```
/* ============================================================================
System Name:      HVACI_Sensored

File Name:        HVACI_Sensored.C

Description:      Primary system file for the Real Implementation of Sensored
                  Field Orientation Control for Induction Motors. Supports F2803x.
============================================================================
*/

// Include header files used in the main function

#include "PeripheralHeaderIncludes.h"
#define   MATH_TYPE      IQ_MATH
#include "IQmathLib.h"
#include "HVACI_Sensored.h"
#include "HVACI_Sensored-Settings.h"
#include <math.h>


#ifdef FLASH
#pragma CODE_SECTION(MainISR,"ramfuncs");
#pragma CODE_SECTION(OffsetISR,"ramfuncs");
#endif

// Prototype statements for functions found within this file.
interrupt void MainISR(void);
interrupt void OffsetISR(void);
void DeviceInit();
void MemCopy();
void InitFlash();
void HVDMC_Protection(void);

// State Machine function prototypes
//-----------------------------------
// Alpha states
void A0(void);    //state A0
void B0(void);    //state B0
void C0(void);    //state C0

// A branch states
void A1(void);    //state A1
void A2(void);    //state A2
void A3(void);    //state A3

// B branch states
void B1(void);    //state B1
void B2(void);    //state B2
void B3(void);    //state B3

// C branch states
void C1(void);    //state C1
void C2(void);    //state C2
void C3(void);    //state C3

// Variable declarations
void (*Alpha_State_Ptr)(void);     // Base States pointer
void (*A_Task_Ptr)(void);          // State pointer A branch
void (*B_Task_Ptr)(void);          // State pointer B branch
```

```
void (*C_Task_Ptr)(void);          // State pointer C branch

// Used for running BackGround in flash, and ISR in RAM
extern Uint16 *RamfuncsLoadStart, *RamfuncsLoadEnd, *RamfuncsRunStart;


int16 VTimer0[4];            // Virtual Timers slaved off CPU Timer 0 (A events)
int16 VTimer1[4];            // Virtual Timers slaved off CPU Timer 1 (B events)
int16 VTimer2[4];            // Virtual Timers slaved off CPU Timer 2 (C events)
int16 SerialCommsTimer;

// Global variables used in this system
float32 T = 0.001/ISR_FREQUENCY;    // Samping period (sec), see HVACI_Sensored-
Settings.H

Uint16 OffsetFlag=0;
_iq offsetA=0;
_iq offsetB=0;
_iq offsetC=0;
_iq K1=_IQ(0.998);           //Offset filter coefficient K1: 0.05/(T+0.05);
_iq K2=_IQ(0.001999);   //Offset filter coefficient K2: T/(T+0.05);

extern _iq IQsinTable[];
extern _iq IQcosTable[];

_iq VdTesting = _IQ(0.2);               // Vd reference (pu)
_iq VqTesting = _IQ(0.2);               // Vq reference (pu)
_iq IdRef     = _IQ(0.1);               // Id reference (pu)
_iq IqRef     = _IQ(0.05);              // Iq reference (pu)
_iq SpeedRef  = _IQ(0.3);               // Speed reference (pu)


Uint32 IsrTicker = 0;
Uint16 BackTicker = 0;
Uint16 lsw=0;                                 // LoopSwitch
Uint16 TripFlagDMC=0;                   // PWM trip status

// Default ADC initialization
int ChSel[16]   = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int   TrigSel[16] = {5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5};
int ACQPS[16]   = {8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8};

int16 DlogCh1 = 0;
int16 DlogCh2 = 0;
int16 DlogCh3 = 0;
int16 DlogCh4 = 0;

volatile Uint16 EnableFlag = FALSE;

Uint16 SpeedLoopPrescaler = 10;      // Speed loop prescaler
Uint16 SpeedLoopCount = 1;           // Speed loop counter

// Instance a current model object
CURMOD cm1 = CURMOD_DEFAULTS;

// Instance a current model constant object
CURMOD_CONST cm1_const = CURMOD_CONST_DEFAULTS;

// Instance a QEP interface driver
```

```
QEP qep1 = QEP_DEFAULTS;

// Instance a Capture interface driver
CAPTURE cap1 = CAPTURE_DEFAULTS;

// Instance a few transform objects (ICLARKE is added into SVGEN module)
CLARKE clarke1 = CLARKE_DEFAULTS;
PARK park1 = PARK_DEFAULTS;
IPARK ipark1 = IPARK_DEFAULTS;

// Instance PI regulators to regulate the d and q  axis currents, and speed
PI_CONTROLLER pi_spd = PI_CONTROLLER_DEFAULTS;
PI_CONTROLLER pi_id  = PI_CONTROLLER_DEFAULTS;
PI_CONTROLLER pi_iq  = PI_CONTROLLER_DEFAULTS;

// Instance a PWM driver instance
PWMGEN pwm1 = PWMGEN_DEFAULTS;

// Instance a PWM DAC driver instance
PWMDAC pwmdac1 = PWMDAC_DEFAULTS;

// Instance a Space Vector PWM modulator. This modulator generates a, b and c
// phases based on the d and q stationery reference frame inputs
SVGEN svgen1 = SVGEN_DEFAULTS;

// Instance a ramp controller to smoothly ramp the frequency
RMPCNTL rc1 = RMPCNTL_DEFAULTS;

//    Instance a ramp(sawtooth) generator to simulate an Anglele
RAMPGEN rg1 = RAMPGEN_DEFAULTS;

// Instance a speed calculator based on QEP
SPEED_MEAS_QEP speed1 = SPEED_MEAS_QEP_DEFAULTS;

// Instance a speed calculator based on capture Qep (for eQep of 280x only)
SPEED_MEAS_CAP speed2 = SPEED_MEAS_CAP_DEFAULTS;

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



void main(void)
{

      DeviceInit();     // Device Life support & GPIO

// Only used if running from FLASH
// Note that the variable FLASH is defined by the compiler

#ifdef FLASH
// Copy time critical code and Flash setup code to RAM
// The  RamfuncsLoadStart, RamfuncsLoadEnd, and RamfuncsRunStart
// symbols are created by the linker. Refer to the linker files.
      MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);

// Call Flash Initialization to setup flash waitstates
// This function must reside in RAM
      InitFlash();      // Call the flash wrapper init function
```

```
#endif //(FLASH)

   // Waiting for enable flag set
   while (EnableFlag==FALSE)
    {
      BackTicker++;
    }

// Timing sync for background loops
// Timer period definitions found in device specific PeripheralHeaderIncludes.h
      CpuTimer0Regs.PRD.all =  mSec1;          // A tasks
      CpuTimer1Regs.PRD.all =  mSec5;          // B tasks
      CpuTimer2Regs.PRD.all =  mSec50;    // C tasks

// Tasks State-machine init
      Alpha_State_Ptr = &A0;
      A_Task_Ptr = &A1;
      B_Task_Ptr = &B1;
      C_Task_Ptr = &C1;

// Initialize PWM module
    pwm1.PeriodMax = SYSTEM_FREQUENCY*1000000*T/2;  // Prescaler X1 (T1), ISR
period = T x 1
    pwm1.HalfPerMax=pwm1.PeriodMax/2;
    pwm1.Deadband  = 2.0*SYSTEM_FREQUENCY;         // 120 counts -> 2.0 usec for
TBCLK = SYSCLK/1
    PWM_INIT_MACRO(1,2,3,pwm1)

// Initialize PWMDAC module
      pwmdac1.PeriodMax=500;             // @60Mhz, 1500->20kHz, 1000-> 30kHz,
500->60kHz
      pwmdac1.HalfPerMax=pwmdac1.PeriodMax/2;
      PWMDAC_INIT_MACRO(6,pwmdac1)  // PWM 6A,6B
      PWMDAC_INIT_MACRO(7,pwmdac1)  // PWM 7A,7B

// Initialize DATALOG module
    dlog.iptr1 = &DlogCh1;
    dlog.iptr2 = &DlogCh2;
    dlog.iptr3 = &DlogCh3;
    dlog.iptr4 = &DlogCh4;
    dlog.trig_value = 0x1;
    dlog.size = 0x0C8;
    dlog.prescalar = 5;
    dlog.init(&dlog);

// Initialize ADC for DMC Kit Rev 1.1
      ChSel[0]=1;       // Dummy meas. avoid 1st sample issue Rev0 Picollo*/
      ChSel[1]=1;       // ChSelect: ADC A1-> Phase A Current
      ChSel[2]=9;       // ChSelect: ADC B1-> Phase B Current
      ChSel[3]=3;       // ChSelect: ADC A3-> Phase C Current
      ChSel[4]=15;      // ChSelect: ADC B7-> Phase A Voltage
      ChSel[5]=14;      // ChSelect: ADC B6-> Phase B Voltage
      ChSel[6]=12;      // ChSelect: ADC B4-> Phase C Voltage
      ChSel[7]=7;       // ChSelect: ADC A7-> DC Bus  Voltage

      ADC_MACRO_INIT(ChSel,TrigSel,ACQPS)

// Initialize QEP module
      qep1.LineEncoder = 2048;
```

```
    qep1.MechScaler = _IQ30(0.25/qep1.LineEncoder);
    qep1.PolePairs = POLES/2;
    qep1.CalibratedAngle = 0;
      QEP_INIT_MACRO(1,qep1)

// Initialize CAP module
      CAP_INIT_MACRO(1)

// Initialize the Speed module for QEP based speed calculation
    speed1.K1 = _IQ21(1/(BASE_FREQ*T));
    speed1.K2 = _IQ(1/(1+T*2*PI*5));  // Low-pass cut-off frequency
    speed1.K3 = _IQ(1)-speed1.K2;
    speed1.BaseRpm = 120*(BASE_FREQ/POLES);

// Initialize the Speed module for capture eQEP based speed calculation (low speed
range)
    speed2.InputSelect = 1;
      speed2.BaseRpm       = 120*(BASE_FREQ/POLES);
      speed2.SpeedScaler = 60*(SYSTEM_FREQUENCY*1000000/(1*2048*speed2.BaseRpm));

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

// Initialize the CUR_MOD constant module
      cm1_const.Rr = RR;
      cm1_const.Lr = LR;
      cm1_const.fb = BASE_FREQ;
      cm1_const.Ts = T;
      CUR_CONST_MACRO(cm1_const)

// Initialize the CUR_MOD module
      cm1.Kr = _IQ(cm1_const.Kr);
      cm1.Kt = _IQ(cm1_const.Kt);
      cm1.K  = _IQ(cm1_const.K);

// Initialize the PI module for Id
    pi_spd.Kp=_IQ(2.0);
      pi_spd.Ki=_IQ(T*SpeedLoopPrescaler/0.5);
      pi_spd.Umax =_IQ(0.95);
      pi_spd.Umin =_IQ(-0.95);

// Initialize the PI module for Iq
      pi_id.Kp=_IQ(1.0);
      pi_id.Ki=_IQ(T/0.004);
      pi_id.Umax =_IQ(0.3);
      pi_id.Umin =_IQ(-0.3);

// Initialize the PI module for speed
      pi_iq.Kp=_IQ(1.0);
      pi_iq.Ki=_IQ(T/0.004);
      pi_iq.Umax =_IQ(0.80);
      pi_iq.Umin =_IQ(-0.80);

//  Note that the vectorial sum of d-q PI outputs should be less than 1.0 which
refers to maximum duty cycle for SVGEN.
//  Another duty cycle limiting factor is current sense through shunt resistors
which depends on hardware/software implementation.
//  Depending on the application requirements 3,2 or a single shunt resistor can be
used for current waveform reconstruction.
```

```
//  The higher number of shunt resistors allow the higher duty cycle operation and
better dc bus utilization.
//  The users should adjust the PI saturation levels carefully during open loop
tests (i.e pi_id.Umax, pi_iq.Umax and Umins) as in project manuals.
//  Violation of this procedure yields distorted current waveforms and unstable
closed loop operations which may damage the inverter.

//Call HVDMC Protection function
      HVDMC_Protection();

// Reassign ISRs.

      EALLOW;     // This is needed to write to EALLOW protected registers

      PieVectTable.ADCINT1 = &OffsetISR;

// Enable PIE group 1 interrupt 1 for ADC1_INT
      PieCtrlRegs.PIEIER1.bit.INTx1 = 1;

// Enable EOC interrupt(after the 4th conversion)

      AdcRegs.ADCINTOVFCLR.bit.ADCINT1=1;
      AdcRegs.ADCINTFLGCLR.bit.ADCINT1=1;
      AdcRegs.INTSEL1N2.bit.INT1CONT=1;
      AdcRegs.INTSEL1N2.bit.INT1SEL=4;
      AdcRegs.INTSEL1N2.bit.INT1E=1;

// Enable CPU INT1 for ADC1_INT:
      IER |= M_INT1;

// Enable global Interrupts and higher priority real-time debug events:
      EINT;   // Enable Global interrupt INTM
      ERTM; // Enable Global realtime interrupt DBGM

      EDIS;


// IDLE loop. Just sit and loop forever:
      for(;;)  //infinite loop
      {
            // State machine entry & exit point
            //===========================================================
            (*Alpha_State_Ptr)();   // jump to an Alpha state (A0,B0,...)
            //===========================================================

      }
} //END MAIN CODE



//===========================================================================
//    STATE-MACHINE SEQUENCING AND SYNCRONIZATION FOR SLOW BACKGROUND TASKS
//===========================================================================

//--------------------------------- FRAMEWORK -------------------------------------
void A0(void)
{
      // loop rate synchronizer for A-tasks
      if(CpuTimer0Regs.TCR.bit.TIF == 1)
```

```
	{
		CpuTimer0Regs.TCR.bit.TIF = 1;	// clear flag

		//-----------------------------------------------------------
		(*A_Task_Ptr)();	// jump to an A Task (A1,A2,A3,...)
		//-----------------------------------------------------------

		VTimer0[0]++;			// virtual timer 0, instance 0 (spare)
		SerialCommsTimer++;
	}

	Alpha_State_Ptr = &B0;	// Comment out to allow only A tasks
}

void B0(void)
{
	// loop rate synchronizer for B-tasks
	if(CpuTimer1Regs.TCR.bit.TIF == 1)
	{
		CpuTimer1Regs.TCR.bit.TIF = 1;			// clear flag

		//-----------------------------------------------------------
		(*B_Task_Ptr)();	// jump to a B Task (B1,B2,B3,...)
		//-----------------------------------------------------------
		VTimer1[0]++;			// virtual timer 1, instance 0 (spare)
	}

	Alpha_State_Ptr = &C0;	// Allow C state tasks
}

void C0(void)
{
	// loop rate synchronizer for C-tasks
	if(CpuTimer2Regs.TCR.bit.TIF == 1)
	{
		CpuTimer2Regs.TCR.bit.TIF = 1;			// clear flag

		//-----------------------------------------------------------
		(*C_Task_Ptr)();	// jump to a C Task (C1,C2,C3,...)
		//-----------------------------------------------------------
		VTimer2[0]++;			//virtual timer 2, instance 0 (spare)
	}

	Alpha_State_Ptr = &A0;  // Back to State A0
}


//===========================================================================
//    A - TASKS (executed in every 1 msec)
//===========================================================================
//-----------------------------------------------------------
void A1(void) // SPARE (not used)
//-----------------------------------------------------------
{

	//-------------------
	//the next time CpuTimer0 'counter' reaches Period value go to A2
	A_Task_Ptr = &A2;
	//-------------------
```

```
}

//---------------------------------------------------------------
void A2(void) // SPARE (not used)
//---------------------------------------------------------------
{

      //-------------------
      //the next time CpuTimer0 'counter' reaches Period value go to A3
      A_Task_Ptr = &A3;
      //-------------------
}

//---------------------------------------
void A3(void) // SPARE (not used)
//---------------------------------------
{

      //-----------------
      //the next time CpuTimer0 'counter' reaches Period value go to A1
      A_Task_Ptr = &A1;
      //-----------------
}



//===========================================================================
//    B - TASKS (executed in every 5 msec)
//===========================================================================

//----------------------------------- USER -----------------------------------

//---------------------------------------
void B1(void) // Toggle GPIO-00
//---------------------------------------
{

      //-----------------
      //the next time CpuTimer1 'counter' reaches Period value go to B2
      B_Task_Ptr = &B2;
      //-----------------
}

//---------------------------------------
void B2(void) //  SPARE
//---------------------------------------
{

      //-----------------
      //the next time CpuTimer1 'counter' reaches Period value go to B3
      B_Task_Ptr = &B3;
      //-----------------
}

//---------------------------------------
void B3(void) //  SPARE
//---------------------------------------
{
```

```
      //-----------------
      //the next time CpuTimer1 'counter' reaches Period value go to B1
      B_Task_Ptr = &B1;
      //-----------------
}


//===========================================================================
//    C - TASKS (executed in every 50 msec)
//===========================================================================

//--------------------------------- USER -----------------------------------

//---------------------------------------
void C1(void)     // Toggle GPIO-34, GPIO42/44
//---------------------------------------
{
      if(EPwm1Regs.TZFLG.bit.OST==0x1)               // TripZ for PWMs is low
(fault trip)
        { TripFlagDMC=1;
        GpioDataRegs.GPBTOGGLE.bit.GPIO42 = 1;
        }

      if(GpioDataRegs.GPADAT.bit.GPIO15 == 1)        // Over Current Prot. for
Integrated Power Module is high (fault trip)
        { TripFlagDMC=1;
        GpioDataRegs.GPBTOGGLE.bit.GPIO44 = 1;
        }

      GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;      // Turn on/off LD3 on the
controlCARD
      //-----------------
      //the next time CpuTimer2 'counter' reaches Period value go to C2
      C_Task_Ptr = &C2;
      //-----------------

}

//---------------------------------------
void C2(void) //  SPARE
//---------------------------------------
{

      //-----------------
      //the next time CpuTimer2 'counter' reaches Period value go to C3
      C_Task_Ptr = &C3;
      //-----------------
}


//---------------------------------------
void C3(void) //  SPARE
//---------------------------------------
{

      //-----------------
      //the next time CpuTimer2 'counter' reaches Period value go to C1
      C_Task_Ptr = &C1;
      //-----------------
```

```
}


//MainISR
interrupt void MainISR(void)
{

// Verifying the ISR
    IsrTicker++;


// ================================ LEVEL 1 ====================================
//      Checks target independent modules, duty cycle waveforms and PWM update
//      Keep the motors disconnected at this level!
// =============================================================================

#if (BUILDLEVEL==LEVEL1)

// -----------------------------------------------------------------------------
//  Connect inputs of the RMP module and call the ramp control macro
// -----------------------------------------------------------------------------
    rc1.TargetValue = SpeedRef;
      RC_MACRO(rc1)

// -----------------------------------------------------------------------------
//  Connect inputs of the RAMP GEN module and call the ramp generator macro
// -----------------------------------------------------------------------------
    rg1.Freq = rc1.SetpointValue;
      RG_MACRO(rg1)

// -----------------------------------------------------------------------------
//  Connect inputs of the INV_PARK module and call the inverse park trans. macro
//    There are two option for trigonometric functions:
//  IQ sin/cos look-up table provides 512 discrete sin and cos points in Q30 format
//  IQsin/cos PU functions interpolate the data in the lookup table yielding higher
resolution.
// -----------------------------------------------------------------------------
    ipark1.Ds = VdTesting;
    ipark1.Qs = VqTesting;

      //ipark1.Sine  =_IQ30toIQ(IQsinTable[_IQtoIQ9(rg1.Out)]);
    //ipark1.Cosine=_IQ30toIQ(IQcosTable[_IQtoIQ9(rg1.Out)]);

      ipark1.Sine=_IQsinPU(rg1.Out);
    ipark1.Cosine=_IQcosPU(rg1.Out);
      IPARK_MACRO(ipark1)

// -----------------------------------------------------------------------------
//  Connect inputs of the SVGEN module and call the space-vector gen. macro
// -----------------------------------------------------------------------------
      svgen1.Ualpha = ipark1.Alpha;
      svgen1.Ubeta  = ipark1.Beta;
      SVGENDQ_MACRO(svgen1)

// -----------------------------------------------------------------------------
//  Connect inputs of the PWM_DRV module and call the PWM signal generation macro
// -----------------------------------------------------------------------------
    pwm1.MfuncC1 = svgen1.Ta;
```

```
    pwm1.MfuncC2 = svgen1.Tb;
    pwm1.MfuncC3 = svgen1.Tc;
      PWM_MACRO(1,2,3,pwm1)                                  // Calculate the
new PWM compare values

// ---------------------------------------------------------------------------
//    Connect inputs of the PWMDAC module
// ---------------------------------------------------------------------------
      pwmdac1.MfuncC1 = svgen1.Ta;
    pwmdac1.MfuncC2 = svgen1.Tb;
    PWMDAC_MACRO(6,pwmdac1)                                  // PWMDAC 6A, 6B

    pwmdac1.MfuncC1 = svgen1.Tc;
    pwmdac1.MfuncC2 = svgen1.Tb-svgen1.Tc;
      PWMDAC_MACRO(7,pwmdac1)                                // PWMDAC 7A, 7B

// ---------------------------------------------------------------------------
//    Connect inputs of the DATALOG module
// ---------------------------------------------------------------------------
    DlogCh1 = (int16)_IQtoIQ15(svgen1.Ta);
    DlogCh2 = (int16)_IQtoIQ15(svgen1.Tb);
    DlogCh3 = (int16)_IQtoIQ15(svgen1.Tc);
    DlogCh4 = (int16)_IQtoIQ15(svgen1.Tb-svgen1.Tc);

#endif // (BUILDLEVEL==LEVEL1)

// ================================ LEVEL 2 ===================================
//      Level 2 verifies the analog-to-digital conversion, offset compensation,
//    clarke/park transformations (CLARKE/PARK),
// ============================================================================

#if (BUILDLEVEL==LEVEL2)


// ---------------------------------------------------------------------------
//  Connect inputs of the RMP module and call the ramp control macro
// ---------------------------------------------------------------------------
    rc1.TargetValue = SpeedRef;
      RC_MACRO(rc1)

// ---------------------------------------------------------------------------
//  Connect inputs of the RAMP GEN module and call the ramp generator macro
// ---------------------------------------------------------------------------
    rg1.Freq = rc1.SetpointValue;
      RG_MACRO(rg1)

// ---------------------------------------------------------------------------
//  Measure phase currents, subtract the offset and normalize from (-0.5,+0.5) to
(-1,+1).
//    Connect inputs of the CLARKE module and call the clarke transformation macro
// ---------------------------------------------------------------------------
      clarke1.As = _IQmpy2(_IQ12toIQ(AdcResult.ADCRESULT1)-offsetA); // Phase A
curr.
      clarke1.Bs = _IQmpy2(_IQ12toIQ(AdcResult.ADCRESULT2)-offsetB); // Phase B
curr.
      CLARKE_MACRO(clarke1)

// ---------------------------------------------------------------------------
//  Connect inputs of the PARK module and call the park trans. macro
```

```
// ---------------------------------------------------------------------------
      park1.Alpha  = clarke1.Alpha;
      park1.Beta   = clarke1.Beta;
      park1.Angle  = rg1.Out;
      park1.Sine   = _IQsinPU(park1.Angle);
      park1.Cosine = _IQcosPU(park1.Angle);
      PARK_MACRO(park1)

// ---------------------------------------------------------------------------
//    Connect inputs of the INV_PARK module and call the inverse park trans. macro
// ---------------------------------------------------------------------------
    ipark1.Ds    = VdTesting;
    ipark1.Qs    = VqTesting;
      ipark1.Sine  = park1.Sine;
    ipark1.Cosine= park1.Cosine;
      IPARK_MACRO(ipark1)

// ---------------------------------------------------------------------------
//  Connect inputs of the SVGEN module and call the space-vector gen. macro
// ---------------------------------------------------------------------------
      svgen1.Ualpha = ipark1.Alpha;
      svgen1.Ubeta  = ipark1.Beta;
      SVGENDQ_MACRO(svgen1)

// ---------------------------------------------------------------------------
//  Connect inputs of the PWM_DRV module and call the PWM signal generation macro
// ---------------------------------------------------------------------------
    pwm1.MfuncC1 = svgen1.Ta;
    pwm1.MfuncC2 = svgen1.Tb;
    pwm1.MfuncC3 = svgen1.Tc;
      PWM_MACRO(1,2,3,pwm1)                                  // Calculate the
new PWM compare values

// ---------------------------------------------------------------------------
//    Connect inputs of the PWMDAC module
// ---------------------------------------------------------------------------
      pwmdac1.MfuncC1 = clarke1.As;
    pwmdac1.MfuncC2 = clarke1.Bs;
    PWMDAC_MACRO(6,pwmdac1)                                  // PWMDAC 6A, 6B

    pwmdac1.MfuncC1 = rg1.Out;
    pwmdac1.MfuncC2 = svgen1.Ta;
      PWMDAC_MACRO(7,pwmdac1)                                // PWMDAC 7A, 7B

// ---------------------------------------------------------------------------
//    Connect inputs of the DATALOG module
// ---------------------------------------------------------------------------
    DlogCh1 = (int16)_IQtoIQ15(clarke1.As);
    DlogCh2 = (int16)_IQtoIQ15(clarke1.Bs);
    DlogCh3 = (int16)_IQtoIQ15(rg1.Out);
    DlogCh4 = (int16)_IQtoIQ15(svgen1.Ta);


#endif // (BUILDLEVEL==LEVEL2)

// ================================ LEVEL 3 ===================================
//    Level 3 verifies the dq-axis current regulation performed by PI and
//    speed measurement modules
// ============================================================================
```

```
#if (BUILDLEVEL==LEVEL3)

// --------------------------------------------------------------------------
//  Connect inputs of the RMP module and call the ramp control macro
// --------------------------------------------------------------------------
    rc1.TargetValue = SpeedRef;
      RC_MACRO(rc1)

// --------------------------------------------------------------------------
//  Connect inputs of the RAMP GEN module and call the ramp generator macro
// --------------------------------------------------------------------------
    rg1.Freq = rc1.SetpointValue;
      RG_MACRO(rg1)

// --------------------------------------------------------------------------
//  Measure phase currents, subtract the offset and normalize from (-0.5,+0.5) to
(-1,+1).
//    Connect inputs of the CLARKE module and call the clarke transformation macro
// --------------------------------------------------------------------------
      clarke1.As = _IQmpy2(_IQ12toIQ(AdcResult.ADCRESULT1)-offsetA); // Phase A
curr.
      clarke1.Bs = _IQmpy2(_IQ12toIQ(AdcResult.ADCRESULT2)-offsetB); // Phase B
curr.
      CLARKE_MACRO(clarke1)

// --------------------------------------------------------------------------
//  Connect inputs of the PARK module and call the park trans. macro
// --------------------------------------------------------------------------
      park1.Alpha  = clarke1.Alpha;
      park1.Beta   = clarke1.Beta;
      park1.Angle  = rg1.Out;
      park1.Sine   = _IQsinPU(park1.Angle);
      park1.Cosine = _IQcosPU(park1.Angle);
      PARK_MACRO(park1)

// --------------------------------------------------------------------------
//  Connect inputs of the PI module and call the PI IQ controller macro
// --------------------------------------------------------------------------
    pi_iq.Ref = IqRef;
      pi_iq.Fbk = park1.Qs;
      PI_MACRO(pi_iq)

// --------------------------------------------------------------------------
//  Connect inputs of the PI module and call the PI ID controller macro
// --------------------------------------------------------------------------
    pi_id.Ref = IdRef;
      pi_id.Fbk = park1.Ds;
      PI_MACRO(pi_id)

// --------------------------------------------------------------------------
//    Connect inputs of the INV_PARK module and call the inverse park trans. macro
// --------------------------------------------------------------------------
    ipark1.Ds       = pi_id.Out;
    ipark1.Qs       = pi_iq.Out ;
      ipark1.Sine   = park1.Sine;
    ipark1.Cosine = park1.Cosine;
      IPARK_MACRO(ipark1)
```

```
// ---------------------------------------------------------------------------
//  Call the QEP macro (if incremental encoder used for speed sensing)
//  Connect inputs of the SPEED_FR module and call the speed calculation macro
// ---------------------------------------------------------------------------
    QEP_MACRO(1,qep1)

    speed1.ElecTheta = _IQ24toIQ((int32)qep1.ElecTheta);
    speed1.DirectionQep = (int32)(qep1.DirectionQep);
    SPEED_FR_MACRO(speed1)

// ---------------------------------------------------------------------------
//  Call the CAP macro (if sprocket or spur gear used for speed sensing)
//  Connect inputs of the SPEED_PR module and call the speed calculation macro
// ---------------------------------------------------------------------------
      CAP_MACRO(1,cap1)

      if(cap1.CapReturn ==0)                                  // Check the capture
return
    {
    speed2.EventPeriod=(int32)(cap1.EventPeriod);    // Read out new event period
    SPEED_PR_MACRO(speed2)                             // Call the speed macro

    }

// ---------------------------------------------------------------------------
//  Connect inputs of the SVGEN module and call the space-vector gen. macro
// ---------------------------------------------------------------------------
      svgen1.Ualpha = ipark1.Alpha;
      svgen1.Ubeta  = ipark1.Beta;
      SVGENDQ_MACRO(svgen1)

// ---------------------------------------------------------------------------
//  Connect inputs of the PWM_DRV module and call the PWM signal generation macro
// ---------------------------------------------------------------------------
    pwm1.MfuncC1 = svgen1.Ta;
    pwm1.MfuncC2 = svgen1.Tb;
    pwm1.MfuncC3 = svgen1.Tc;
      PWM_MACRO(1,2,3,pwm1)                                   // Calculate the
new PWM compare values

// ---------------------------------------------------------------------------
//    Connect inputs of the PWMDAC module
// ---------------------------------------------------------------------------
      pwmdac1.MfuncC1 = clarke1.As;
    pwmdac1.MfuncC2 = clarke1.Bs;
    PWMDAC_MACRO(6,pwmdac1)                                   // PWMDAC 6A, 6B

    pwmdac1.MfuncC1 = rg1.Out;
    pwmdac1.MfuncC2 = speed1.ElecTheta;
      PWMDAC_MACRO(7,pwmdac1)                                 // PWMDAC 7A, 7B

// ---------------------------------------------------------------------------
//    Connect inputs of the DATALOG module
// ---------------------------------------------------------------------------
    DlogCh1 = (int16)_IQtoIQ15(clarke1.As);
    DlogCh2 = (int16)_IQtoIQ15(svgen1.Ta);
    DlogCh3 = (int16)_IQtoIQ15(rg1.Out);
    DlogCh4 = (int16)_IQtoIQ15(speed1.ElecTheta);
```

```
#endif // (BUILDLEVEL==LEVEL3)

// ============================== LEVEL 4 ====================================
//      Level 4 verifies the current model (CUR_MOD).
// ===========================================================================

#if (BUILDLEVEL==LEVEL4)

// ---------------------------------------------------------------------------
//  Connect inputs of the RMP module and call the ramp control macro
// ---------------------------------------------------------------------------
    rc1.TargetValue = SpeedRef;
      RC_MACRO(rc1)

// ---------------------------------------------------------------------------
//  Connect inputs of the RAMP GEN module and call the ramp generator macro
// ---------------------------------------------------------------------------
    rg1.Freq = rc1.SetpointValue;
      RG_MACRO(rg1)

// ---------------------------------------------------------------------------
//  Measure phase currents, subtract the offset and normalize from (-0.5,+0.5) to
(-1,+1).
//    Connect inputs of the CLARKE module and call the clarke transformation macro
// ---------------------------------------------------------------------------
      clarke1.As = _IQmpy2(_IQ12toIQ(AdcResult.ADCRESULT1)-offsetA); // Phase A
curr.
      clarke1.Bs = _IQmpy2(_IQ12toIQ(AdcResult.ADCRESULT2)-offsetB); // Phase B
curr.
      CLARKE_MACRO(clarke1)

// ---------------------------------------------------------------------------
//  Connect inputs of the PARK module and call the park trans. macro
// ---------------------------------------------------------------------------
      park1.Alpha = clarke1.Alpha;
      park1.Beta  = clarke1.Beta;
      park1.Angle = rg1.Out;
      park1.Sine  = _IQsinPU(park1.Angle);
      park1.Cosine= _IQcosPU(park1.Angle);
      PARK_MACRO(park1)

// ---------------------------------------------------------------------------
//  Connect inputs of the PI module and call the PID IQ controller macro
// ---------------------------------------------------------------------------
    pi_iq.Ref = IqRef;
      pi_iq.Fbk = park1.Qs;
      PI_MACRO(pi_iq)

// ---------------------------------------------------------------------------
//  Connect inputs of the PI module and call the PID ID controller macro
// ---------------------------------------------------------------------------
    pi_id.Ref = IdRef;
      pi_id.Fbk = park1.Ds;
      PI_MACRO(pi_id)

// ---------------------------------------------------------------------------
//    Connect inputs of the INV_PARK module and call the inverse park trans. macro
```

```
// ---------------------------------------------------------------------------
    ipark1.Ds       = pi_id.Out;
    ipark1.Qs       = pi_iq.Out ;
      ipark1.Sine   = park1.Sine;
    ipark1.Cosine = park1.Cosine;
      IPARK_MACRO(ipark1)

// ---------------------------------------------------------------------------
//  Call the QEP macro (if incremental encoder used for speed sensing)
//  Connect inputs of the SPEED_FR module and call the speed calculation macro
// ---------------------------------------------------------------------------
    QEP_MACRO(1,qep1)

    speed1.ElecTheta = _IQ24toIQ((int32)qep1.ElecTheta);
    speed1.DirectionQep = (int32)(qep1.DirectionQep);
    SPEED_FR_MACRO(speed1)

// ---------------------------------------------------------------------------
//  Call the CAP macro (if sprocket or spur gear used for speed sensing)
//  Connect inputs of the SPEED_PR module and call the speed calculation macro
// ---------------------------------------------------------------------------
      CAP_MACRO(1,cap1)

    if(cap1.CapReturn ==0)                                      // Check the
capture return
    {
        speed2.EventPeriod=(int32)(cap1.EventPeriod);    // Read out new event
period
        SPEED_PR_MACRO(speed2)                              // Call the speed macro

    }

// ---------------------------------------------------------------------------
//    Connect inputs of the CUR_MOD module and call the current model
//    calculation function.
// ---------------------------------------------------------------------------
    cm1.IDs = park1.Ds;
    cm1.IQs = park1.Qs;
      cm1.Wr = speed1.Speed;
      CUR_MOD_MACRO(cm1)

// ---------------------------------------------------------------------------
//  Connect inputs of the SVGEN module and call the space-vector gen. macro
// ---------------------------------------------------------------------------
      svgen1.Ualpha = ipark1.Alpha;
      svgen1.Ubeta  = ipark1.Beta;
      SVGENDQ_MACRO(svgen1)

// ---------------------------------------------------------------------------
//  Connect inputs of the PWM_DRV module and call the PWM signal generation macro
// ---------------------------------------------------------------------------
    pwm1.MfuncC1 = svgen1.Ta;
    pwm1.MfuncC2 = svgen1.Tb;
    pwm1.MfuncC3 = svgen1.Tc;
      PWM_MACRO(1,2,3,pwm1)                                      // Calculate the
new PWM compare values

// ---------------------------------------------------------------------------
//    Connect inputs of the PWMDAC module
```

```
// ---------------------------------------------------------------------------
      pwmdac1.MfuncC1 = clarke1.As;
    pwmdac1.MfuncC2 = cm1.Theta;
    PWMDAC_MACRO(6,pwmdac1)                                     // PWMDAC 6A, 6B

    pwmdac1.MfuncC1 = rg1.Out;
    pwmdac1.MfuncC2 = speed1.ElecTheta ;
      PWMDAC_MACRO(7,pwmdac1)                                   // PWMDAC 7A, 7B

// ---------------------------------------------------------------------------
//    Connect inputs of the DATALOG module
// ---------------------------------------------------------------------------
    DlogCh1 = (int16)_IQtoIQ15(svgen1.Ta);
    DlogCh2 = (int16)_IQtoIQ15(cm1.Theta);
    DlogCh3 = (int16)_IQtoIQ15(clarke1.As);
    DlogCh4 = (int16)_IQtoIQ15(clarke1.Bs);

#endif // (BUILDLEVEL==LEVEL4)


// ================================ LEVEL 5 ===================================
//    Level 5 verifies verifies the speed regulator performed by PI module.
//    The system speed loop is closed by using the measured speed from capture
//    signal as a feedback.
// ============================================================================

#if (BUILDLEVEL==LEVEL5)

// ---------------------------------------------------------------------------
//  Connect inputs of the RMP module and call the ramp control macro
// ---------------------------------------------------------------------------
    rc1.TargetValue = SpeedRef;
      RC_MACRO(rc1)

// ---------------------------------------------------------------------------
//  Connect inputs of the RAMP GEN module and call the ramp generator macro
// ---------------------------------------------------------------------------
    rg1.Freq = rc1.SetpointValue;
      RG_MACRO(rg1)

// ---------------------------------------------------------------------------
//  Measure phase currents, subtract the offset and normalize from (-0.5,+0.5) to
(-1,+1).
//    Connect inputs of the CLARKE module and call the clarke transformation macro
// ---------------------------------------------------------------------------
      clarke1.As = _IQmpy2(_IQ12toIQ(AdcResult.ADCRESULT1)-offsetA); // Phase A
curr.
      clarke1.Bs = _IQmpy2(_IQ12toIQ(AdcResult.ADCRESULT2)-offsetB); // Phase B
curr.
      CLARKE_MACRO(clarke1)

// ---------------------------------------------------------------------------
//  Connect inputs of the PARK module and call the park trans. macro
// ---------------------------------------------------------------------------
      park1.Alpha = clarke1.Alpha;
      park1.Beta  = clarke1.Beta;
      park1.Angle = cm1.Theta;
      park1.Sine  = _IQsinPU(park1.Angle);
      park1.Cosine= _IQcosPU(park1.Angle);
```

```
      PARK_MACRO(park1)

// ------------------------------------------------------------------------------
//  Connect inputs of the PI module and call the PID IQ controller macro
// ------------------------------------------------------------------------------
    if (SpeedLoopCount==SpeedLoopPrescaler)
     {
        pi_spd.Ref = rc1.SetpointValue;
        pi_spd.Fbk = speed1.Speed;
        PI_MACRO(pi_spd)

      SpeedLoopCount=1;
     }
    else SpeedLoopCount++;

// ------------------------------------------------------------------------------
//  Connect inputs of the PI module and call the PID IQ controller macro
// ------------------------------------------------------------------------------
    pi_iq.Ref = pi_spd.Out;
      pi_iq.Fbk = park1.Qs;
      PI_MACRO(pi_iq)

// ------------------------------------------------------------------------------
//  Connect inputs of the PI module and call the PID ID controller macro
// ------------------------------------------------------------------------------
      pi_id.Ref = IdRef;
      pi_id.Fbk = park1.Ds;
      PI_MACRO(pi_id)

// ------------------------------------------------------------------------------
//    Connect inputs of the INV_PARK module and call the inverse park trans. macro
// ------------------------------------------------------------------------------
      ipark1.Ds = pi_id.Out;
    ipark1.Qs = pi_iq.Out ;
      ipark1.Sine   = park1.Sine;
    ipark1.Cosine = park1.Cosine;
      IPARK_MACRO(ipark1)

// ------------------------------------------------------------------------------
//  Call the QEP macro (if incremental encoder used for speed sensing)
//  Connect inputs of the SPEED_FR module and call the speed calculation macro
// ------------------------------------------------------------------------------
    QEP_MACRO(1,qep1)

    speed1.ElecTheta = _IQ24toIQ((int32)qep1.ElecTheta);
    speed1.DirectionQep = (int32)(qep1.DirectionQep);
    SPEED_FR_MACRO(speed1)

/* ------------------------------------------------------------------------------
//  Call the CAP macro (if sprocket or spur gear used for speed sensing)
//  Connect inputs of the SPEED_PR module and call the speed calculation macro
// ------------------------------------------------------------------------------
      CAP_MACRO(1,cap1)

    if(cap1.CapReturn ==0)                                          // Check the
capture return
    {
        speed2.EventPeriod=(int32)(cap1.EventPeriod);    // Read out new event
period
```

```
        SPEED_PR_MACRO(speed2)                          // Call the speed macro

    }
*/
// ---------------------------------------------------------------------------
//    Connect inputs of the CUR_MOD module and call the current model
//    calculation function.
// ---------------------------------------------------------------------------
    cm1.IDs = park1.Ds;
    cm1.IQs = park1.Qs;
      cm1.Wr = speed1.Speed;
      CUR_MOD_MACRO(cm1)

// ---------------------------------------------------------------------------
//  Connect inputs of the SVGEN module and call the space-vector gen. macro
// ---------------------------------------------------------------------------
      svgen1.Ualpha = ipark1.Alpha;
      svgen1.Ubeta  = ipark1.Beta;
      SVGENDQ_MACRO(svgen1)

// ---------------------------------------------------------------------------
//  Connect inputs of the PWM_DRV module and call the PWM signal generation macro
// ---------------------------------------------------------------------------
    pwm1.MfuncC1 = svgen1.Ta;
    pwm1.MfuncC2 = svgen1.Tb;
    pwm1.MfuncC3 = svgen1.Tc;
      PWM_MACRO(1,2,3,pwm1)                                 // Calculate the
new PWM compare values

// ---------------------------------------------------------------------------
//    Connect inputs of the PWMDAC module
// ---------------------------------------------------------------------------
      pwmdac1.MfuncC1 = clarke1.As;
    pwmdac1.MfuncC2 = cm1.Theta;
    PWMDAC_MACRO(6,pwmdac1)                                 // PWMDAC 6A, 6B

    pwmdac1.MfuncC1 = rg1.Out;
    pwmdac1.MfuncC2 = speed1.ElecTheta ;
      PWMDAC_MACRO(7,pwmdac1)                               // PWMDAC 7A, 7B

// ---------------------------------------------------------------------------
//    Connect inputs of the DATALOG module
// ---------------------------------------------------------------------------
    DlogCh1 = (int16)_IQtoIQ15(svgen1.Ta);
    DlogCh2 = (int16)_IQtoIQ15(cm1.Theta);
    DlogCh3 = (int16)_IQtoIQ15(clarke1.As);
    DlogCh4 = (int16)_IQtoIQ15(clarke1.Bs);

#endif // (BUILDLEVEL==LEVEL5)


// ---------------------------------------------------------------------------
//    Call the DATALOG update function.
// ---------------------------------------------------------------------------
    dlog.update(&dlog);

// Enable more interrupts from this timer
      AdcRegs.ADCINTFLG.bit.ADCINT1=1;
```

```
// Acknowledge interrupt to recieve more interrupts from PIE group 3
      PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

}// MainISR Ends Here




//===========================================================/
//====================Offset Compensation====================/
//===========================================================/

interrupt void OffsetISR(void)
{
// Verifying the ISR
    IsrTicker++;

// DC offset measurement for ADC

    if (IsrTicker>=5000)
      {
            offsetA= _IQmpy(K1,offsetA)+_IQmpy(K2,_IQ12toIQ(AdcResult.ADCRESULT1));
            //Phase A offset
            offsetB= _IQmpy(K1,offsetB)+_IQmpy(K2,_IQ12toIQ(AdcResult.ADCRESULT2));
                  //Phase B offset
            offsetC= _IQmpy(K1,offsetC)+_IQmpy(K2,_IQ12toIQ(AdcResult.ADCRESULT3));
                  //Phase C offset
      }

      if (IsrTicker > 20000)
      {
            EALLOW;
            PieVectTable.ADCINT1=&MainISR;
            EDIS;
      }

// Enable more interrupts from this timer
      AdcRegs.ADCINTFLG.bit.ADCINT1=1;

// Acknowledge interrupt to recieve more interrupts from PIE group 1
      PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

}//End of offset compensation


//===========================================================/
//===============Protection Configuration====================/
//===========================================================/


void HVDMC_Protection(void)
{

// Configure Trip Mechanism for the Motor control software
// -Cycle by cycle trip on CPU halt
// -One shot IPM trip zone trip
// These trips need to be repeated for EPWM1 ,2 & 3
```

```
//=========================================================================
//Motor Control Trip Config, EPwm1,2,3
//=========================================================================
      EALLOW;
// CPU Halt Trip
      EPwm1Regs.TZSEL.bit.CBC6=0x1;
      EPwm2Regs.TZSEL.bit.CBC6=0x1;
      EPwm3Regs.TZSEL.bit.CBC6=0x1;


      EPwm1Regs.TZSEL.bit.OSHT1   = 1;  //enable TZ1 for OSHT
      EPwm2Regs.TZSEL.bit.OSHT1   = 1;  //enable TZ1 for OSHT
      EPwm3Regs.TZSEL.bit.OSHT1   = 1;  //enable TZ1 for OSHT

// What do we want the OST/CBC events to do?
// TZA events can force EPWMxA
// TZB events can force EPWMxB

      EPwm1Regs.TZCTL.bit.TZA = TZ_FORCE_LO; // EPWMxA will go low
      EPwm1Regs.TZCTL.bit.TZB = TZ_FORCE_LO; // EPWMxB will go low

      EPwm2Regs.TZCTL.bit.TZA = TZ_FORCE_LO; // EPWMxA will go low
      EPwm2Regs.TZCTL.bit.TZB = TZ_FORCE_LO; // EPWMxB will go low

      EPwm3Regs.TZCTL.bit.TZA = TZ_FORCE_LO; // EPWMxA will go low
      EPwm3Regs.TZCTL.bit.TZB = TZ_FORCE_LO; // EPWMxB will go low

      EDIS;

     // Clear any spurious OV trip
      EPwm1Regs.TZCLR.bit.OST = 1;
      EPwm2Regs.TZCLR.bit.OST = 1;
      EPwm3Regs.TZCLR.bit.OST = 1;

}//End of protection configuration

//=========================================================================
// No more.
//=========================================================================
```