;           // Pass inputs to svgendpwm1

        svgendpwm2.Ualpha = Ualpha2;          // Pass inputs to svgendpwm2
        svgendpwm2.Ubeta = Ubeta2;           // Pass inputs to svgendpwm2

        SVGENDPWM_MACRO (svgendpwm1);         // Call compute macro for svgendpwm1
        SVGENDPWM_MACRO (svgendpwm2);         // Call compute macro for svgendpwm2

        Ta1 = svgendpwm1.Ta;                  // Access the outputs of svgen_dpwm1
        Tb1 = svgendpwm1.Tb;                  // Access the outputs of svgen_dpwm1
        Tc1 = svgendpwm1.Tc;                  // Access the outputs of svgen_dpwm1

        Ta2 = svgendpwm2.Ta;                  // Access the outputs of svgendpwm2
        Tb2 = svgendpwm2.Tb;                  // Access the outputs of svgendpwm2
        Tc2 = svgendpwm2.Tc;                  // Access the outputs of svgendpwm2
}
```

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 its magnitude and frequency.



Availability

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: svgen_mf.h

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

C Interface

Object Definition

The structure of SVGENMF object is defined by following structure definition

```
typedef struct  { _iq  Gain;            // Input: reference gain voltage
                  _iq  Offset;          // Input: reference offset voltage
                  _iq  Freq;            // Input: reference frequency
                  _iq  FreqMax;         // Parameter: Maximum step angle
                  _iq  Alpha;           // History: Sector angle
                  _iq  NewEntry;        // History: Sine (angular) look-up pointer
                  Uint32  SectorPointer; // History: Sector number (Q0)
                  _iq  Ta;              // Output: reference phase-a switching function
                  _iq  Tb;              // Output: reference phase-b switching function
                  _iq  Tc;              // Output: reference phase-c switching function
                } SVGENMF;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	Gain	reference gain voltage	GLOBAL_Q	80000000-7FFFFFFF
	Offset	reference offset voltage	GLOBAL_Q	80000000-7FFFFFFF
	Freq	reference frequency	GLOBAL_Q	80000000-7FFFFFFF
Outputs	Ta	Duty ratio of PWM1 (CMPR1 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
	Tb	Duty ratio of PWM3 (CMPR2 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
	Tc	Duty ratio of PWM5 (CMPR3 register value as a fraction of associated period register, TxPR, value).	GLOBAL_Q	80000000-7FFFFFFF
SVGENMF parameters	FreqMax	FreqMax = 6*fb*T	GLOBAL_Q	00000000-7FFFFFFF
Internal	Alpha	Sector angle	GLOBAL_Q	80000000-7FFFFFFF
	NewEntry	Sine (angular) look-up pointer	GLOBAL_Q	80000000-7FFFFFFF
	SectorPointer	Sector number	Q0	0-5

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

SVGENMF

The module definition is created as a data type. This makes it convenient to instance an interface to space vector generator using magnitude and frequency. To create multiple instances of the module simply declare variables of type SVGENMF.

SVGENMF_DEFAULTS

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

Module Usage

Instantiation

The following example instances two SVGENMF objects

SVGENMF svgen_mf1, svgen_mf2;

Initialization

To Instance pre-initialized objects

SVGENMF svgen_mf1 = SVGENMF_DEFAULTS;

SVGENMF svgen_mf2 = SVGENMF_DEFAULTS;

Invoking the computation function

SVGENMF_MACRO(svgen_mf1);

SVGENMF_MACRO(svgen_mf2);

Example

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

```
main()
{

}
void interrupt periodic_interrupt_isr()
{
        svgen_mf1.Gain = gain1;          // Pass inputs to svgen_mf1
        svgen_mf1.Freq = offset1;        // Pass inputs to svgen_mf1

        svgen_mf2.Gain = gain2;          // Pass inputs to svgen_mf2
        svgen_mf2.Freq = offset2;        // Pass inputs to svgen_mf2

        SVGENMF_MACRO(svgen_mf1);        // Call compute macro for svgen_mf1
        SVGENMF_MACRO(svgen_mf2);        // Call compute macro for svgen_mf2

        Ta1 = svgen_mf1.Ta;              // Access the outputs of svgen_mf1
        Tb1 = svgen_mf1.Tb;              // Access the outputs of svgen_mf1
        Tc1 = svgen_mf1.Tc;              // Access the outputs of svgen_mf1

        Ta2 = svgen_mf2.Ta;              // Access the outputs of svgen_mf2
        Tb2 = svgen_mf2.Tb;              // Access the outputs of svgen_mf2
        Tc2 = svgen_mf2.Tc;              // Access the outputs of svgen_mf2
}
```

Technical Background

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.

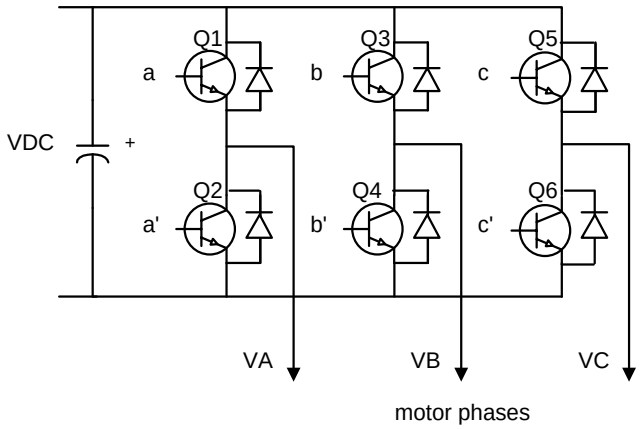


Figure 3: Power circuit topology for a three-phase VSI

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.

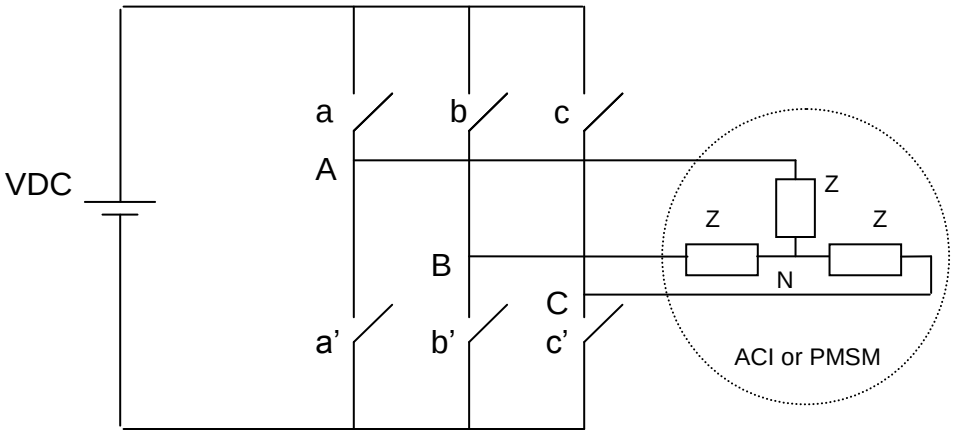


Figure 2: Power bridge for a three-phase VSI

For the three phase power inverter configurations shown in Figure 1 and Figure 2, 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 1.

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

Table 1: Device on/off patterns and resulting instantaneous voltages of a 3-phase power inverter

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 2.

c	b	a	V_{ds}	V_{qs}	Vector
0	0	0	0	0	O_0
0	0	1	$2\ V_{DC}/3$	0	U_0
0	1	0	$-V_{DC}/3$	$V_{DC}/\sqrt{3}$	U_{120}
0	1	1	$V_{DC}/3$	$V_{DC}/\sqrt{3}$	U_{60}
1	0	0	$-V_{DC}/3$	$-V_{DC}/\sqrt{3}$	U_{240}
1	0	1	$V_{DC}/3$	$-V_{DC}/\sqrt{3}$	U_{300}
1	1	0	$-2\ V_{DC}/3$	0	U_{180}
1	1	1	0	0	O_{111}

Table 2: Switching patterns, corresponding space vectors and their (d-q) components

These values of V_{ds} and $V_{qs,}$ listed in Table 2, 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 2. 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 3.

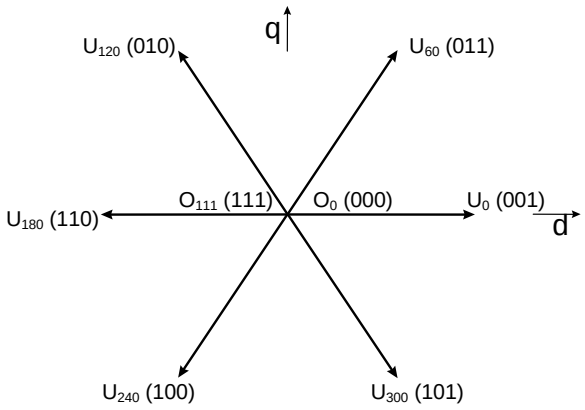


Figure 3: Basic space vectors

In Figure 3, 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 3, 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 2, 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 3.

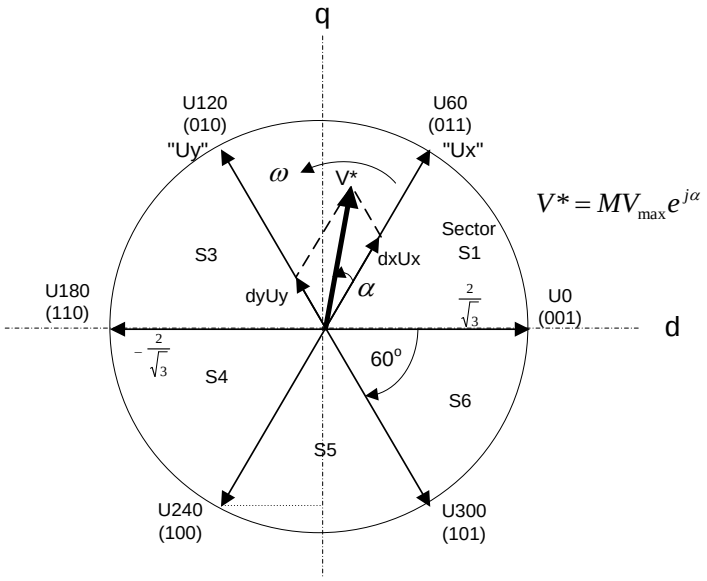


Figure 3: Projection of the reference voltage vector

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 Fig. 4 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.

As shown in Fig. 4, 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:

$$\omega = \frac{STEP \times fs}{6 \times 2^m}$$

Where f_s = sampling frequency (i.e. PWM frequency), STEP = angle stepping increment, and m = # bits in the integration register.

For example, if f_s = 24KHz, m = 16 bits & STEP ranges from 0→2048 then the resulting angular frequencies will be as shown in Table 3.

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

Table 3: Frequency mapping

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 4 shows the theoretical resolution which results from various sizes of m.

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

Table 4: Resolution of frequency mapping

For IQmath implementation, the maximum step size in per-unit, *FreqMax*, for a given base frequency, fb and a defined GLOBAL_Q number is therefore computed as follows:

$$\mathrm{FreqMax} = 6 \times \mathrm{f_b} \times \mathrm{T_s} \times 2^{GLOBAL\_Q}$$

Equivalently, by using _IQ() function for converting from a floating-point number to a _iq number, the *FreqMax* can also be computed as

$$\mathrm{FreqMax} = \_IQ(6 \times \mathrm{f_b} \times \mathrm{T_s})$$

where Ts is the sampling period (sec).

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 28xx can be determined. The switching pattern in Figure 4 is adopted here and is implemented with the Full Compare Units of 28xx. A set of 3 new compare values, Ta, Tb and Tc, need to be calculated every PWM period to generate this switching pattern.

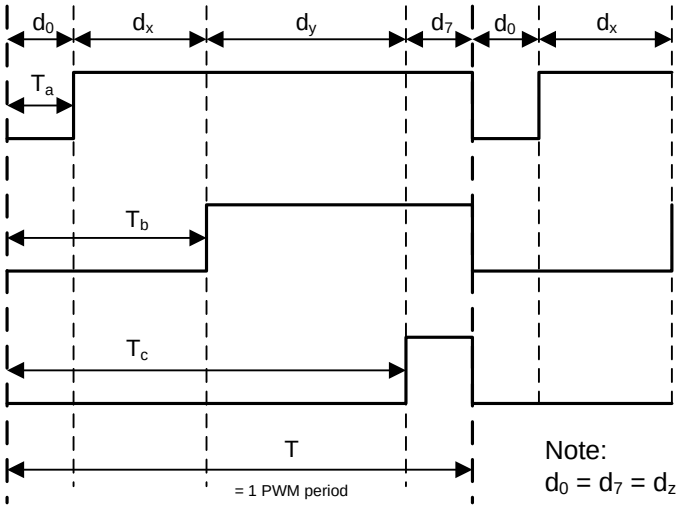


Figure 4: PWM output switching pattern

From Figure 5, it can be seen:

$$Ta = (T - dx - dy)/2$$

$$Tb = dx + Ta$$

$$Tc = T - Ta$$

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 5 depicts this determination.

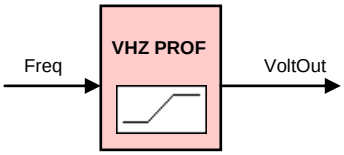
Sectors	U_0, U_{60}	U_{60}, U_{120}	U_{120}, U_{180}	U_{180}, U_{240}	U_{240}, U_{300}	U_{300}, U_0
Ta	T1	dy+Tb	T-Tb	T-Tc	dx+Tc	T1
Tb	dx+Ta	T1	T1	dy+Tc	T-Tc	T-Ta
Tc	T-Ta	T-Tb	dx+Tb	T1	T1	dy+Ta

Table 5: Calculation of duty cycle for different sectors

The switching pattern shown in Figure 5 is an asymmetric PWM implementation. However, 28xx 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

C interface version

Module Properties

Type: Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: vhzprof.h

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

C Interface

Object Definition

The structure of VHZPROF object is defined by following structure definition

```
typedef struct  { _iq  Freq;        // Input: Input Frequency
                  _iq  VoltOut;     // Output: Output voltage
                  _iq  LowFreq;     // Parameter: Low Frequency
                  _iq  HighFreq;    // Parameter: High Frequency at rated voltage
                  _iq  FreqMax;     // Parameter: Maximum Frequency
                  _iq  VoltMax;     // Parameter: Rated voltage
                  _iq  VoltMin;     // Parameter: Voltage at low Frequency range
                  } VHZPROF;
```

Item	Name	Description	Format*	Range(Hex)
Inputs	Freq	Input Frequency	GLOBAL_Q	80000000-7FFFFFFF
Outputs	VoltOut	Output voltage	GLOBAL_Q	80000000-7FFFFFFF
VHZPROF parameter	LowFreq	Low Frequency	GLOBAL_Q	80000000-7FFFFFFF
	HighFreq	High Frequency at rated voltage	GLOBAL_Q	80000000-7FFFFFFF
	FreqMax	Maximum Frequency	GLOBAL_Q	80000000-7FFFFFFF
	VoltMax	Rated voltage	GLOBAL_Q	80000000-7FFFFFFF
	VoltMin	Voltage at low Frequency range	GLOBAL_Q	80000000-7FFFFFFF

*GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.

Special Constants and Data types

VHZPROF

The module definition is created as a data type. This makes it convenient to instance an interface to volt/hertz profile. To create multiple instances of the module simply declare variables of type VHZPROF.

VHZPROF_DEFAULTS

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

Module Usage

Instantiation

The following example instances two VHZPROF objects

VHZPROF vhz1, vhz2;

Initialization

To Instance pre-initialized objects

VHZPROF vhz1 = VHZPROF_DEFAULTS;

VHZPROF vhz2 = VHZPROF_DEFAULTS;

Invoking the computation macro

VHZ_PROF_MACRO(vhz1);

VHZ_PROF_MACRO(vhz2);

Example

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

```
main()
{

}

void interrupt periodic_interrupt_isr()
{
        vhz1.Freq = Freq1;              // Pass inputs to vhz1
        vhz2.Freq = Freq2;              // Pass inputs to vhz2

        VHZ_PROF_MACRO(vhz1);               // Call compute macro for vhz1
        VHZ_PROF_MACRO(vhz2);               // Call compute macro for vhz2

        VoltOut1 = vhz1.VoltOut;        // Access the outputs of vhz1
        VoltOut2 = vhz2.VoltOut;        // Access the outputs of vhz2
}
```

Technical Background

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

$$\Lambda \approx \frac{V}{\omega} = \frac{1}{2\pi}\frac{V}{f}$$

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 1.

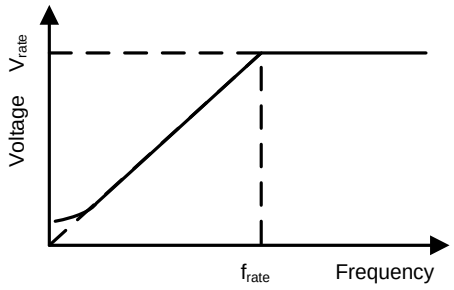


Figure 1: Voltage versus frequency under the constant V/Hz principle

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 2. So by regulating the slip speed, the torque and speed of an AC Induction motor can be controlled with the constant V/Hz principle.

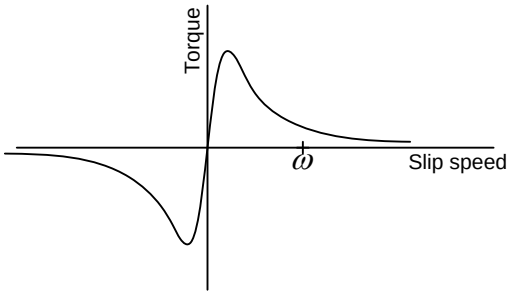


Figure 2: Torque versus slip speed of an induction motor with constant stator flux

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 1 is modified by imposing a lower limit on frequency. This is shown in Figure 3. 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.

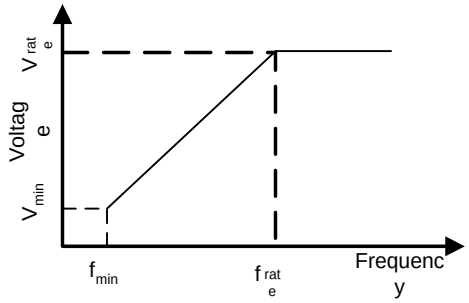


Figure 3: Modified V/Hz profile

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}.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.