RESUMEN

El anexo B presenta el código del control que se ha de introducir en cada práctica. Están los cuatro códigos distintos que se han de implementar en el total de sesiones prácticas. El orden, en el que se incluyen en el anexo, es el mismo orden en el que aparecen en el libro de prácticas.

También incluye los documentos en formato pdf que son entregados durante las sesiones de clase para ayudar al alumnado a implementar los módulos. En los documentos se explica cada una de las funciones que se utilizan en la implementación del código del control, así como las variables que tienen, junto con otros parámetros de interés.
SUMARIO

RESUMEN ........................................................................................................................... 1

SUMARIO ............................................................................................................................ 3

B. ANEXO B ........................................................................................................................... 5

B.1. Código del control ......................................................................................................... 5

B.1.1. Control V/f en lazo abierto ...................................................................................... 5

B.1.2. Lazo de par .............................................................................................................. 13

B.1.3. Medida del ángulo. .................................................................................................. 15

B.1.4. Lazo de velocidad .................................................................................................. 17

B.2. Documentos pdf de los módulos ............................................................................... 19
B. ANEXO B

B.1. Código del control

B.1.1. Control V/f en lazo abierto

/*
===================================================================
===
System Name: PMSM31

File Name: PMSM3_1.C

Description: Primary system file for the Real Implementation of Sensored Field Orientation Control for a Three Phase Permanent-Magnet Synchronous Motor (PMSM)

Originator: Digital control systems Group - Texas Instruments

===================================================================
==================
History:
-------------------------------------------------------------------------------------
08-25-2003  Release  Rev 2.0

===================================================================  */

#include "IQmathLib.h"    /* Include header for IQmath library */
#include "DSP28_Device.h"
#include "pmsm.h"
#include "parameter.h"
#include "build.h"
#include "dlog4ch.h"

// Prototype statements for functions found within this file.
interrupt void EvaTimer1(void);
interrupt void EvaTimer2(void);
void ADC_calibration(void);

// temporary variables used to update the data LOG module
int temp_buf1,temp_buf2,temp_buf3,temp_buf4;

// Global variables used in this system
float Vd_testing = 0;    /* Vd testing (pu) */
float Vq_testing = 0.2;  /* Vq testing (pu) */
float Id_ref = 0;        /* Id reference (pu) */
float Iq_ref = 0.14;     /* Iq reference (pu) */
float speed_ref = 0.2;           /* Speed reference (pu) */
int speed_ref_rpm;             /* Speed reference (rpm) */
float T = 0.001/ISR_FREQUENCY;   /* Samping period (sec), see parameter.h */
float Kp_i=0.023, Ki_i=0.01, Kd_i=0.0;  /* PID constants for inner current/torque loop */
float Kp_s=0.5, Ki_s=0.0005, Kd_s=0.0;  /* PID constants for outer speed loop */

int isr_ticker = 0;
int back_ticker = 0;

volatile int enable_flg=0;
volatile int ADC_calibration_flg=0;
int ADC_buf[256];

int counter=0;

int speed_loop_ps = 2;       // Speed loop prescaler
int speed_loop_count = 1;    // Speed loop counter

// Create an instance of DATALOG Module
DLOG_4CH dlog = DLOG_4CH_DEFAULTS;

CLARKE clarke1 = CLARKE_DEFAULTS;
PARK park1 = PARK_DEFAULTS;
IPARK ipark1 = IPARK_DEFAULTS;

PIDREG3 pid1_id = PIDREG3_DEFAULTS;
PIDREG3 pid1_iq = PIDREG3_DEFAULTS;
PIDREG3 pid1_spd = PIDREG3_DEFAULTS;

PWMGEN pwm1 = PWMGEN_DEFAULTS;

SVGENDQ svgen_dq1 = SVGENDQ_DEFAULTS;
QEP qep1 = QEP_DEFAULTS;
SPEED_MEAS_QEP speed1 = SPEED_MEAS_QEP_DEFAULTS;

RMPCNTL rc1 = RMPCNTL_DEFAULTS;
RAMPGEN rg1 = RAMPGEN_DEFAULTS;

ILEG2DCBUSMEAS_VCON ilg2_vdc_vcon1 = ILEG2DCBUSMEAS_VCON_DEFAULTS;

void main(void) {
  // Initialize System Control registers, PLL, WatchDog, Clocks to default state:
  // This function is found in the DSP28_SysCtrl.c file.
  InitSysCtrl();

  // HISPCP prescale register settings, normally it will be set to default values
  EALLOW;   // This is needed to write to EALLOW protected registers
  SysCtrlRegs.HISPCP.all = 0x0000;   /* SYSCLKOUT/1 */
EDIS; // This is needed to disable write to EALLOW protected registers

// Disable and clear all CPU interrupts:
DINT;
IER = 0x0000;
IFR = 0x0000;

// Initialize Pie Control Registers To Default State:
// This function is found in the DSP28_PieCtrl.c file.
InitPieCtrl();

// Initialize the PIE Vector Table To a Known State:
// This function is found in DSP28_PieVect.c.
// This function populates the PIE vector table with pointers
// to the shell ISR functions found in DSP28_DefaultIsr.c.
InitPieVectTable();

// User specific functions, Reassign vectors (optional), Enable Interrupts:

// Initialize EVA Timer 1:
// Setup Timer 1 Registers (EVA)
EvaRegs.GPTCONA.all = 0;

/* The CDM2480 has an over current protection that activate the */
/* Power Drive Protection INTerrupt of the DSP if needed        */
/* When powering up the CDM2480, this protection may be activated*/
/* therefore, in the DSP init code, we check the value of the    */
/* PDPINT pin and we stop if the active low signal is low       */
/* If this situation happens, the only solution is to unplug the */
/* board and to restart the system                              */
/* Check JP4 of eZdsp rev D which has to be connected to        */
/* provide 5V                                                   */
if(EvaRegs.COMCONA.bit.PDPINTASTATUS == 0) {
    ESTOP0;
}

/* For proper software operation, the F2812 eZdsp must be in    */
/* Microcontroller mode MP/MC = 0, after reset. Please, make sure */
/* that JP1 is on position 2-3 on the eZdsp                     */
if(XintfRegs.XINTCNF2.bit.MPNMC == 1) {
    ESTOP0;
}

/* Initialize the DATALOG module */
dlog.iptr1=&temp_buf1;
dlog.iptr2=&temp_buf2;
dlog.iptr3=&temp_buf3;
dlog.iptr4=&temp_buf4;
dlog.trig_value=0x10;
dlog.size=0x800;
dlog.init(&dlog);

// Set input qualifier
EALLOW;
GpioMuxRegs.GPDQUAL.all=0x003F; // Input qualifier
GpioMuxRegs.GPAQUAL.all=0x003F; // Input qualifier
EDIS;
EvaRegs.EVAIMRA.bit.PDPINTA = 1;
EvaRegs.EVAIFRA.bit.PDPINTA = 1;

// Waiting for enable flag set
while (enable_flg==0) {
    // Enable Underflow interrupt bits for GP timer 1
    EvaRegs.EVAIMRA.bit.T1UFINT = 1;
    EvaRegs.EVAIFRA.bit.T1UFINT = 1;

    // Enable CAP3 interrupt bits for GP timer 2
    EvaRegs.EVAIMRC.bit.CAP3INT = 1;
    EvaRegs.EVAIFRC.bit.CAP3INT = 1;
}

/* Initialize modules */
pwm1.n_period = SYSTEM_FREQUENCY*1000000*T/2; /* Perscaler X1 (T1), ISR period = T x 1 */
pwm1.init(&pwm1);

qep1.init(&qep1);
ilg2_vdc_vcon1.init(&ilg2_vdc_vcon1);

/* Initialize the RAMP_CTRL module */
rc1.rmp_dly_max = 10; // for emulated angle of ramg gen

/* Initialize the SPEED_FRQ module */
speed1.K1 = _IQ21(1/(BASE_FREQ*T));
speed1.K2 = _IQ(1/(1+T*2*PI*60)); /* Low-pass cut-off frequency = 60 Hz */
speed1.K3 = _IQ(1)-speed1.K2;
speed1.rpm_max = BASE_FREQ*(120/P);

/* Initialize the RAMPGEN module */
rg1.step_angle_max = _IQ(BASE_FREQ*T);

/* Initialize the PID_REG3 module for Id */
pid1_id.Kp_reg3 = _IQ(Kp_i);
pid1_id.Ki_reg3 = _IQ(Ki_i);
pid1_id.Kd_reg3 = _IQ(Kd_i);
pid1_id.Kc_reg3 = _IQ(0.5);
pid1_id.pid_out_max = _IQ(0.30);
pid1_id.pid_out_min = _IQ(-0.30);
/* Initialize the PID_REG3 module for Iq */

    pid1_iq.Kp_reg3 = _IQ(Kp_i);
    pid1_iq.Ki_reg3 = _IQ(Ki_i);
    pid1_iq.Kd_reg3 = _IQ(Kd_i);
    pid1_iq.Kc_reg3 = _IQ(0.5);
    pid1_iq.pid_out_max = _IQ(0.95);
    pid1_iq.pid_out_min = _IQ(-0.95);

/* Initialize the PID_REG3 module for speed */

    pid1_spd.Kp_reg3 = _IQ(Kp_s);
    pid1_spd.Ki_reg3 = _IQ(Ki_s);
    pid1_spd.Kd_reg3 = _IQ(Kd_s);
    pid1_spd.Kc_reg3 = _IQ(0.5);
    pid1_spd.pid_out_max = _IQ(1);
    pid1_spd.pid_out_min = _IQ(-1);

// Reassign ISRs.
// Reassign the PIE vector for T1UFINT and CAPINT3 to point to a different
// ISR then the shell routine found in DSP28_DefaultIsr.c.
// This is done if the user does not want to use the shell ISR routine
// but instead wants to use their own ISR. This step is optional:

    EALLOW;  // This is needed to write to EALLOW protected registers
    PieVectTable.T1UFINT = &EvaTimer1;
    PieVectTable.CAPINT3 = &EvaTimer2;
    EDIS;   // This is needed to disable write to EALLOW protected registers

    PieCtrlRegs. PIEIER1.all = M_INT1;
    PieCtrlRegs. PIEIER2.all = M_INT6;
    PieCtrlRegs. PIEIER3.all = M_INT7;
    IER |= (M_INT2 | M_INT3 | M_INT1);

    EINT;   // Enable Global interrupt INTM
    ERTM; // Enable Global realtime interrupt DBGM

    // IDLE loop. Just sit and loop forever:
    for(;;) {
        backTicker++;
        // Refresh the PID_REG3 constants for Id loop
        pid1_id.Kp_reg3 = _IQ(Kp_i);
pid1_id.Ki_reg3 = _IQ(Ki_i);
pid1_id.Kd_reg3 = _IQ(Kd_i);

// Refresh the PID_REG3 constants for Iq loop
pid1_iq.Kp_reg3 = _IQ(Kp_i);
pid1_iq.Ki_reg3 = _IQ(Ki_i);
pid1_iq.Kd_reg3 = _IQ(Kd_i);

// Refresh the PID_REG3 constants for speed loop
pid1_spd.Kp_reg3 = _IQ(Kp_s);
pid1_spd.Ki_reg3 = _IQ(Ki_s);
pid1_spd.Kd_reg3 = _IQ(Kd_s);

}}

interrupt void EvaTimer1(void) {
    isr_tick++;  

    if(ADC_calibration_flg == 0) {
        ilg2_vdc_vcon1.read(&ilg2_vdc_vcon1);
        ADC_calibration();
    }
    else {
        rc1.target_value = _IQ(speed_ref); // target_value is in IQ
        rc1.calc(&rc1);

        rg1.rpm_freq = rc1.setpt_value;
        rg1.calc(&rg1);

        ilg2_vdc_vcon1.read(&ilg2_vdc_vcon1);
        ilg2_vdc_vcon1.lmeas_a = ilg2_vdc_vcon1.lmeas_a - ilg2_vdc_vcon1.lmeas_a_offset;
        ilg2_vdc_vcon1.lmeas_b = ilg2_vdc_vcon1.lmeas_b - ilg2_vdc_vcon1.lmeas_b_offset;

        clark1.as = _IQ15toIQ((long)ilg2_vdc_vcon1.lmeas_a);
        clark1.bs = _IQ15toIQ((long)ilg2_vdc_vcon1.lmeas_b);
        clark1.calc(&clark1);

        park1.ds = clark1.ds;
        park1.qs = clark1.qs;
        park1.ang = rg1.rpm_out;
        park1.calc(&park1);

        ipark1.de = _IQ(Vd_testing);
        ipark1.qe = _IQ(Vq_testing);
        ipark1.ang = rg1.rpm_out;
        ipark1.calc(&ipark1);

        svgen_dq1.Ualfa = ipark1.ds;
        svgen_dq1.Ubeta = ipark1.qs;
svgen_dq1.calc(&svgen_dq1);

pwm1.Mfunc_c1 = (int)_IQtoIQ15(svgen_dq1.Ta); // Mfunc_c1 is in Q15
pwm1.Mfunc_c2 = (int)_IQtoIQ15(svgen_dq1.Tb); // Mfunc_c2 is in Q15
pwm1.Mfunc_c3 = (int)_IQtoIQ15(svgen_dq1.Tc); // Mfunc_c3 is in Q15
pwm1.update(&pwm1);

// select which of the variables are being displayed through data LOG
temp_buf1 = (int)_IQtoIQ15(rg1.rmp_out); // temp_buf1 is in Q15
temp_buf2 = (int)_IQtoIQ15(svgen_dq1.Ta); // temp_buf2 is in Q15
temp_buf3 = (int)_IQtoIQ15(svgen_dq1.Ualfa); // temp_buf3 is in Q15
temp_buf4 = (int)_IQtoIQ15(svgen_dq1.Ubeta); // temp_buf4 is in Q15
// temp_buf3 = (int)(ilg2_vdc_vcon1.Imeas_a); // temp_buf3 is in Q15
// temp_buf4 = (int)(ilg2_vdc_vcon1.Imeas_b); // temp_buf4 is in Q15
}

speed_ref_rpm = (int)(speed1.rpm_max*speed_ref);

// update the data LOG module
dlog.update(&dlog);

// Enable more interrupts from this timer
EvaRegs.EVAIMRA.bit.T1UFINT = 1;

// Note: To be safe, use a mask value to write to the entire
// EVAIFRA register. Writing to one bit will cause a read-modify-write
// operation that may have the result of writing 1’s to clear
// bits other then those intended.
EvaRegs.EVAIFRA.all = BIT9;

// Acknowledge interrupt to receive more interrupts from PIE group 2
PieCtrlRegs.PIEACK.all |= PIEACK_GROUP2;
}

interrupt void EvaTimer2(void) {
    qep1.isr(&qep1);

    // Enable more interrupts from this timer
    EvaRegs.EVAIMRC.bit.CAP3INT = 1;

    // Note: To be safe, use a mask value to write to the entire
    // EVAIFRA register. Writing to one bit will cause a read-modify-write
    // operation that may have the result of writing 1's to clear
    // bits other then those intended.
    EvaRegs.EVAIFRC.all = BIT2;

    // Acknowledge interrupt to receive more interrupts from PIE group 3
    PieCtrlRegs.PIEACK.all |= PIEACK_GROUP3;
}
void ADC_calibration(void) {
    ADC_buf[counter] = ilg2_vdc_vcon1.Imeas_a;
    ADC_buf[counter + 128] = ilg2_vdc_vcon1.Imeas_b;
    counter++;
    if (counter == 127) {
        ADC_calibration_flg = 1;
        for(counter=0; counter<128; counter++) {
            ilg2_vdc_vcon1.Imeas_a_offset = ilg2_vdc_vcon1.Imeas_a_offset +
            ADC_buf[counter];
            ilg2_vdc_vcon1.Imeas_b_offset = ilg2_vdc_vcon1.Imeas_b_offset +
            ADC_buf[counter + 128];
        }
        ilg2_vdc_vcon1.Imeas_a_offset = ilg2_vdc_vcon1.Imeas_a_offset>>7;
        ilg2_vdc_vcon1.Imeas_b_offset = ilg2_vdc_vcon1.Imeas_b_offset>>7;
        counter = 0;
    }
}
B.1.2. Lazo de par

// Torque Closed Loop

rc1.target_value = _IQ(speed_ref); // target_value is in IQ
rc1.calc(&rc1);

rg1.rmp_freq = rc1.setpt_value;
rg1.calc(&rg1);

ilg2_vdc_vcon1.read(&ilg2_vdc_vcon1);
ilg2_vdc_vcon1.Imeas_a = ilg2_vdc_vcon1.Imeas_a - ilg2_vdc_vcon1.Imeas_a_offset;
ilg2_vdc_vcon1.Imeas_b = ilg2_vdc_vcon1.Imeas_b - ilg2_vdc_vcon1.Imeas_b_offset;

clarke1.as = _IQ15toIQ((long)ilg2_vdc_vcon1.Imeas_a);
clarke1.bs = _IQ15toIQ((long)ilg2_vdc_vcon1.Imeas_b);
clarke1.calc(&clarke1);

park1.ds = clarke1.ds;
park1.qs = clarke1.qs;
park1.ang = rg1.rmp_out;
park1.calc(&park1);

pid1_iq.pid_ref_reg3 = _IQ(Iq_ref);
pid1_iq.pid_fdb_reg3 = park1.qe;
pid1_iq.calc(&pid1_iq);

pid1_id.pid_ref_reg3 = _IQ(Id_ref);
pid1_id.pid_fdb_reg3 = park1.de;
pid1_id.calc(&pid1_id);

ipark1.de = pid1_id.pid_out_reg3;
ipark1.qe = pid1_iq.pid_out_reg3;
ipark1.ang = rg1.rmp_out;
ipark1.calc(&ipark1);

svgen_dq1.Ualfa = ipark1.ds;
svgen_dq1.Ubeta = ipark1.qs;
svgen_dq1.calc(&svgen_dq1);

pwm1.Mfunc_c1 = (int)_IQtoIQ15(svgen_dq1.Ta); // Mfunc_c1 is in Q15
pwm1.Mfunc_c2 = (int)_IQtoIQ15(svgen_dq1.Tb); // Mfunc_c2 is in Q15
pwm1.Mfunc_c3 = (int)_IQtoIQ15(svgen_dq1.Tc); // Mfunc_c3 is in Q15
pwm1.update(&pwm1);
// select which of the variables are being displayed through data LOG
    temp_buf1 = (int)_IQtoIQ15(rg1.rmp_out);  // temp_buf1 is in Q15
    temp_buf2 = (int)_IQtoIQ15(svgen_dq1.Ta);  // temp_buf2 is in Q15
    temp_buf3 = (int)(ilg2_vdc_vcon1.Imeas_a);  // temp_buf3 is in Q15
    temp_buf4 = (int)(ilg2_vdc_vcon1.Imeas_b);  // temp_buf4 is in Q15
B.1.3. **Medida del ángulo.**

// Torque Closed Loop and Speed Measurement

rc1.target_value = _IQ(speed_ref); // target_value is in IQ
rc1.calc(&rc1);

rg1.rmp_freq = rc1.setpt_value;
rg1.calc(&rg1);

qep1.calc(&qep1);

speed1.theta_elec = _IQ15toIQ((long)qep1.theta_elec);
speed1.dir_QEP = (long)(qep1.dir_QEP);
speed1.calc(&speed1);

ilg2_vdc_vcon1.read(&ilg2_vdc_vcon1);
ilg2_vdc_vcon1.Imeas_a = ilg2_vdc_vcon1.Imeas_a - ilg2_vdc_vcon1.Imeas_a_offset;
ilg2_vdc_vcon1.Imeas_b = ilg2_vdc_vcon1.Imeas_b - ilg2_vdc_vcon1.Imeas_b_offset;

clarke1.as = _IQ15toIQ((long)ilg2_vdc_vcon1.Imeas_a);
clarke1.bs = _IQ15toIQ((long)ilg2_vdc_vcon1.Imeas_b);
clarke1.calc(&clarke1);

park1.ds = clarke1.ds;
park1.qs = clarke1.qs;
park1.ang = rg1.rmp_out;
park1.calc(&park1);

pid1_iq.pid_ref_reg3 = _IQ(Iq_ref);
pid1_iq.pid_fdb_reg3 = park1.qe;
pid1_iq.calc(&pid1_iq);

pid1_id.pid_ref_reg3 = _IQ(Id_ref);
pid1_id.pid_fdb_reg3 = park1.de;
pid1_id.calc(&pid1_id);

ipark1.de = pid1_id.pid_out_reg3;
ipark1.qe = pid1_iq.pid_out_reg3;
ipark1.ang = rg1.rmp_out;
ipark1.calc(&ipark1);

svgen_dq1.Ualfa = ipark1.ds;
svgen_dq1.Ubeta = ipark1.qs;
svgen_dq1.calc(&svgen_dq1);

pwm1.Mfunc_c1 = (int)_IQtoIQ15(svgen_dq1.Ta); // Mfunc_c1 is in Q15
pwm1.Mfunc_c2 = (int)_IQtoIQ15(svgen_dq1.Tb); // Mfunc_c2 is in Q15
pwm1.Mfunc_c3 = (int)_IQtoIQ15(svgen_dq1.Tc); // Mfunc_c3 is in Q15
pwm1.update(&pwm1);

// select which of the variables are being displayed through data LOG
  temp_buf1 = (int)_IQtoIQ15(rg1.rmp_out); // temp_buf1 is in Q15
  temp_buf2 = (int)_IQtoIQ15(speed1.theta_elec); // temp_buf2 is in Q15
  temp_buf3 = (int)(ilg2_vdc_vcon1.Imeas_a); // temp_buf3 is in Q15
  temp_buf4 = (int)(ilg2_vdc_vcon1.Imeas_b); // temp_buf4 is in Q15
B.1.4. Lazo de velocidad

// Speed Closed Loop

rc1.target_value = _IQ(speed_ref);
rc1.calc(&rc1);

qep1.calc(&qep1);

speed1.theta_elec = _IQ15toIQ((long)qep1.theta_elec);
speed1.dir_QEP = (long)(qep1.dir_QEP);
speed1.calc(&speed1);

if (speed_loop_count==speed_loop_ps) {
    pid1_spd.pid_ref_reg3 = rc1.setpt_value;
    pid1_spd.pid_fdb_reg3 = speed1.speed_frq;
    pid1_spd.calc(&pid1_spd);
    speed_loop_count=1;
}
else speed_loop_count++;

ilg2_vdc_vcon1.read(&ilg2_vdc_vcon1);
ilg2_vdc_vcon1.Imeas_a = ilg2_vdc_vcon1.Imeas_a - ilg2_vdc_vcon1.Imeas_a_offset;
ilg2_vdc_vcon1.Imeas_b = ilg2_vdc_vcon1.Imeas_b - ilg2_vdc_vcon1.Imeas_b_offset;

clarke1.as = _IQ15toIQ((long)ilg2_vdc_vcon1.Imeas_a);
clarke1.bs = _IQ15toIQ((long)ilg2_vdc_vcon1.Imeas_b);
clarke1.calc(&clarke1);

park1.ds = clarke1.ds;
park1.qs = clarke1.qs;
park1.ang = speed1.theta_elec;
park1.calc(&park1);

pid1_iq.pid_ref_reg3 = pid1_spd.pid_out_reg3;
pid1_iq.pid_fdb_reg3 = park1.qe;
pid1_iq.calc(&pid1_iq);

pid1_id.pid_ref_reg3 = _IQ(Id_ref);
pid1_id.pid_fdb_reg3 = park1.de;
pid1_id.calc(&pid1_id);

ipark1.de = pid1_id.pid_out_reg3;
ipark1.qe = pid1_iq.pid_out_reg3;
ipark1.ang = speed1.theta_elec;
ipark1.calc(&ipark1);

svgen_dq1.Ualfa = ipark1.ds;
svgen_dq1.Ubeta = ipark1.qs;
svgen_dq1.calc(&svgen_dq1);

pwm1.Mfunc_c1 = (int)_IQtoIQ15(svgen_dq1.Ta); // Mfunc_c1 is in Q15
pwm1.Mfunc_c2 = (int)_IQtoIQ15(svgen_dq1.Tb); // Mfunc_c2 is in Q15
pwm1.Mfunc_c3 = (int)_IQtoIQ15(svgen_dq1.Tc); // Mfunc_c3 is in Q15
pwm1.update(&pwm1);

// select which of the variables are being displayed through data LOG
  temp_buf1 = (int)_IQtoIQ15(speed1.theta_elec); // temp_buf1 is in Q15
  temp_buf2 = (int)_IQtoIQ15(svgen_dq1.Ta); // temp_buf2 is in Q15
  temp_buf3 = (int)(ilg2_vdc_vcon1.Imeas_a); // temp_buf3 is in Q15
  temp_buf4 = (int)(ilg2_vdc_vcon1.Imeas_b); // temp_buf4 is in Q15
B.2. Documentos pdf de los módulos
**Description**

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

![Diagram](image)

**Availability**

This IQ module is available in one interface format:

1) The C interface version

**Module Properties**

**Type:** Target Independent, Application Independent

**Target Devices:** x281x or x280x

**C Version File Names:** ipark.c, ipark.h

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

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size(^\circ)</td>
<td>45/45 words</td>
<td></td>
</tr>
<tr>
<td>(x281x/x280x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words*</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

\(^\circ\) Each pre-initialized "_iq" IPARK structure consumes 12 words in the data memory

\(^\circ\) Code size mentioned here is the size of the `calc()` function
C Interface

Object Definition

The structure of IPARK object is defined by following structure definition

```c
typedef struct {  
    _iq  Alpha;   // Output: stationary d-axis stator variable 
    _iq  Beta;   // Output: stationary q-axis stator variable  
    _iq  Angle;  // Input: rotating angle (pu) 
    _iq  Ds;  // Input: rotating d-axis stator variable  
    _iq  Qs;  // Input: rotating q-axis stator variable  
    void  (*calc)();  // Pointer to calculation function 
} IPARK;
```

typedef IPARK *IPARK_handle;

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range(Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Ds</td>
<td>Direct axis(D) component of transformed signal in rotating reference frame</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Inputs</td>
<td>Qs</td>
<td>Quadrature axis(Q) component of transformed signal in rotating reference frame</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Inputs</td>
<td>Angle</td>
<td>Phase angle between stationary and rotating frame</td>
<td>GLOBAL_Q</td>
<td>00000000-7FFFFFFF (0 – 360 degree)</td>
</tr>
<tr>
<td>Outputs</td>
<td>Alpha</td>
<td>Direct axis(d) component of the transformed signal</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Outputs</td>
<td>Beta</td>
<td>Quadrature axis(q) component of the transformed signal</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
</tbody>
</table>

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_handle**

User defined Data type of pointer to IPARK module

**IPARK_DEFAULTS**

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

Methods

```c
void ipark_calc(IPARK_handle);
```

This definition implements one method viz., the inverse Park variable transformation computation function. The input argument to this function is the module handle.

Module Usage

**Instantiation**

The following example instances two IPARK objects

```c
IPARK  ipark1, ipark2;
```
Initialization
To instance pre-initialized objects
IPARK ipark1 = IPARK_DEFAULTS;
IPARK ipark2 = IPARK_DEFAULTS;

Invoking the computation function
ipark1.calc(&ipark1);
ipark2.calc(&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

    ipark1.calc(&ipark1);              // Call compute function for ipark1
    ipark2.calc(&ipark2);              // Call compute function 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:

\[
\begin{align*}
Id &= ID \times \cos \theta - IQ \times \sin \theta \\
Iq &= ID \times \sin \theta + IQ \times \cos \theta
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Equation Variables</th>
<th>Program Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>Ds</td>
</tr>
<tr>
<td>IQ</td>
<td>Qs</td>
</tr>
<tr>
<td>(\theta)</td>
<td>Angle</td>
</tr>
<tr>
<td>Outputs</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>Alpha</td>
</tr>
<tr>
<td>iq</td>
<td>Beta</td>
</tr>
</tbody>
</table>

Table 1: Correspondence of notations
**SVGEN_DQ**

*Space Vector Generator With Quadrature Control*

**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 \((\alpha, \beta)\) components, Ualpha and Ubeta.

![Space Vector Generator With Quadrature Control Diagram](image)

**Availability**

This IQ module is available in one interface format:

1) The C interface version

**Module Properties**

**Type:** Target Independent, Application Independent

**Target Devices:** x281x or x280x

**C Version File Names:** svgen_dq.c, svgen_dq.h

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

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size(^{\circ})</td>
<td>256/256 words</td>
<td></td>
</tr>
<tr>
<td>(x281x/x280x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words(^{\ast})</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

\(^{\ast}\) Each pre-initialized ".iq" SVGENDQ structure consumes 12 words in the data memory

\(^{\circ}\) Code size mentioned here is the size of the `calc()` function
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
  void (*calc)(); // Pointer to calculation function
} SVGENDQ;

typedef SVGENDQ *SVGENDQ_handle;

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range(Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Ualpha</td>
<td>Component of reference stator voltage vector on direct axis stationary reference frame.</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Outputs</td>
<td>Ubeta</td>
<td>Component of reference stator voltage vector on quadrature axis stationary reference frame.</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Outputs</td>
<td>Ta</td>
<td>Duty ratio of PWM1 (CMPR1 register value as a fraction of associated period register, TxPR, value).</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Outputs</td>
<td>Tb</td>
<td>Duty ratio of PWM3 (CMPR2 register value as a fraction of associated period register, TxPR, value).</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Outputs</td>
<td>Tc</td>
<td>Duty ratio of PWM5 (CMPR3 register value as a fraction of associated period register, TxPR, value).</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
</tbody>
</table>

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_handle
User defined Data type of pointer to SVGENDQ module

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

    void svgendq_calc(SVGENDQ_handle);

    This definition implements one method viz., the space vector generator computation function. The input argument to this function is the module handle.

Module Usage

Instantiation
The following example instances two SVGENDQ objects
SVGENDQ svgen_dq1, svgen_dq2;

Initialization
To instance pre-initialized objects
SVGENDQ svgen_dq1 = SVGENDQ_DEFAULTS;
SVGENDQ svgen_dq2 = SVGENDQ_DEFAULTS;

Invoking the computation function
svgen_dq1.calc(&svgen_dq1);
svgen_dq2.calc(&svgen_dq2);

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

main()
{
}

void interrupt periodic_interrupt_isr()
{
    svgen_dq1.Ualpha = Ualpha1;             // Pass inputs to svgen_dq1
    svgen_dq1.Ubeta = Ubeta1;             // Pass inputs to svgen_dq1
    svgen_dq2.Ualpha = Ualpha2;             // Pass inputs to svgen_dq2
    svgen_dq2.Ubeta = Ubeta2;             // Pass inputs to svgen_dq2
    svgen_dq1.calc(&svgen_dq1);  // Call compute function for svgen_dq1
    svgen_dq2.calc(&svgen_dq2);  // Call compute function for svgen_dq2
    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 = svgen_dq2.Ta;   // Access the outputs of svgen_dq2
    Tb2 = svgen_dq2.Tb;   // Access the outputs of svgen_dq2
    Tc2 = svgen_dq2.Tc;   // Access the outputs of svgen_dq2
}
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.

![Power circuit topology for a three-phase VSI](image1)

Figure 1 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.

![Power bridge for a three-phase VSI](image2)

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.
Technical Background

Table 1. Device on/off patterns and resulting instantaneous voltages of a 3-phase power inverter

<table>
<thead>
<tr>
<th>c</th>
<th>b</th>
<th>a</th>
<th>V&lt;sub&gt;AN&lt;/sub&gt;</th>
<th>V&lt;sub&gt;BN&lt;/sub&gt;</th>
<th>V&lt;sub&gt;CN&lt;/sub&gt;</th>
<th>V&lt;sub&gt;AB&lt;/sub&gt;</th>
<th>V&lt;sub&gt;BC&lt;/sub&gt;</th>
<th>V&lt;sub&gt;CA&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;</td>
<td>0</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>2V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>-2V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>0</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>2V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>0</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>-2V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-2V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;/3</td>
<td>-V&lt;sub&gt;DC&lt;/sub&gt;</td>
<td>0</td>
<td>V&lt;sub&gt;DC&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The quadrature quantities (in the (α,β) frame) corresponding to these 3 phase voltages are given by the general Clarke transform equation:

\[ V_{s\alpha} = V_{AN} \]
\[ V_{s\beta} = \left(2V_{BN} + V_{AN}\right)/\sqrt{3} \]

In matrix form from the above equation is also expressed as,

\[
\begin{bmatrix}
V_{s\alpha} \\
V_{s\beta}
\end{bmatrix}
= \frac{2}{3}
\begin{bmatrix}
1 & -\frac{1}{2} & -\frac{1}{2} \\
\frac{\sqrt{3}}{2} & \frac{\sqrt{3}}{2} & 0
\end{bmatrix}
\begin{bmatrix}
V_{AN} \\
V_{BN} \\
V_{CN}
\end{bmatrix}
\]

Due to the fact that only 8 combinations are possible for the power switches, \(V_{s\alpha}\) and \(V_{s\beta}\) can also take only a finite number of values in the (α,β) frame according to the status of the transistor command signals (c, b, a). These values of \(V_{s\alpha}\) and \(V_{s\beta}\) for the corresponding instantaneous values of the phase voltages (\(V_{AN}, V_{BN}, V_{CN}\)) are listed in Table 2.
Table 2: Switching patterns, corresponding space vectors and their ($\alpha$, $\beta$) components

<table>
<thead>
<tr>
<th>c</th>
<th>b</th>
<th>a</th>
<th>$V_{sa}$</th>
<th>$V_{sb}$</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$O_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\frac{2}{3}V_{DC}$</td>
<td>0</td>
<td>$U_0$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\frac{-V_{DC}}{3}$</td>
<td>$\frac{V_{DC}}{\sqrt{3}}$</td>
<td>$U_{120}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\frac{V_{DC}}{3}$</td>
<td>$\frac{V_{DC}}{\sqrt{3}}$</td>
<td>$U_{80}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\frac{-V_{DC}}{3}$</td>
<td>$\frac{-V_{DC}}{\sqrt{3}}$</td>
<td>$U_{240}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\frac{V_{DC}}{3}$</td>
<td>$\frac{-V_{DC}}{\sqrt{3}}$</td>
<td>$U_{300}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\frac{-2}{3}V_{DC}$</td>
<td>0</td>
<td>$U_{180}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$O_{111}$</td>
</tr>
</tbody>
</table>

These values of $V_{sa}$ and $V_{sb}$, listed in Table 2, are called the ($\alpha$, $\beta$) 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.
Projection of the stator reference voltage vector $U_{\text{out}}$

The objective of Space Vector PWM technique is to approximate a given stator reference voltage vector $U_{\text{out}}$ by combination of the switching pattern corresponding to the basic space vectors. The reference vector $U_{\text{out}}$ is represented by its $(\alpha, \beta)$ components, $U_{\alpha}$ and $U_{\beta}$. Figure 4 shows the reference voltage vector, its $(\alpha, \beta)$ components and two of the basic space vectors, $U_0$ and $U_{60}$. The figure also indicates the resultant $\alpha$ and $\beta$ components for the space vectors $U_0$ and $U_{60}$. $\Sigma V_{s\beta}$ represents the sum of the $\beta$ components of $U_0$ and $U_{60}$, while $\Sigma V_{s\alpha}$ represents the sum of the $\alpha$ components of $U_0$ and $U_{60}$. Therefore,

$$\begin{align*}
\Sigma V_{s\beta} &= 0 + \frac{V_{DC}}{\sqrt{3}} = \frac{V_{DC}}{\sqrt{3}} \\
\Sigma V_{s\alpha} &= \frac{2V_{DC}}{3} + \frac{V_{DC}}{3} = V_{DC}
\end{align*}$$

For the case in Figure 4, the reference vector $U_{\text{out}}$ is in the sector contained by $U_0$ and $U_{60}$. Therefore $U_{\text{out}}$ is represented by $U_0$ and $U_{60}$. So we can write,

$$\begin{align*}
T &= T_1 + T_3 + T_0 \\
U_{\text{out}} &= \frac{T_1}{T} U_o + \frac{T_3}{T} U_{60}
\end{align*}$$

where, $T_1$ and $T_3$ are the respective durations in time for which $U_0$ and $U_{60}$ are applied within period $T$. $T_0$ is the time duration for which the null vector is applied. These time durations can be calculated as follows:
\[
\begin{align*}
U_{\beta} &= \frac{T_3}{T} U_{60} \sin(60^\circ) \\
U_{\alpha} &= \frac{T_1}{T} |U_0| + \frac{T_2}{T} |U_{60}| \cos(60^\circ)
\end{align*}
\]

From Table 2 and Figure 4 it is evident that the magnitude of all the space vectors is \(2V_{DC}/3\). When this is normalized by the maximum phase voltage (line to neutral), \(V_{DC}/\sqrt{3}\), the magnitude of the space vectors become \(2/\sqrt{3}\) i.e., the normalized magnitudes are \(|U_0| = |U_{60}| = 2/\sqrt{3}\). Therefore, from the last two equations the time durations are calculated as,

\[T_1 = \frac{T}{2} \left(\sqrt{3} U_{\alpha} - U_{\beta}\right)\]

\[T_3 = TU_{\beta}\]

Where, \(U_{\alpha}\) and \(U_{\beta}\) also represent the normalized \((\alpha,\beta)\) components of \(U_{out}\) with respect to the maximum phase voltage \(V_{DC}/\sqrt{3}\). The rest of the period is spent in applying the null vector \(T_o\). The time durations, as a fraction of the total \(T\), are given by,

\[t_1 = \frac{T_1}{T} = \frac{1}{2} \left(\sqrt{3} U_{\alpha} - U_{\beta}\right)\]

\[t_2 = \frac{T_3}{T} = U_{\beta}\]

In a similar manner, if \(U_{out}\) is in sector contained by \(U_{60}\) and \(U_{120}\), then by knowing \(|U_{60}| = |U_{120}| = 2/\sqrt{3}\) (normalized with respect to \(V_{DC}/\sqrt{3}\)), the time durations can be derived as,

\[t_1 = \frac{T_2}{T} = \frac{1}{2} \left(-\sqrt{3} U_{\alpha} + U_{\beta}\right)\]

\[t_2 = \frac{T_3}{T} = \frac{1}{2} \left(\sqrt{3} U_{\alpha} + U_{\beta}\right)\]

where, \(T_2\) is the duration in time for which \(U_{120}\) is applied within period \(T\)

Now, if we define 3 variables \(X\), \(Y\) and \(Z\) according to the following equations,

\[X = U_{\beta}\]

\[Y = \frac{1}{2} \left(\sqrt{3} U_{\alpha} + U_{\beta}\right)\]

\[Z = \frac{1}{2} \left(-\sqrt{3} U_{\alpha} + U_{\beta}\right)\]

Then for the first example, when \(U_{out}\) is in sector contained by \(U_0\) and \(U_{60}\), \(t_1 = -Z\), \(t_2 = X\).

For the second example, when \(U_{out}\) is in sector contained by \(U_{60}\) and \(U_{120}\), \(t_1 = Z\), \(t_2 = Y\).
In a similar manner, \( t1 \) and \( t2 \) can be calculated for the cases when \( U_{out} \) is in sectors contained by other space vectors. For different sectors, the expressions for \( t1 \) and \( t2 \) in terms of \( X \), \( Y \) and \( Z \) are listed in Table 3.

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{Sector} & U_{60}, U_{60} & U_{60}, U_{120} & U_{120}, U_{180} & U_{180}, U_{240} & U_{240}, U_{300} & U_{300}, U_{0} \\
\hline
\text{t1} & -Z & Z & X & -X & -Y & Y \\
\text{t2} & X & Y & Y & Z & -Z & -X \\
\hline
\end{array}
\]

Table 3: \( t1 \) and \( t2 \) definitions for different sectors in terms of \( X \), \( Y \) and \( Z \) variables

In order to know which of the above variables apply, the knowledge of the sector containing the reference voltage vector is needed. This is achieved by first converting the \((\alpha, \beta)\) components of the reference vector \( U_{out} \) into a balanced three-phase quantities. That is, \( U_{alpha} \) and \( U_{beta} \) are converted to a balanced three-phase quantities \( V_{ref1} \), \( V_{ref2} \), and \( V_{ref3} \) according to the following inverse Clarke transformation:

\[
\begin{align*}
V_{ref1} &= U_{beta} \\
V_{ref2} &= \frac{-U_{beta} + U_{alpha} \times \sqrt{3}}{2} \\
V_{ref3} &= \frac{-U_{beta} - U_{alpha} \times \sqrt{3}}{2}
\end{align*}
\]

Note that, this transformation projects the quadrature or \( \beta \) component, \( U_{beta} \), into \( V_{ref1} \). This means that the voltages \( V_{ref1}, V_{ref2}, \) and \( V_{ref3} \) are all phase advanced by \( 90^\circ \) when compared to the corresponding voltages generated by the conventional inverse Clarke transformation which projects the \( \alpha \) component, \( U_{alpha} \), into phase voltage \( V_{AN} \). The following equations describe the \((\alpha, \beta)\) components and the reference voltages:

\[
\begin{align*}
U_{alpha} &= \sin \omega t \\
U_{beta} &= \cos \omega t \\
V_{ref1} &= \cos \omega t \\
V_{ref2} &= \cos(\omega t - 120^\circ) \\
V_{ref3} &= \cos(\omega t + 120^\circ)
\end{align*}
\]

Note that, the above voltages are all normalized by the maximum phase voltage \( (V_{DC}/\sqrt{3}) \).
From the last three equations the following decisions can be made on the sector information:

If \( V_{ref 1} > 0 \) then \( a=1 \), else \( a=0 \)
If \( V_{ref 2} > 0 \) then \( b=1 \), else \( b=0 \)
If \( V_{ref 3} > 0 \) then \( c=1 \), else \( c=0 \)

The variable sector in the code is defined as, \( \text{sector} = 4c + 2b + a \)

For example, in Figure 3 \( a=1 \) for the vectors \( U_{300}, U_{0} \) and \( U_{60} \). For these vectors the phase of \( V_{ref 1} \) are \( \omega t=300^\circ \), \( \omega t=0 \) and \( \omega t=60^\circ \) respectively. Therefore, \( V_{ref 1} > 0 \) when \( a=1 \).

The \((\alpha, \beta)\) components, \( U_{\alpha} \) and \( U_{\beta} \), defined above represent the output phase voltages \( V_{AN}, V_{BN} \) and \( V_{CN} \). The following equations describe these phase voltages:

\[
\begin{align*}
V_{AN} &= \sin \omega t \\
V_{BN} &= \sin(\omega t + 120^\circ) \\
V_{CN} &= \sin(\omega t - 120^\circ)
\end{align*}
\]

The Space Vector PWM module is divided in several parts:
- Determination of the sector
- Calculation of \( X, Y \) and \( Z \)
- Calculation of \( t_1 \) and \( t_2 \)
- Determination of the duty cycle \( t_{aon}, t_{bon} \) and \( t_{con} \)
- Assignment of the duty cycles to \( T_a, T_b \) and \( T_c \)

The variables \( t_{aon}, t_{bon} \) and \( t_{con} \) are calculated using the following equations:
Then the right duty cycle \((txon)\) is assigned to the right motor phase (in other words, to \(Ta, Tb\) and \(Tc\)) according to the sector. Table 4 depicts this determination.

<table>
<thead>
<tr>
<th>sectors</th>
<th>(U_0, U_{60})</th>
<th>(U_{60}, U_{120})</th>
<th>(U_{120}, U_{180})</th>
<th>(U_{180}, U_{240})</th>
<th>(U_{240}, U_{300})</th>
<th>(U_{300}, U_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ta</td>
<td>taon</td>
<td>tbon</td>
<td>tcon</td>
<td>tcon</td>
<td>tbon</td>
<td>taon</td>
</tr>
<tr>
<td>Tb</td>
<td>tbon</td>
<td>taon</td>
<td>taon</td>
<td>tbon</td>
<td>tcon</td>
<td>tcon</td>
</tr>
<tr>
<td>Tc</td>
<td>tcon</td>
<td>tcon</td>
<td>tbon</td>
<td>taon</td>
<td>taon</td>
<td>tbon</td>
</tr>
</tbody>
</table>

Table 4: Table Assigning the Right Duty Cycle to the Right Motor Phase

**Example:** Sector contained by \(U_0\) and \(U_{60}\).

Next, Table 5 shows the correspondence of notations between variables used here and variables used in the program (i.e., svgen_dq.c, svgen_dq.h). The software module requires that both input and output variables are in per unit values.
### Table 5: Correspondence of notations

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Equation Variables</th>
<th>Program Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$U_{\alpha}$</td>
<td>$U_{\alpha}$</td>
</tr>
<tr>
<td></td>
<td>$U_{\beta}$</td>
<td>$U_{\beta}$</td>
</tr>
<tr>
<td>Outputs</td>
<td>$T_a$</td>
<td>$T_a$</td>
</tr>
<tr>
<td></td>
<td>$T_b$</td>
<td>$T_b$</td>
</tr>
<tr>
<td></td>
<td>$T_c$</td>
<td>$T_c$</td>
</tr>
<tr>
<td>Others</td>
<td>$V_{ref1}$</td>
<td>$V_a$</td>
</tr>
<tr>
<td></td>
<td>$V_{ref2}$</td>
<td>$V_b$</td>
</tr>
<tr>
<td></td>
<td>$V_{ref3}$</td>
<td>$V_c$</td>
</tr>
</tbody>
</table>
Description
This module uses the duty ratio information and calculates the compare values for generating PWM outputs. The compare values are used in the full compare unit in 281x event manager (EVA) or EPWM unit in 280x. This also allows PWM period modulation.

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

1) The C interface version

Module Properties
Type: Target Dependent, Application Independent
Target Devices: x281x or x280x

C Version File Names: f281xpwm.c, f281xpwm.h (for x281x)
                         f280xpwm.c, f280xpwm.h (for x280x)

IQmath library files for C: N/A

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size</td>
<td>90/257 words</td>
<td></td>
</tr>
<tr>
<td>(x281x/x280x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words*</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

* Each pre-initialized PWMGEN structure consumes 9 words in the data memory

\(^{\circledast}\) Code size mentioned here is the size of the \textit{init()} and \textit{update()} functions
**C Interface**

**Object Definition**

The structure of PWMGEN object is defined by following structure definition:

```c
typedef struct {   Uint16 PeriodMax;       // Parameter: PWM Half-Period in CPU clock cycles (Q0)
                   int16 MfuncPeriod;      // Input: Period scaler (Q15)
                   int16 MfuncC1;            // Input: PWM 1&2 Duty cycle ratio (Q15)
                   int16 MfuncC2;            // Input: PWM 3&4 Duty cycle ratio (Q15)
                   int16 MfuncC3;            // Input: PWM 5&6 Duty cycle ratio (Q15)
                   void (*init)();                // Pointer to the init function
                   void (*update)();          // Pointer to the update function
} PWMGEN;
```

```c
typedef PWMGEN *PWMGEN_handle;
```

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>MfuncC (x=1,2,3)</td>
<td>PWM duty cycle ratio</td>
<td>Q15</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>MfuncPeriod</td>
<td>Period scaler</td>
<td>Q15</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td>Outputs</td>
<td>PWmx (x=1,2,3,4,5,6)</td>
<td>Output signals from the 6 PWM pins in EVA on the x2812eZdsp.</td>
<td>N/A</td>
<td>0-3.3 V</td>
</tr>
<tr>
<td>PWMGEN parameter</td>
<td>PeriodMax</td>
<td>PWM Half-Period in CPU clock cycles</td>
<td>Q0</td>
<td>8000-7FFF</td>
</tr>
</tbody>
</table>

**Special Constants and Data types**

**PWMGEN**

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

**PWMGEN_handle**

User defined Data type of pointer to PWMGEN module

**PWMGEN_DEFAULTS**

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

**Methods**

```c
void F281X_EV1_PWM_Init(PWMGEN *);
void F281X_EV1_PWM_Update(PWMGEN *);
void F280X_PWM_Init(PWMGEN *);
void F280X_PWM_Update(PWMGEN *);
```
This default definition of the object implements two methods – the initialization and the runtime compute function for PWMGEN generation. This is implemented by means of a function pointer, and the initializer sets this to F281X_EV1_PWM_Init and F281X_EV1_PWM_Update functions for x281x or F280X_PWM_Init and F280X_PWM_Update functions for x280x. The argument to this function is the address of the PWMGEN object.

Module Usage

Instantiation
The following example instances one PWMGEN object
PWMGEN  pwm1;

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

Invoking the computation function
pwm1.init(&pwm1);
pwm1.update(&pwm1);

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

```c
main()
{
    pwm1.PeriodMax = 3750;   // PWM frequency = 20 kHz, clock = 150 MHz
    pwm1.init(&pwm1);  // Call init function for pwm1
}

void interrupt periodic_interrupt_isr()
{
    pwm1.MfuncC1 = (int)_IQtoIQ15(svgen_dq1.Ta);    // svgen_dq1.Ta is in GLOBAL_Q
    pwm1.MfuncC2 = (int)_IQtoIQ15(svgen_dq1.Tb);    // svgen_dq1.Tb is in GLOBAL_Q
    pwm1.MfuncC3 = (int)_IQtoIQ15(svgen_dq1.Tc);    // svgen_dq1.Tc is in GLOBAL_Q
    pwm1.update(&pwm1);         // Call update function for pwm1
}
```

This module allows 3-channel analog-to-digital conversion with programmable gains and offsets. The conversions are triggered on GP Timer 1 underflow (for 281x) or EPWM1 CNT_zero event (for 280x). The converted results represent load currents and DC-bus voltage in the inverter.

1. GP Timer 1 is the time base for symmetrical Pulse-Width Modulation (for 281x) or Time Base of EPWM1 CNT_zero event (for 280x);
2. For line current measurement, the analog inputs are the voltage across resistors placed between the sources or emitters of low-side power devices and low-side DC rail; and
3. For DC-bus voltage measurement, the analog input is the voltage across resistor by means of voltage divider concept.

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

1) The C interface version

Module Properties
Type: Target Dependent, Application Independent
Target Devices: x281x or x280x
C Version File Names: f281xileg_vdc.c, f281xileg_vdc.h (for x281x)
                    f280xileg_vdc.c, f280xileg_vdc.h (for x280x)
IQmath library files for C: N/A

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size(^1)</td>
<td>147/155 words</td>
<td></td>
</tr>
<tr>
<td>(x281x/x280x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words*</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) Each pre-initialized ILEG2DCBUSMEAS structure consumes 15 words in the data memory

\(^2\) Code size mentioned here is the size of the *init()* and *read()* functions
C Interface

Object Definition
The structure of ILEG2DCBUSMEAS object is defined by following structure definition

```c
typedef struct { int16 ImeasAGain; // Parameter: gain for Ia (Q13) int16 ImeasAOffset; // Parameter: offset for Ia (Q15) int16 ImeasA; // Output: measured Ia (Q15) int16 ImeasBGain; // Parameter: gain for Ib (Q13) int16 ImeasBOffset; // Parameter: offset for Ib (Q15) int16 ImeasB; // Output: measured Ib (Q15) int16 VdcMeasGain; // Parameter: gain for Vdc (Q13) int16 VdcMeasOffset; // Parameter: offset for Vdc (Q15) int16 VdcMeas; // Output: measured Vdc (Q15) int16 ImeasC; // Output: computed Ic (Q15) Uint16 ChSelect; // Parameter: ADC channel selection void (*init)(); // Pointer to the init function void (*read)(); // Pointer to the read function } ILEG2DCBUSMEAS;
```

typedef ILEG2DCBUSMEAS *ILEG2DCBUSMEAS_handle;

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range(Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>ADCINx, ADCINy, ADCINz</td>
<td>ADC pins in 281x device where x,y,z correspond to the channel numbers selected by ChSelect</td>
<td>N/A</td>
<td>0-3 V</td>
</tr>
<tr>
<td>Outputs</td>
<td>ImeasA</td>
<td>x^th channel digital representation for current Ia</td>
<td>Q15</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>ImeasB</td>
<td>y^th channel digital representation for current Ib</td>
<td>Q15</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>ImeasC</td>
<td>Computing current Ic</td>
<td>Q15</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>VdcMeas</td>
<td>z^th channel digital representation for DC-bus voltage Vdc</td>
<td>Q15</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td>ILEG2DCBUSMEAS</td>
<td>ChSelect</td>
<td>16-bit ADC channel select format can be seen as: ChSelect = 0xyz</td>
<td>Q0</td>
<td>x,y,z are in between 0h -&gt; Fh</td>
</tr>
<tr>
<td>parameter</td>
<td>ImeasAGain</td>
<td>Gain for x^th channel. Modify this if default gain is not used.</td>
<td>Q13</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>ImeasBGain</td>
<td>Gain for y^th channel. Modify this if default gain is not used.</td>
<td>Q13</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>VdcMeasGain</td>
<td>Gain for z^th channel. Modify this if default gain is not used.</td>
<td>Q13</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>ImeasAOffset</td>
<td>Offset for x^th channel. Modify this if default offset is not used.</td>
<td>Q15</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>ImeasBOffset</td>
<td>Offset for y^th channel. Modify this if default offset is not used.</td>
<td>Q15</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>VdcMeasOffset</td>
<td>Offset for z^th channel. Modify this if default offset is not used.</td>
<td>Q15</td>
<td>8000-7FFF</td>
</tr>
</tbody>
</table>

Special Constants and Data types

ILEG2DCBUSMEAS
The module definition is created as a data type. This makes it convenient to instance an interface to the ILEG2DCBUSMEAS driver. To create multiple instances of the module simply declare variables of type ILEG2DCBUSMEAS.
**ILEG2DCBUSMEAS_handle**  
User defined Data type of pointer to ILEG2DCBUSMEAS module

**ILEG2DCBUSMEAS_DEFAULTS**  
Structure symbolic constant to initialize ILEG2DCBUSMEAS module. This provides the initial values to the terminal variables as well as method pointers.

**Methods**

```c
void F281X_ileg2_dcbus_drv_init(ILEG2DCBUSMEAS *);
void F281X_ileg2_dcbus_drv_read(ILEG2DCBUSMEAS *);
void F280X_ileg2_dcbus_drv_init(ILEG2DCBUSMEAS *);
void F280X_ileg2_dcbus_drv_read(ILEG2DCBUSMEAS *);
```

This default definition of the object implements two methods – the initialization and the runtime read function for ILEG2DCBUSMEAS measurement. This is implemented by means of a function pointer, and the initializer sets this to F281X_ileg2_dcbus_drv_init and F281X_ileg2_dcbus_drv_read functions for x281x or F280X_ileg2_dcbus_drv_init and F280X_ileg2_dcbus_drv_read functions for x280x. The argument to this function is the address of the ILEG2DCBUSMEAS object.

**Module Usage**

**Instantiation**  
The following example instanciates one ILEG2DCBUSMEAS object  
ILEG2DCBUSMEAS ilg2_vdc1;

**Initialization**  
To Instance pre-initialized objects  
ILEG2DCBUSMEAS ilg2_vdc1 = ILEG2DCBUSMEAS_DEFAULTS;

**Invoking the computation function**  
ilg2_vdc1.init(&ilg2_vdc1);  
ilg2_vdc1.read(&ilg2_vdc1);

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

```c
main()
{
    ilg2_vdc1.init(&ilg2_vdc1); // Call init function for ilg2_vdc1
}
```
void interrupt periodic_interrupt_isr()
{
    ilg2_vdc1.read(&ilg2_vdc1); // Call read function for ilg2_vdc1
    Ias = _IQ15toIQ((int32)ilg2_vdc1.ImeasA); // Ias is in GLOBAL_Q
    Ibs = _IQ15toIQ((int32)ilg2_vdc1.ImeasB); // Ibs is in GLOBAL_Q
    DC_bus = _IQ15toIQ((int32)ilg2_vdc1.VdcMeas); // DC_bus GLOBAL_Q
}

Technical Background

The ADCIN pins accepts the analog input signals (I_a, I_b, and V_{dc}) in the range of 0-3 volts for x28xx based DSP with ground referenced to 0 volt (VREFLO = 0).

Consequently, before connecting these signals to ADCIN pins, the hardware adjustment by external op-amp circuits (for gain and offset adjustments) for these analog signals such that they represent according to the selected base quantities and the appropriate voltage range is required.

Four output variables of the module (I_{measA}, I_{measB}, I_{measC}, and V_{dcMeas}) are computed after three ADC analog input signals are digitized as seen below:

\[
\begin{align*}
I_{measA} &= I_{measAGain} \times \text{ADC}_{-}I_a_{-}Q15 + I_{measAOffset} \\
I_{measB} &= I_{measBGain} \times \text{ADC}_{-}I_b_{-}Q15 + I_{measBOffset} \\
I_{measC} &= -(I_{measA} + I_{measB}) \\
V_{dcMeas} &= V_{dcMeasGain} \times \text{ADC}_{-}V_{dc_{-}}Q15 + V_{dcMeasOffset}
\end{align*}
\]

Note that ADC_{-}I_x_{-}Q15 (x=a,b) and ADC_{-}Vdc_{-}Q15 are already converted to Q15 number.

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

The input AC currents (I_a, I_b) are typically sensed and re-scaled within the range of 0-3 volts for x28xx based DSP with the appropriate base current. Thus, the Q15-number conversion is necessary for the current measurements after they are digitized as seen in Figure 1.

It is emphasized that the ADC unit is 12-bit resolution with left justified in the 16-bit ADC result register. Thus, the ADC output range is in between 0000h and FFF0h.

![Figure 1: Q15-number conversion for current measurements (bipolar signal)](image_url)

For DC-bus voltage (V_{dc}), practically the input signal is already represented in the positive range only, so its digitized variable has to be rescaled corresponding to the Q15 number.
For better understanding, Figure 2 illustrates the Q15-number conversion for the DC-bus voltage measurement.

Figure 2: Q15-number conversion for DC-bus voltage measurement (unipolar signal)

In both cases of Q15-number conversion, the number is distorted a little bit about the maximum value (e.g., 7FF0h for bipolar and 7FF8h for unipolar at the maximum value of 7FFFh).
**CLARKE**

---

**Clarke Variable Transformation**

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

![Diagram of CLARKE transformation](image)

**Availability**
This IQ module is available in one interface format:
1) The C interface version

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

**Target Devices:** x281x or x280x

**C Version File Names:** clarke.c, clarke.h

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

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size</td>
<td>23/23 words (x281x/x280x)</td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words*</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

* Each pre-initialized “_iq” CLARKE structure consumes 10 words in the data memory

* Code size mentioned here is the size of the **calc()** function
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  Alpha;  // Output: stationary d-axis stator variable
    _iq  Beta;  // Output: stationary q-axis stator variable
    void  (*calc)();  // Pointer to calculation function
} CLARKE;

typedef CLARKE *CLARKE_handle;

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range(Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>As</td>
<td>Phase ‘a’ component of the balanced three phase quantities</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Bs</td>
<td>Phase ‘b’ component of the balanced three phase quantities</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Outputs</td>
<td>Alpha</td>
<td>Direct axis(d) component of the transformed signal</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Beta</td>
<td>Quadrature axis(q) component of the transformed signal</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
</tbody>
</table>

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_handle
User defined Data type of pointer to CLARKE module

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

Methods

void clarke_calc(CLARKE_handle);

This definition implements one method viz., the Clarke variable transformation computation function. The input argument to this function is the module handle.

Module Usage

Instantiation
The following example instances two CLARKE objects
CLARKE clarke1, clarke2;
\textbf{Initialization}

To instance pre-initialized objects

\begin{verbatim}
CLARKE clarke1 = CLARKE_DEFAULTS;
CLARKE clarke2 = CLARKE_DEFAULTS;
\end{verbatim}

\textbf{Invoking the computation function}

\begin{verbatim}
clarke1.calc(&clarke1);
clarke2.calc(&clarke2);
\end{verbatim}

\textbf{Example}

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

\begin{verbatim}
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

    clarke1.calc(&clarke1);  // Call compute function for clarke1
    clarke2.calc(&clarke2);  // Call compute function 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
}
\end{verbatim}
Technical Background

Implement the following equations:

\[
\begin{aligned}
Id &= Ia \\
Iq &= (2Ib + Ia) / \sqrt{3}
\end{aligned}
\]

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

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

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

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

<table>
<thead>
<tr>
<th>Equation Variables</th>
<th>Program Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Outputs</td>
</tr>
<tr>
<td>ia</td>
<td>As</td>
</tr>
<tr>
<td>ib</td>
<td>Bs</td>
</tr>
<tr>
<td>id</td>
<td>Alpha</td>
</tr>
<tr>
<td>iq</td>
<td>Beta</td>
</tr>
</tbody>
</table>

Table 1: Correspondence of notations
**PARK**

**Park Variable Transformation**

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

![Diagram of PARK transformation]

**Availability**
This IQ module is available in one interface format:

1) The C interface version

**Module Properties**

**Type:** Target Independent, Application Independent

**Target Devices:** x281x or x280x

**C Version File Names:** park.c, park.h

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

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size(^{\circ})</td>
<td>45/45 words</td>
<td>(x281x/x280x)</td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words*</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

* Each pre-initialized “_iq” PARK structure consumes 12 words in the data memory

\(^{\circ}\) Code size mentioned here is the size of the `calc()` function
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 Angle;     // Input: rotating angle (pu)
    _iq Ds;       // Output: rotating d-axis stator variable
    _iq Qs;       // Output: rotating q-axis stator variable
    void (*calc)();  // Pointer to calculation function
} PARK;

typedef PARK *PARK_handle;

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Alpha</td>
<td>Direct axis(d) component of the transformed signal</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Beta</td>
<td>Quadrature axis(q) component of the transformed signal</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Angle</td>
<td>Phase angle between stationary and rotating frame</td>
<td>GLOBAL_Q</td>
<td>00000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Ds</td>
<td>Direct axis(D) component of transformed signal in rotating reference frame</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Qs</td>
<td>Quadrature axis(Q) component of transformed signal in rotating reference frame</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
</tbody>
</table>

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_handle
User defined Data type of pointer to PARK module

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

Methods

void park_calc(PARK_handle);

This definition implements one method viz., the Park variable transformation computation function. The input argument to this function is the module handle.

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 function**
park1.calc(&park1);
park2.calc(&park2);

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

```c
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

    park1.calc(&park1);  // Call compute function for park1
    park2.calc(&park2);  // Call compute function 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:

\[
\begin{aligned}
    ID &= Id \times \cos \theta + Iq \times \sin \theta \\
    IQ &= -Id \times \sin \theta + Iq \times \cos \theta
\end{aligned}
\]

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:

\[
\begin{aligned}
    id &= I \times \sin(\omega t) \\
    iq &= I \times \sin(\omega t + \pi / 2)
\end{aligned}
\]

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

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Equation Variables</th>
<th>Program Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Alpha</td>
<td></td>
</tr>
<tr>
<td>iq</td>
<td>Beta</td>
<td></td>
</tr>
<tr>
<td>(\theta)</td>
<td>Angle</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>Equation Variables</th>
<th>Program Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Ds</td>
<td></td>
</tr>
<tr>
<td>IQ</td>
<td>Qs</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Correspondence of notations
**RAMP_GEN**

**Ramp Generator**

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

![Diagram of RAMP_GEN module]

**Availability**
This IQ module is available in one interface format:

1) The C interface version

**Module Properties**

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size ($)</td>
<td>57/57 words</td>
<td></td>
</tr>
<tr>
<td>(x281x/x280x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words*</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

* Each pre-initialized "_iq" RAMPGEN structure consumes 14 words in the data memory

($) Code size mentioned here is the size of the `calc()` function
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
  void (*calc)(); // Pointer to calculation function
} RAMPGEN;

typedef RAMPGEN *RAMPGEN_handle;

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range(Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Freq</td>
<td>Ramp frequency</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Gain</td>
<td>Ramp gain</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Offset</td>
<td>Ramp offset</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Outputs</td>
<td>Out</td>
<td>Ramp signal</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>StepAngleMax</td>
<td>sv_freq_max = fb*T</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
</tbody>
</table>

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_handle
User defined Data type of pointer to RAMPGEN module

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

Methods

void rampgen_calc(RAMPGEN_handle);

This definition implements one method viz., the ramp generator computation function. The input argument to this function is the module handle.
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 function**
rg1.calc(&rg1);
rg2.calc(&rg2);

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

```c
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

    rg1.calc(&rg1);    // Call compute function for rg1
    rg2.calc(&rg2);    // Call compute function 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.

\[
\text{Angle} = \text{Angle} + \text{StepAngleMax} \times \text{Freq}
\]

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:

\[
f = \frac{\text{StepAngle} \times f_s}{2^m}
\]

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 \(f_s=20\text{kHz}\). This defines the frequency setting resolution of the IQmath implementation, the maximum step size in per-unit, StepAngleMax, for a given base frequency, \(f_b\) and a defined GLOBAL_Q number is therefore computed as follows:

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

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

\[
\text{StepAngleMax} = \_\text{IQ}(f_b \times T_s)
\]

where Ts is the sampling period (sec).
**RMP_CNTL**

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

![Diagram of RMP_CNTL](attachment://diagram.png)

**Availability**
This IQ module is available in one interface format:

1) The C interface version

**Module Properties**

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size(^\circ)</td>
<td>55/55 words</td>
<td></td>
</tr>
<tr>
<td>(x281x/x280x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words(^*)</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

\(^*\) Each pre-initialized "_iq" RMP_CNTL structure consumes 16 words in the data memory

\(^\circ\) Code size mentioned here is the size of the `calc()` function
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)
  void (*calc)();     // Pointer to calculation function
} RMPCNTL;

typedef RMPCNTL *RMPCNTL_handle;

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range(Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>TargetValue</td>
<td>Target input</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>SetpointValue</td>
<td>Target output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Outputs</td>
<td>EqualFlag</td>
<td>Flag output</td>
<td>Q0</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>RampDelayMax</td>
<td>Maximum delay rate</td>
<td>Q0</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>parameter</td>
<td>RampLowLimit</td>
<td>Minimum limit</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>RampHighLimit</td>
<td>Maximum limit</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Internal</td>
<td>RampDelayCount</td>
<td>Incremental delay</td>
<td>Q0</td>
<td>80000000-7FFFFFFF</td>
</tr>
</tbody>
</table>

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_handle

User defined Data type of pointer to RMPCNTL module

RMPCNTL_DEFAULTS

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

Methods

void rmp_cntl_calc(RMPCNTL_handle);

This definition implements one method viz., the ramp control computation function. The input argument to this function is the module handle.
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 function**
rc1.calc(&rc1);
rc2.calc(&rc2);

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

```c
main()
{
}
void interrupt periodic_interrupt_isr()
{
    rc1.TargetValue = target1;              // Pass inputs to rc1
    rc2.TargetValue = target2;              // Pass inputs to rc2
    rc1.calc(&rc1);     // Call compute function for rc1
    rc2.calc(&rc2);     // Call compute function 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 $\text{TargetValue} > \text{SetpointValue}$

\[
\text{SetpointValue} = \text{SetpointValue} + \_IQ(0.0000305), \text{ for } t = n, \text{ Td, } n = 1, 2, 3, \ldots
\]

and $(\text{SetpointValue} + \_IQ(0.0000305)) < \text{RampHighLimit}$

$= \text{RampHighLimit}, \text{ for } (\text{SetpointValue} + \_IQ(0.0000305)) > \text{RampHighLimit}$

where, $\text{Td} = \text{RampDelayMax} \times \text{Ts}$

$\text{Ts} = \text{Sampling time period}$

**Case 2:** When $\text{TargetValue} < \text{SetpointValue}$

\[
\text{SetpointValue} = \text{SetpointValue} - \_IQ(0.0000305), \text{ for } t = n, \text{ Td, } n = 1, 2, 3, \ldots
\]

and $(\text{SetpointValue} - \_IQ(0.0000305)) > \text{RampLowLimit}$

$= \text{RampLowLimit}, \text{ for } (\text{SetpointValue} - \_IQ(0.0000305)) < \text{RampLowLimit}$

where, $\text{Td} = \text{RampDelayMax} \times \text{Ts}$

$\text{Ts} = \text{Sampling time period}$

**Note that** $\text{TargetValue}$ and $\text{SetpointValue}$ variables are in \_iq format.

---

Example:

$\text{SetpointValue} = 0$ (initial value), $\text{TargetValue} = 1000$ (user specified),

$\text{RampDelayMax} = 500$ (user specified), sampling loop time period $\text{Ts} = 0.000025$ Sec.

This means that the time delay for each ramp step is $\text{Td} = 500 \times 0.000025 = 0.0125$ Sec.

Therefore, the total ramp time will be $\text{Tramp} = 1000 \times 0.0125$ Sec = 12.5 Sec
**Description**
This module implements a 32-bit digital PID controller with anti-windup correction. It can be used for PI or PD controller as well. In this digital PID controller, the differential equation is transformed to the difference equation by means of the backward approximation.

**Availability**
This IQ module is available in one interface format:
1) The C interface version

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

*Target Devices*: x281x or x280x

*C Version File Names*: pid_reg3.c, pid_reg3.h

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

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size (x281x/x280x)</td>
<td>85/85 words</td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words*</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

* Each pre-initialized ".iq" PID_REG3 structure consumes 34 words in the data memory

* Code size mentioned here is the size of the calc() function
C Interface

Object Definition

The structure of PID_REG3 object is defined by following structure definition

typedef struct {
  _iq Ref;  // Input: Reference input
  _iq Fdb;  // Input: Feedback input
  _iq Err;  // Variable: Error
  _iq Kp;   // Parameter: Proportional gain
  _iq Up;   // Variable: Proportional output
  _iq Ui;   // Variable: Integral output
  _iq Ud;   // Variable: Derivative output
  _iq OutPreSat;  // Variable: Pre-saturated output
  _iq OutMax;   // Parameter: Maximum output
  _iq OutMin;   // Parameter: Minimum output
  _iq Out;  // Output: PID output
  _iq SatErr; // Variable: Saturated difference
  _iq Ki;   // Parameter: Integral gain
  _iq Kc;  // Parameter: Integral correction gain
  _iq Kd;  // Parameter: Derivative gain
  _iq Up1; // History: Previous proportional output
  void (*calc)();  // Pointer to calculation function
} PIDREG3;

typedef PIDREG3 *PIDREG3_handle;

Module Terminal Variables/Functions

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range(Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Ref</td>
<td>Reference input</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Fdb</td>
<td>Feedback input</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>OutMax</td>
<td>Maximum PID32 module output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>OutMin</td>
<td>Minimum PID32 module output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Output</td>
<td>Out</td>
<td>PID Output (Saturated)</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>PID parameter</td>
<td>Kp</td>
<td>Proportional gain</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Ki</td>
<td>Integral gain</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Kd</td>
<td>Derivative gain</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Kc</td>
<td>Integral correction gain</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td>Internal</td>
<td>Err</td>
<td>Error=Reference-feedback</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>SatErr</td>
<td>SatErr=output-preSatOut</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Up</td>
<td>Proportional output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Up1</td>
<td>Previous proportional output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Ui</td>
<td>Integral output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Ud</td>
<td>Differential output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>OutPreSat</td>
<td>PID output before saturation</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFF</td>
</tr>
</tbody>
</table>

GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.
Special Constants and Data types

**PIDREG3**
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 PIDREG3.

**PIDREG3_handle**
User defined Data type of pointer to PID_REG3 module

**PIDREG3_DEFAULTS**
Structure symbolic constant to initialize PID_REG3 module. This provides the initial values to the terminal variables as well as method pointers.

**Methods**

```c
void pid_reg3_calc(PIDREG3_handle);
```

This function implements the digital PID controller (IQ implementation) using backward approximation technique. The input argument to this function is the module handle.

**Module Usage**

**Instantiation**
The following example instances two PID objects
```c
PIDREG3 pid1, pid2;
```

**Initialization**
To instance pre-initialized objects
```c
PIDREG3 pid1 = PIDREG3_DEFAULTS;
PIDREG3 pid2 = PIDREG3_DEFAULTS;
```

**Invoking the computation function**
```c
pid1.calc(&pid1);
pid2.calc(&pid2);
```

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

```c
/* Instance the PID_REG3 module */
PIDREG3 pid1=PIDREG3_DEFAULTS;
PIDREG3 pid2=PIDREG3_DEFAULTS;
main()
{
    pid1.Kp = _IQ(0.5);   // Pass _iq parameters to pid1
    pid1.Ki  = _IQ(0.001);   // Pass _iq parameters to pid1
    pid1.Kd = _IQ(0.01);   // Pass _iq parameters to pid1
    pid1.Kc = _IQ(0.9);   // Pass _iq parameters to pid1
```
C Interface

```c
pid2.Kp = _IQ(0.8);   // Pass _iq parameters to pid2
pid2.Ki = _IQ(0.0001);   // Pass _iq parameters to pid2
pid2.Kd = _IQ(0.02);   // Pass _iq parameters to pid2
pid2.Kc = _IQ(0.8);   // Pass _iq parameters to pid2
}

void interrupt periodic_interrupt_isr()
{
    pid1.Ref = input1_1;   // Pass _iq inputs to pid1
    pid1.Fdb = input1_2;   // Pass _iq inputs to pid1
    pid2.Ref = input2_1;   // Pass _iq inputs to pid2
    pid2.Fdb = input2_2;   // Pass _iq inputs to pid2
    pid1.calc(&pid1);  // Call compute function for pid1
    pid2.calc(&pid2);  // Call compute function for pid2
    output1 = pid1.Out;  // Access the output of pid1
    output2 = pid2.Out;  // Access the output of pid2
}
```
Technical Background

The block diagram of a conventional PID controller with anti-windup correction can be shown in Figure 1.

![Block diagram of PID controller with anti-windup](image)

Figure 1: Block diagram of PID controller with anti-windup

The differential equation for PID controller with anti-windup before saturation is described in the following equation [1].

\[ u_{\text{presat}}(t) = u_p(t) + u_i(t) + u_d(t) \]  \hspace{1cm} (1)

Each term can be expressed as follows:

Proportional term:
\[ u_p(t) = K_p e(t) \]  \hspace{1cm} (2)

Integral term with saturation correction:
\[ u_i(t) = \frac{K_p}{T_i} \int_0^t e(\zeta) d\zeta + K_c (u(t) - u_{\text{presat}}(t)) \]  \hspace{1cm} (3)

Derivative term:
\[ u_d(t) = K_p T_d \frac{de(t)}{dt} \]  \hspace{1cm} (4)

where
- \( u(t) \) is the output of PID controller
- \( u_{\text{presat}}(t) \) is the output before saturation
- \( e(t) \) is the error between the reference and feedback variables
- \( K_p \) is the proportional gain of PID controller
- \( T_i \) is the integral time (or reset time) of PID controller
- \( T_d \) is the derivative time of PID controller
- \( K_c \) is the integral correction gain of PID controller

Equations (1)-(4) can be discretized using backward approximation as follows:

Pre-saturated output:
\[ u_{\text{presat}}(k) = u_p(k) + u_i(k) + u_d(k) \]  \hspace{1cm} (5)

Proportional term:
Technical Background

\[ u_p(k) = K_p e(k) \]  \hspace{1cm} (6)

Integral term with saturation correction:

\[ u_i(k) = u_i(k-1) + K_p \frac{T}{T_i} e(k) + K_c \left( u(k) - u_{\text{presat}}(k) \right) \]  \hspace{1cm} (7)

Derivative term:

\[ u_d(k) = K_p \frac{T_d}{T} \left( e(k) - e(k-1) \right) \]  \hspace{1cm} (8)

Defining \( K_i = \frac{T}{T_i} \), and \( K_d = \frac{T_d}{T} \), then integral with saturation correction and derivative terms finally become

\[ u_i(k) = u_i(k-1) + K_i u_p(k) + K_c \left( u(k) - u_{\text{presat}}(k) \right) \]  \hspace{1cm} (9)

\[ u_d(k) = K_d (u_p(k) - u_p(k-1)) \]  \hspace{1cm} (10)

where \( T \) is sampling period (sec).

Table 1 shows the correspondence of notation between variables used here and variables used in the program (i.e., pid_reg3.c and pid_reg3.h). The software module requires that both input and output variables are in per unit values.

<table>
<thead>
<tr>
<th></th>
<th>Equation Variables</th>
<th>Program Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td>Ref</td>
<td>Ref</td>
</tr>
<tr>
<td></td>
<td>Fdb</td>
<td>Fdb</td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
<td>u(k)</td>
<td>Out</td>
</tr>
<tr>
<td></td>
<td>e(k)</td>
<td>Err</td>
</tr>
<tr>
<td></td>
<td>u_p(k)</td>
<td>Up</td>
</tr>
<tr>
<td></td>
<td>u_p(k-1)</td>
<td>Up1</td>
</tr>
<tr>
<td></td>
<td>u_i(k)</td>
<td>Ui</td>
</tr>
<tr>
<td></td>
<td>u_d(k)</td>
<td>Ud</td>
</tr>
<tr>
<td></td>
<td>u_{\text{presat}}(k)</td>
<td>OutPreSat</td>
</tr>
<tr>
<td></td>
<td>e_{\text{sat}}(k)</td>
<td>SatErr</td>
</tr>
<tr>
<td></td>
<td>K_p</td>
<td>Kp</td>
</tr>
<tr>
<td></td>
<td>K_i</td>
<td>Ki</td>
</tr>
<tr>
<td></td>
<td>K_d</td>
<td>Kd</td>
</tr>
<tr>
<td></td>
<td>K_c</td>
<td>Kc</td>
</tr>
</tbody>
</table>

Table 1: Correspondence of notations

References:

**QEP_THETA_DRV Quadrature Encoder Pulse Driver**

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

**Availability**
This 16-bit module is available in one interface format:
1) The C interface version

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

**Target Devices:** x281x or x280x

**C Version File Names:** f281xqep.c, f281xqep.h (for x281x)
f280xqep.c, f280xqep.h (for x280x)

**IQmath library files for C:** N/A

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size (x281x/x280x)</td>
<td>88/150 words</td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words*</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

* Each pre-initialized QEP structure consumes 17 words in the data memory (for x281x) and 21 words in the data memory (for x280x)

* Code size mentioned here is the size of the **init()**, **calc()**, and **isr()** functions (for x281x) and the size of the **init()** and **calc()** functions (for x280x)
C Interface

Object Definition

The structure of QEP object is defined by following structure definition:

```c
typedef struct {
    int16 ElecTheta;               // Output: Motor Electrical angle (Q15)
    int16 MechTheta;              // Output: Motor Mechanical Angle (Q15)
    Uint16 DirectionQep;        // Output: Motor rotation direction (Q0)
    Uint16 QepCountIndex;    // Variable: Encoder counter index (Q0)
    Uint16 RawTheta;               // Variable: Raw angle from Timer 2 (Q0)
    Uint16 MechScaler;          // Parameter: 0.9999/total count (Q30)
    Uint16 LineEncoder;        // Parameter: Number of line encoder (Q0)
    Uint16 PolePairs;             // Parameter: Number of pole pairs (Q0)
    int16 CalibratedAngle;     // Parameter: Raw offset between encoder and ph-a (Q0)
    Uint16 IndexSyncFlag;    // Output: Index sync status (Q0)
    void (*init)();                     // Pointer to the init function
    void (*calc)();                   // Pointer to the calc function
    void (*isr)();                      // Pointer to the isr function
}  QEP;
```

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

typedef QEP *QEP_handle;

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td>QEP_A</td>
<td>QEP_A signal applied to CAP1</td>
<td>N/A</td>
<td>0-3.3 v</td>
</tr>
<tr>
<td></td>
<td>QEP_B</td>
<td>QEP_A signal applied to CAP2</td>
<td>N/A</td>
<td>0-3.3 v</td>
</tr>
<tr>
<td></td>
<td>QEP_Index</td>
<td>QEP_Index signal applied to CAP3</td>
<td>N/A</td>
<td>0-3.3 v</td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
<td>ElecTheta</td>
<td>Motor Electrical angle</td>
<td>Q15 (281x) Q24 (280x)</td>
<td>0000-7FFF 00000000-7FFFFFFF (0 – 360 degree)</td>
</tr>
<tr>
<td></td>
<td>MechTheta</td>
<td>Motor Mechanical Angle</td>
<td>Q15 (281x) Q24 (280x)</td>
<td>0000-7FFF 00000000-7FFFFFFF (0 – 360 degree)</td>
</tr>
<tr>
<td></td>
<td>DirectionQep</td>
<td>Motor rotation direction</td>
<td>Q0</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>IndexSyncFlag</td>
<td>Index sync status</td>
<td>Q0</td>
<td>0 or F0</td>
</tr>
<tr>
<td><strong>QEP parameter</strong></td>
<td>MechScaler*</td>
<td>MechScaler = 1/total count, total count = 4*no_lines_encoder</td>
<td>Q30</td>
<td>00000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>PolePairs</td>
<td>Number of pole pairs</td>
<td>Q0</td>
<td>1, 2, 3, ...</td>
</tr>
<tr>
<td></td>
<td>CalibratedAngle</td>
<td>Raw offset between encoder and phase a</td>
<td>Q0</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td><strong>Internal</strong></td>
<td>QepCountIndex</td>
<td>Encoder counter index</td>
<td>Q0</td>
<td>8000-7FFF</td>
</tr>
<tr>
<td></td>
<td>RawTheta</td>
<td>Raw angle from Timer 2</td>
<td>Q0</td>
<td>8000-7FFF</td>
</tr>
</tbody>
</table>

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

**Special Constants and Data types**

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

**QEP_handle**
User defined Data type of pointer to QEP module

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

**Methods**

```c
void F281X_EV1_QEP_Init(QEP *);
void F281X_EV1_QEP_Calc(QEP *);
void F281X_EV1_QEP_Isr(QEP *);

void F280X_QEP_Init(QEP *);
void F280X_QEP_Calc(QEP *);
```

This default definition of the object implements three methods – the initialization, the runtime compute, and interrupt functions for QEP generation. This is implemented by means of a function pointer, and the initializer sets this to F281X_EV1_QEP_Init, F281X_EV1_QEP_Calc, and F281X_EV1_QEP_Isr functions for x281x or F280X_QEP_Init, and F280X_QEP_Calc functions for x280x. The argument to this function is the address of the QEP object.
Module Usage

**Instantiation**
The following example instances one QEP object

```
QEP qep1;
```

**Initialization**
To Instance pre-initialized objects

```
QEP qep1 = QEP_DEFAULTS;
```

**Invoking the computation function**

```
qep1.init(&qep1);
qep1.calc(&qep1);
qep1.isr(&qep1);
```

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

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

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

```
main()
{
    qep1.init(&qep1);       // Call init function for qep1
}

void interrupt periodic_interrupt_isr()
{
    qep1.calc(&qep1);       // Call compute function for qep1
}

void interrupt cap3_interrupt_isr()
{
    qep1.isr(&qep1);        // Call isr function for qep1
}
```

Notice that this example is for x281x only. For x280x devices, there is no capture 3 interrupt.
Technical Background

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

Example:
1000 QEP pulses
= 4000 counter “ticks,” per 360°
These signals are applied to x281x CAP/QEP interface circuit to determine the motor speed, position and direction of rotation. QEP_A and QEP_B signals are applied to the QEP1 and QEP2 pins of x281x device, respectively. The QEP_index signal is applied to the CAP3 pin. The QEP interface circuit in 281x, when enabled (CAPCONx[13,14]), count these QEP pulses and generates two signals internal to the device. These two signals are shown in Figure 2 as QEP_CLK and DIR. QEP_CLK signal is used as the clock input to GP Timer2. DIR signal controls the GP Timer 2 counting direction. For the x280x devices, QEP_A and QEP_B signals are applied to the EQEP1A and EQEP1B pins, respectively. The QEP_index signal is applied to the EQEP1I pin. And the position counter is obtained by QPOSCNT register.

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

In the x281x devices, the counting direction of GP Timer2 is reflected by the status bit, BIT14, in GPTCON register. Therefore, in the s/w, BIT14 of GPTCON is checked to determine the direction of rotation of the motor. For the x280x devices, the QDF bit in QEOSTS register is used to check the rotational direction.

The capture module (CAP3) is configured to generate an interrupt on every rising edge of the QEP_index signal. In the corresponding CAP3 interrupt routine the function F281X_EV1_QEP_Isr() is called. This function resets the timer counter T2CNT and sets the index synchronization flag IndexSyncFlag to 00F0. Thus the counter T2CNT gets reset and starts counting the QEP_CLK pulses every time a QEP_index high pulse is generated.

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

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

\[
\text{MechScaler} \times 4000 = 360^\circ \text{mechanical} = 1 \text{ per unit (pu) mechanical displacement}
\]

\[
\Rightarrow \text{MechScaler} = \left(\frac{1}{4000}\right) \text{pu mech displacement / count}
\]

\[
= 16777 \text{pu mech displacement / count (in Q30)}
\]

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

\[
\text{MechTheta} = \text{MechScaler} \times \text{RawTheta}
\]
If the number of pole pair is \textit{polepairs}, then the pu electrical displacement is given by,

\[
\text{ElecTheta} = \text{PolePairs} \times \text{MechTheta}
\]
**CAP_EVENT_DRV**

**Capture Input Event Driver**

**Description**

This module provides the instantaneous value of the selected time base (GP Timer) captured on the occurrence of an event. Such events can be any specified transition of a signal applied at the event manager (EV) capture input pins of 281x devices or ECAP input pins of 280x devices.

**Availability**

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

1) The C interface version

**Module Properties**

**Type:** Target Dependent, Application Independent

**Target Devices:** x281x or x280x

**C Version File Names:**
- f281xcap.c, f281xcap.h (for x281x)
- f280xcap.c, f280xcap.h (for x280x)

**IQmath library files for C:** N/A

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size (x281x/x280x)</td>
<td>32/34 words</td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

* Each pre-initialized CAPTURE structure consumes 6 words in the data memory

* Code size mentioned here is the size of the `init()` and `read()` functions
C Interface

Object Definition

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

x281x series

typedef struct {
    Uint32 TimeStamp; // Output: Timer value when capture is detected (Q0)
    void (*init)();   // Pointer to the init function
    Uint16 (*read)(); // Pointer to the read function
} CAPTURE;

x280x series

typedef struct {
    int32 EventPeriod; // Output: Timer value difference between two edges (Q0)
    void (*init)();    // Pointer to the init function
    Uint16 (*read)();  // Pointer to the read function
} CAPTURE;

typedef CAPTURE *CAPTURE_handle;

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>CAPn</td>
<td>Capture input signals to 281x device</td>
<td>N/A</td>
<td>0-3.3 v</td>
</tr>
<tr>
<td></td>
<td>(n=1,2,3,4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Outputs</td>
<td>TimeStamp</td>
<td>Timer value for capture unit FIFO registers.</td>
<td>0</td>
<td>00000000-0000FFFF</td>
</tr>
<tr>
<td></td>
<td>(x281x)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EventPeriod</td>
<td>Timer value difference between two edges detected.</td>
<td>0</td>
<td>80000000-7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td>(x280x)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Special Constants and Data types

CAPTURE

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

CAPTURE_handle

User defined Data type of pointer to CAPTURE module

CAPTURE_DEFAULTS

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

Methods

void F281X_EV1_CAP_Init(CAPTURE *);
void F281X_EV1_CAP_Read(CAPTURE *);

void F280X_CAP_Init(CAPTURE *);
void F280X_CAP_Read(CAPTURE *);
This default definition of the object implements two methods – the initialization and the runtime compute function for CAPTURE generation. This is implemented by means of a function pointer, and the initializer sets this to F281X_EV1_CAP_Init and F281X_EV1_CAP_Read functions for x281x or F280X_CAP_Init and F280X_CAP_Read functions for x280x. The argument to this function is the address of the CAPTURE object.

Module Usage

Instantiation
The following example instances one CAPTURE object
CAPTURE cap1;

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

Invoking the computation function
cap1.init(&cap1);
cap1.read(&cap1);

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

main()
{
    cap1.init(&cap1); // Call init function for cap1
}

void interrupt periodic_interrupt_isr()
{
    Uint16 Status;
    Uint32 time_of_event;
    status = cap1.read(&cap1); // Call the capture read function

    // if status==1 then a time stamp was not read,
    // if status==0 then a time stamp was read
    if(status==0)
    {
        time_of_event = (Uint32)(cap1.TimeStamp); // Read out new time stamp
    }
}
**SPEED_FRQ**

**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**
This IQ module is available in one interface format:
1) The C interface version

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

**Target Devices:** x281x or x280x

**C Version File Names:** speed_fr.c, speed_fr.h

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

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size(^1)</td>
<td>104/104 words</td>
<td></td>
</tr>
<tr>
<td>x281x/x280x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words(^*)</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

\(^*\) Each pre-initialized "_iq" SPEED_MEAS_QEP structure consumes 20 words in the data memory

\(^1\) Code size mentioned here is the size of the `calc()` function
C Interface

Object Definition

The structure of SPEED_MEAS_QEP object is defined by the 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)
    void (*calc)();          // Pointer to the calculation function
} SPEED_MEAS_QEP;        // Data type created

typedef SPEED_MEAS_QEP * SPEED_MEAS_QEP_handle;

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

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_handle**
User defined Data type of pointer to SPEED_MEAS_QEP module

**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.
Methods

    void speed_calc(SPEED_MEAS_QEP_handle);

This definition implements one method viz., the speed calculation based on measured rotor angle computation function. The input argument to this function is the module handle.

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 function
speed1.calc(&speed1);
speed2.calc(&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

    speed1.calc(&speed1);       // Call compute function for speed1
    speed2.calc(&speed2);       // Call compute function 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, $\theta_e$, 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 $360^\circ$ 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 rotor speed.

\[ \omega_e(k) = K_1(\theta_e(k) - \theta_e(k-1)) \]  

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

Figure 1: The waveforms of rotor position in both directions
Technical Background

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

\[
\frac{d\hat{\omega}_e}{dt} = \frac{1}{\tau_c} (\omega_e - \hat{\omega}_e)
\]  

(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

\[
\hat{\omega}_e(k) = K_2 \hat{\omega}_e(k-1) + K_3 \omega_e(k)
\]  

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

<table>
<thead>
<tr>
<th>Equation Variables</th>
<th>Program Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td></td>
</tr>
<tr>
<td>( \theta_e )</td>
<td>ElecTheta</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>( \hat{\omega}_e )</td>
<td>Speed</td>
</tr>
<tr>
<td><strong>Others</strong></td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td>K1</td>
</tr>
<tr>
<td>K2</td>
<td>K2</td>
</tr>
<tr>
<td>K3</td>
<td>K3</td>
</tr>
</tbody>
</table>

Table 1: Correspondence of notations
**Description**

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

**Availability**

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

1) The CcA interface version

**Module Properties**

**Type**: Target Dependent, Application Independent

**Target Devices**: x281x or x280x

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

**IQmath library files for C**: N/A

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size†</td>
<td>75/75 words</td>
<td></td>
</tr>
<tr>
<td>(x281x/x280x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words*</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

* Each pre-initialized DLOG_4CH structure consumes 21 words in the data memory

† Code size mentioned here is the size of the *init()* and *update()* functions
C Interface

Object Definition

The structure of DLOG_4CH object is defined by following structure definition

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

typedef DLOG_4CH *DLOG_4CH_handle;

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

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

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

Special Constants and Data types

DLOG_4CH

The module definition is created as a data type. This makes it convenient to instance an interface to the DATALOG driver. To create multiple instances of the module simply declare variables of type DLOG_4CH.
**DLOG_4CH_handle**
User defined Data type of pointer to DATALOG module

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

**Methods**

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

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

**Module Usage**

**Instantiation**
The following example instances one DATALOG object

```c
DLOG_4CH dlog1;
```

**Initialization**
To Instance pre-initialized objects

```c
DLOG_4CH dlog1 = DLOG_4CH_DEFAULTS;
```

**Invoking the computation function**

```c
dlog1.init(&dlog1);
dlog1.update(&dlog1);
```

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

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

void interrupt periodic_interrupt_isr()
{
    dlog1.update(&dlog1);            // Call update function for dlog1
}
```
Technical Background

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

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