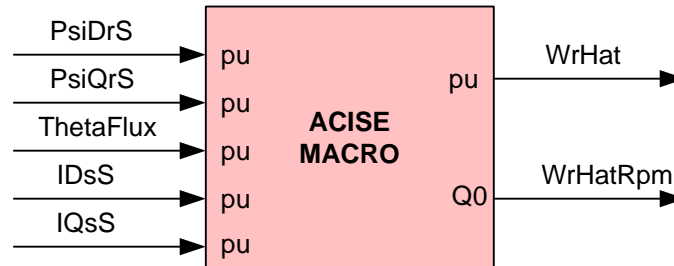


**Description**

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**

This IQ module is available in one interface:

- 1) The C interface version

**Module Properties**

**Type:** Target Independent, Application Dependent

**Target Devices:** 28x Fixed Point or Piccolo

**C Version File Names:** aci\_se.h

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

## C Interface

### 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)
            } ACISE;

typedef ACISE *ACISE_handle;
```

#### Module Terminal Variables

Item	Name	Description	Format	Range(Hex)
<b>Inputs</b>	PsiDrS	stationary d-axis rotor flux	GLOBAL_Q	80000000-7FFFFFFF
	PsiQrS	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
<b>Outputs</b>	WrHat	estimated rotor speed	GLOBAL_Q	80000000-7FFFFFFF
	WrHatRpm	estimated rotor speed in rpm	Q0	80000000-7FFFFFFF
<b>ACISE parameter</b>	K1	$K1 = 1/(Wb \cdot Tr)$	GLOBAL_Q	80000000-7FFFFFFF
	K2	$K2 = 1/(fb \cdot T)$	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
<b>Internal</b>	OldThetaFlux	previous rotor flux linkage angle	GLOBAL_Q	00000000-7FFFFFFF (0 – 360 degree)
	WPsi	synchronous speed	GLOBAL_Q	80000000-7FFFFFFF
	SquaredPsi	squared magnitude of rotor flux	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\_handle**

User defined Data type of pointer to ACISE module

### **ACISE\_DEFAULTS**

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

## Methods

### **ACISE\_MACRO(ACISE\_handle);**

This definition implements one method viz., the speed estimator of Induction Motor computation macro. The input argument to this macro is the module handle.

## 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;

typedef ACISE_CONST *ACISE_CONST_handle;
```

### 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\_handle**

User defined Data type of pointer to ACISE\_CONST module

### **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.

## Methods

### **ACISE\_CONST\_MACRO(ACISE\_CONST\_handle);**

This definition implements one method viz., the speed estimator of Induction Motor constant computation macro. The input argument to this macro is the module handle.

## 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

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

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

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  $\omega_r$  is electrically angular velocity of rotor (rad/sec), and  $R_r$  is rotor resistance ( $\Omega$ ). 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

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

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

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



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

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

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

where  $u = \frac{\lambda_{qr}^s}{\lambda_{dr}^s}$ , yields

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

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

$$\omega_e = \frac{d\theta_{\lambda_r}}{dt} = \omega_r + \frac{1}{(\lambda_r^s)^2} \frac{L_m}{\tau_r} (\lambda_{dr}^s i_{qs}^s - \lambda_{qr}^s i_{ds}^s) \quad (14)$$

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.

$$T_e = \frac{3}{2} \frac{p}{2} \frac{L_m}{L_r} (\lambda_{dr}^s i_{qs}^s - \lambda_{qr}^s i_{ds}^s) \quad (15)$$

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

$$\omega_r = \omega_e - \frac{1}{(\lambda_r^s)^2} \frac{L_m}{\tau_r} (\lambda_{dr}^s i_{qs}^s - \lambda_{qr}^s i_{ds}^s) \quad (16)$$

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

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

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

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

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

The per-unit synchronous speed can be calculated as

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

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

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

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

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

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

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  $360^\circ$  to  $0^\circ$  (CCW) or from  $0^\circ$  to  $360^\circ$  (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.

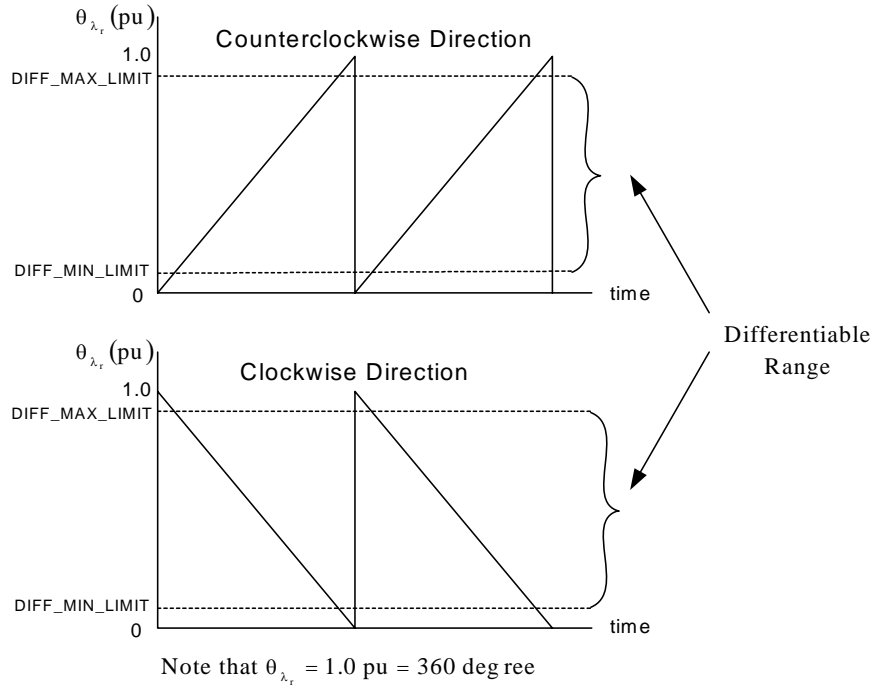


Figure 1: The waveforms of rotor flux angle in both directions

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 1<sup>st</sup>-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 1<sup>st</sup>-order low-pass filter is as

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

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

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

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 ( $\omega_b$ )

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.c, aci\_se.h). The software module requires that both input and output variables are in per unit values.

	<b>Equation Variables</b>	<b>Program Variables</b>
<b>Inputs</b>	$\lambda_{dr}^s$	PsiDrS
	$\lambda_{qr}^s$	PsiQrS
	$\theta_{\lambda_r}$	ThetaFlux
	$i_{ds}^s$	IDsS
	$i_{qs}^s$	IQsS
<b>Output</b>	$\omega_r$	WrHat
<b>Others</b>	$(\lambda_r^s)^2$	SquaredPsi
	$\omega_e$	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.