AN908

Using the dsPIC30F for Vector Control of an ACIM

Author:	*Dave Ross, John Theys*
	Diversified Engineering Inc.
Co-Author:	*Steve Bowling*
	Microchip Technology Inc.

INTRODUCTION

This application note describes a vector control application that is written for the dsPIC30F family of devices. Except for a brief discussion on control theory, the information presented assumes you have a basic understanding of AC Induction Motor (ACIM) characteristics. References are included in some instances to provide background information.

SOFTWARE FEATURES

The Vector Control software has the following features:

- The software implements vector control of an AC induction motor using the indirect flux control method.
- With a 50 μsec control loop period, the software requires approximately 9 MIPS of CPU overhead (less than 1/3 of the total available CPU).
- The application requires 258 bytes of data memory storage and 256 bytes of constant storage. With the user interface, approximately 8 Kbytes of program memory are required.
- The memory requirements of the application allow it to be run on the dsPIC30F2010, which is the smallest and least expensive dsPIC30F device at the time of this writing.
- An optional diagnostics mode can be enabled to allow real-time observation of internal program variables on an oscilloscope. This feature facilitates control loop adjustment.

VECTOR CONTROL THEORY

Background

The AC induction motor is the workhorse of industrial and residential motor applications due to its simple construction and durability. These motors have no brushes to wear out or magnets to add to the cost. The rotor assembly is a simple steel cage.

ACIM's are designed to operate at a constant input voltage and frequency, but you can effectively control an ACIM in an open loop variable speed application if the frequency of the motor input voltage is varied. If the motor is not mechanically overloaded, the motor will operate at a speed that is roughly proportional to the input frequency. As you decrease the frequency of the drive voltage, you also need to decrease the amplitude by a proportional amount. Otherwise, the motor will consume excessive current at low input frequencies. This control method is called “Volts-Hertz control”.

In practice, a custom Volts-Hertz profile is developed that ensures the motor operates correctly at any speed setting. This profile can take the form of a look-up table or can be calculated during run time. Often, a slope variable is used in the application that defines a linear relationship between drive frequency and voltage at any operating point. The Volts-Hertz control method can be used in conjunction with speed and current sensors to operate the motor in a closed loop fashion.

The Volts-Hertz method works very well for slowly changing loads such as fans or pumps. But, it is less effective when fast dynamic response is required. In particular, high current transients can occur during rapid speed or torque changes. The high currents are a result of the high slip factor that occurs during the change. Fast dynamic response can be realized without these high currents if both the torque and flux of the motor are controlled in a closed loop manner. This is accomplished using Vector Control techniques. Vector control is also commonly referred to as Field Oriented Control (FOC).

The benefits of vector control can be directly realized as lower energy consumption. This provides higher efficiency, lower operating costs and reduces the cost of drive components.

Vector Control

Traditional control methods, such as the Volts-Hertz control method described above, control the frequency and amplitude of the motor drive voltage. In contrast, vector control methods control the frequency, amplitude and phase of the motor drive voltage. The key to vector control is to generate a 3-phase voltage as a phasor to control the 3-phase stator current as a phasor that controls the rotor flux vector and finally the rotor current phasor.

Ultimately, the components of the rotor current need to be controlled. The rotor current cannot be measured because the rotor is a steel cage and there are no direct electrical connections. Since the rotor currents cannot be measured directly, the application program calculates these parameters indirectly using parameters that can be directly measured.

The technique described in this application note is called indirect vector control because there is no direct access to the rotor currents. Indirect vector control of the rotor currents is accomplished using the following data:

- Instantaneous stator phase currents, i_a, i_b and i_c
- Rotor mechanical velocity
- Rotor electrical time constant

The motor must be equipped with sensors to monitor the 3-phase stator currents and a rotor velocity feedback device.

A MATTER OF PERSPECTIVE...

The key to understanding how vector control works is to form a mental picture of the coordinate reference transformation process. If you picture how an AC induction motor works, you might imagine the operation from the perspective of the stator. From this perspective, a sinusoidal input current is applied to the stator. This time variant signal causes a rotating magnetic flux to be generated. The speed of the rotor is going to be a function of the rotating flux vector. From a stationary perspective, the stator currents and the rotating flux vector look like AC quantities.

Now, instead of the previous perspective, imagine that you could climb inside the motor. Once you are inside the motor, picture yourself running alongside the spinning rotor at the same speed as the rotating flux vector that is generated by the stator currents. Looking at the motor from this perspective during steady state conditions, the stator currents look like constant values, and the rotating flux vector is stationary! Ultimately, you want to control the stator currents to get the desired rotor currents (which cannot be measured directly). With the coordinate transformation, the stator currents can be controlled like DC values using standard control loops.

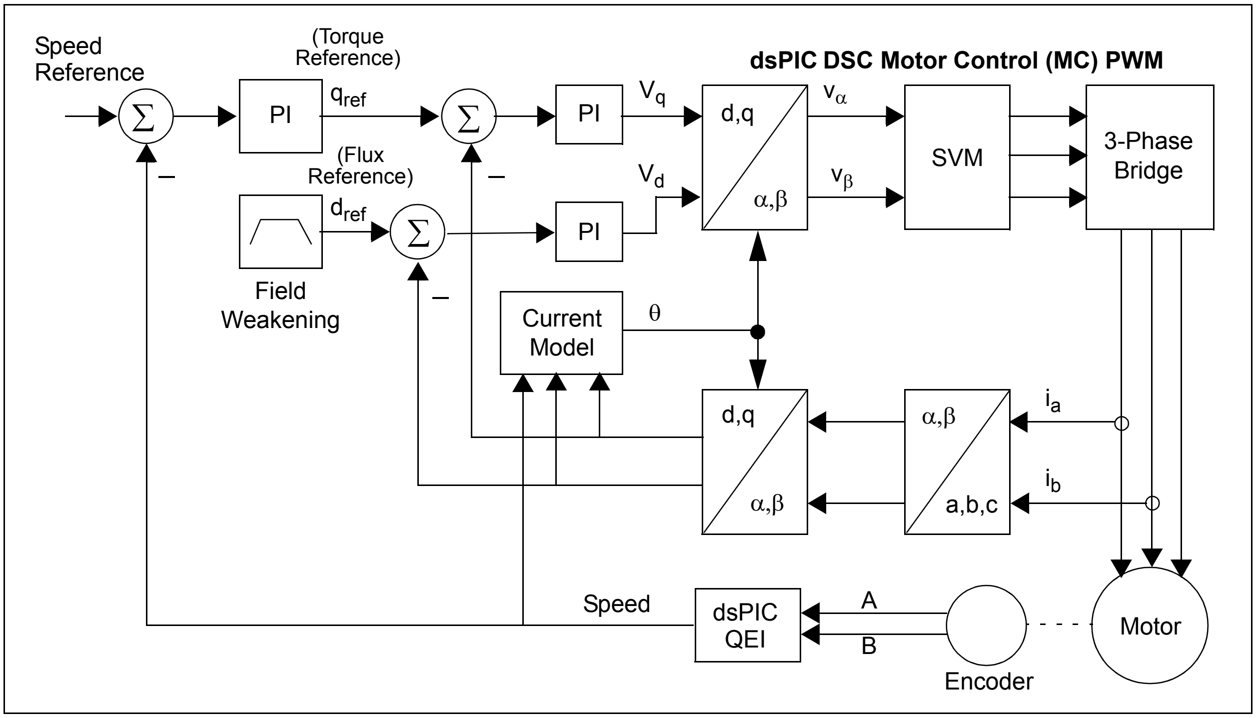
VECTOR CONTROL SUMMARY

To summarize the steps required for indirect vector control:

1. The 3-phase stator currents are measured. This measurement provides i_a, i_b and i_c. The rotor velocity is also measured.
2. The 3-phase currents are converted to a 2-axis system. This conversion provides the variables i_α and i_β from the measured i_a, i_b and i_c values. i_α and i_β are time varying quadrature current values as viewed from the perspective of the stator.
3. The 2-axis coordinate system is rotated to align with the rotor flux using a transformation angle information calculated at the last iteration of the control loop. This conversion provides the I_d and I_q variables from i_α and i_β. I_d and I_q are the quadrature currents transformed to the rotating coordinate system. For steady state conditions, I_d and I_q will be constant.
4. Error signals are formed using I_d, I_q and reference values for each. The I_d reference controls rotor magnetizing flux. The I_q reference controls the torque output of the motor. The error signals are input to PI controllers. The output of the controllers provide V_d and V_q, which is a voltage vector that will be sent to the motor.
5. A new coordinate transformation angle is calculated. The motor speed, rotor electrical time constant, I_d and I_q are the inputs to this calculation. The new angle tells the algorithm where to place the next voltage vector to produce an amount of slip for the present operating conditions.
6. The V_d and V_q output values from the PI controllers are rotated back to the stationary reference frame using the new angle. This calculation provides quadrature voltage values v_α and v_β.
7. The v_α and v_β values are transformed back to 3-phase values v_a, v_b and v_c. The 3-phase voltage values are used to calculate new PWM duty cycle values that generate the desired voltage vector.

The entire process of transforming, PI iteration, transforming back and generating PWM is illustrated in Figure 1.

FIGURE 1: VECTOR CONTROL BLOCK DIAGRAM



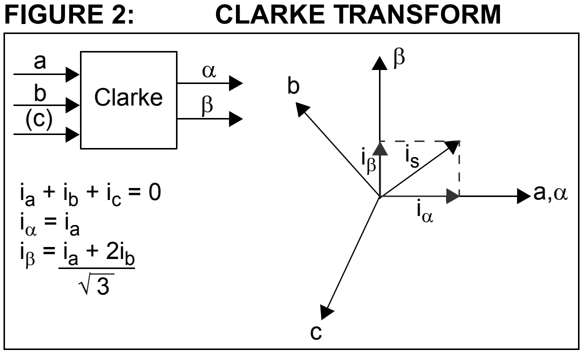
Coordinate Transforms

Through a series of coordinate transforms the time invariant values of torque and flux can be indirectly determined and controlled with classic PI control loops. The process starts out by measuring the three phase motor currents. In practice, you can take advantage of the constraint that in a 3-phase system the instantaneous sum of the three current values will be zero. Thus, by measuring only two of the three currents you can know the third. The cost of the hardware is reduced because only two current sensors are required.

CLARKE TRANSFORM

The first transform is to move from a 3-axis, 2-dimensional coordinate system referenced to the stator of the motor to a 2-axis system also referenced to the stator. This process is called the Clarke Transform, as illustrated in Figure 2.

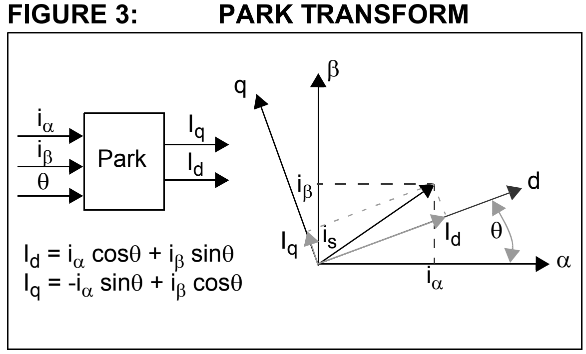
FIGURE 2: CLARKE TRANSFORM



PARK TRANSFORM

At this point you have the stator current Phasor represented on a 2-axis orthogonal system with the axis called α-β. The next step is to transform into another 2-axis system that is rotating with the rotor flux. This transformation uses the Park Transform, as illustrated in Figure 3. This 2-axis rotating coordinate system is called the d-q axis.

FIGURE 3: PARK TRANSFORM



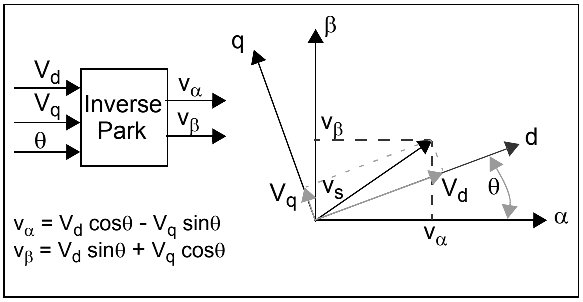
From this perspective the components of the current Phasor in the d-q coordinate system are time invariant. Under steady state conditions they are DC values.

The stator current component along the d axis is proportional to the flux, and the component along the q axis is proportional to the rotor torque. Now that you have these components represented as DC values you can control them independently with classic PI control loops.

INVERSE PARK

After the PI iteration, you have two voltage component vectors in the rotating d-q axis. You will need to go through complementary inverse transforms to get back to the 3-phase motor voltage. First you transform from the 2-axis rotating d-q frame to the 2-axis stationary frame α-β. This transformation uses the Inverse Park Transform, as illustrated in Figure 4.

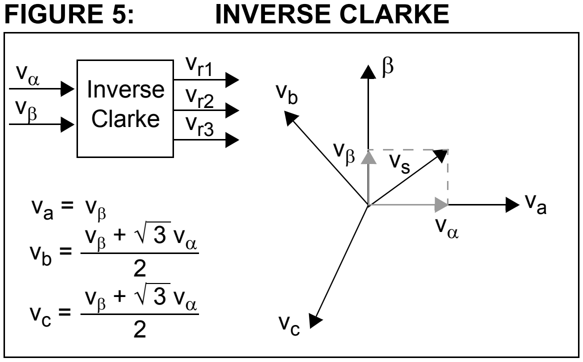
FIGURE 4: INVERSE PARK



INVERSE CLARKE

The next step is to transform from the stationary 2-axis α-β frame to the stationary 3-axis, 3-phase reference frame of the stator. Mathematically, this transformation is accomplished with the Inverse Clarke Transform, as illustrated in Figure 5.

FIGURE 5: INVERSE CLARKE



Flux Estimator

In an asynchronous squirrel cage induction motor the mechanical speed of the rotor is slightly less than the rotating flux field. The difference in angular speed is called slip and is represented as a fraction of the rotating flux speed. For example, if the rotor speed and the flux speed are the same the slip is 0 and if the rotor speed is 0 the slip is 1.

You probably have noticed that the Park and Inverse Transforms require an input angle θ. The variable θ represents the angular position of the rotor flux vector. The correct angular position of the rotor flux vector must be estimated based on known values and motor parameters. This estimation uses a motor equivalent circuit model. The slip required to operate the motor is accounted for in the flux estimator equations and is included in the calculated angle.

The flux estimator calculates a new flux position based on stator currents, the rotor velocity and the rotor electrical time constant. This implementation of the flux estimation is based on the motor current model and in particular these three equations:

EQUATION 1: MAGNETIZING CURRENT

$$I_{mr} = I_{mr} + \frac{T}{T_r}(I_d - I_{mr})$$

EQUATION 2: FLUX SPEED

$$f_s = (P_{pr} \cdot n) + \left(\frac{1}{T_r \omega_b} \cdot \frac{I_q}{I_{mr}}\right)$$

EQUATION 3: FLUX ANGLE

$$\theta = \theta + \omega_b \cdot f_s \cdot T$$

where:

- I_{mr} = Magnetizing current (as calculated from measured values)
- f_s = Flux speed (as calculated from measured values)
- T = Sample (loop) time (parameter in program)
- n = Rotor speed (measured with the shaft encoder)
- T_r = L_r/R_r = Rotor time constant (must be obtained from the motor manufacturer)
- θ = Rotor flux position (output variable from this module)
- ω_b = Electrical nominal flux speed (from motor name plate)
- P_{pr} = Number of pole pairs (from motor name plate)

During steady state conditions, the I_d current component is responsible for generating the rotor flux. For transient changes, there is a low-pass filtered relationship between the measured I_d current component and the rotor flux. The magnetizing current, I_{mr}, is the component of I_d that is responsible for producing the rotor flux. Under steady-state conditions, I_d is equal to I_{mr}. Equation 1 relates I_d and I_{mr}. This equation is dependent upon accurate knowledge of the rotor electrical time constant. Essentially, Equation 1 corrects the flux producing component of I_d during transient changes.

The computed I_{mr} value is then used to compute the slip frequency, as shown in Equation 2. The slip frequency is a function of the rotor electrical time constant, I_q, I_{mr} and the current rotor velocity.

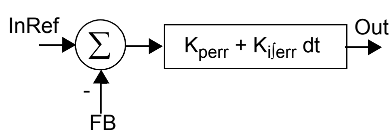
Equation 3 is the final equation of the flux estimator. It calculates the new flux angle based on the slip frequency calculated in Equation 2 and the previously calculated flux angle.

If the slip frequency and stator currents have been related by Equation 1 and Equation 2, then motor flux and torque have been specified. Furthermore, these two equations ensure that the stator currents are properly oriented to the rotor flux. If proper orientation of the stator currents and rotor flux is maintained, then flux and torque can be controlled independently. The I_d current component controls rotor flux and the I_q current component controls motor torque. This is the key principle of indirect vector control.

PI Control

Three PI loops are used to control three interactive variables independently. The rotor speed, rotor flux and rotor torque are each controlled by a separate PI module. The implementation is conventional and includes a term (Kc*Excess) to limit integral windup, as illustrated in Figure 6.

FIGURE 6: PI CONTROL




```
Err = InRef - FB;
U = Sum + Kp*Err;
If (U > Outmax);
        Out = Outmax;
else if (U < Outmin)
        Out = Outmin;
else
        Out = U;
Excess = U - Out;
Sum = Sum + (Ki*Err)-(Kc*Excess);
```

PID CONTROLLER BACKGROUND

A complete discussion of Proportional Integral Derivative (PID) controllers are beyond the scope of this application note, but this section will provide you with the basics of PID operation.

A PID controller responds to an error signal in a closed control loop and attempts to adjust the controlled quantity to achieve the desired system response. The controlled parameter can be any measurable system quantity such as speed, torque or flux. The benefit of the PID controller is that it can be adjusted empirically by adjusting one or more gain values and observing the change in system response.

A digital PID controller is executed at a periodic sampling interval. It is assumed that the controller is executed frequently enough so that the system can be properly controlled. The error signal is formed by subtracting the desired setting of the parameter to be controlled from the actual measured value of that parameter. The sign of the error indicates the direction of change required by the control input.

The Proportional (P) term of the controller is formed by multiplying the error signal by a P gain, causing the PID controller to produce a control response that is a function of the error magnitude. As the error signal becomes larger, the P term of the controller becomes larger to provide more correction.

The effect of the P term tends to reduce the overall error as time elapses. However, the effect of the P term reduces as the error approaches zero. In most systems, the error of the controlled parameter gets very close to zero but does not converge. The result is a small remaining steady state error.

The Integral (I) term of the controller is used to eliminate small steady state errors. The I term calculates a continuous running total of the error signal. Therefore, a small steady state error accumulates into a large error value over time. This accumulated error signal is multiplied by an I gain factor and becomes the I output term of the PID controller.

The Differential (D) term of the PID controller is used to enhance the speed of the controller and responds to the rate of change of the error signal. The D term input is calculated by subtracting the present error value from a prior value. This delta error value is multiplied by a D gain factor that becomes the D output term of the PID controller. The D term of the controller produces more control output the faster the system error is changing.

Not all PID controllers will implement the D or, less commonly, the I terms. For example, this application does not use D terms due to the relatively slow response time of motor speed changes. In this case, the D term could cause excessive changes in PWM duty cycle that could affect the operation of the algorithms and produce over current trips.

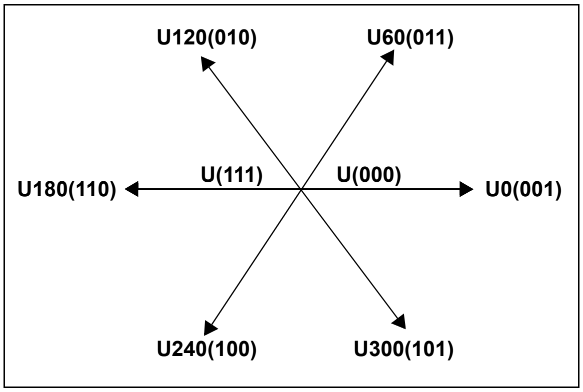
Space Vector Modulation

The final step in the vector control process is to generate pulse-width modulation signals for the 3-phase motor voltage signals. By using Space Vector Modulation (SVM) techniques the process of generating the pulse-width for each of the 3 phases reduces to a few simple equations. In this implementation the Inverse Clarke Transform has been folded into the SVM routine, which further simplifies the calculations.

Each of the three inverter outputs can be in one of two states. The inverter output can be either connected to the + bus rail or the - bus rail, which allows for 2^3=8 possible states that the output can be in (see Table 1).

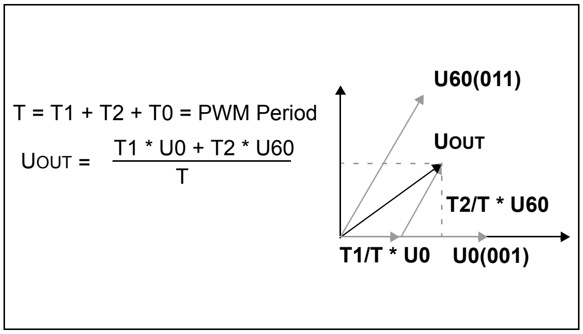
The two states where all three outputs are connected to either the + bus or the - bus are considered null states because there is no line-to-line voltage across any of the phases. These are plotted at the origin of the SVM Star. The remaining six states are represented as vectors with 60 degree rotation between each state, as shown in Figure 7.

FIGURE 7: SPACE VECTOR MODULATION



The process of Space Vector Modulation allows the representation of any resultant vector by the sum of the components of the two adjacent vectors. In Figure 8, UOUT is the desired resultant. It lies in the sector between U60 and U0. If during a given PWM period T U0 is output for T1/T and U60 is output for T2/T, the average for the period will be UOUT.

FIGURE 8: AVERAGE SPACE VECTOR MODULATION



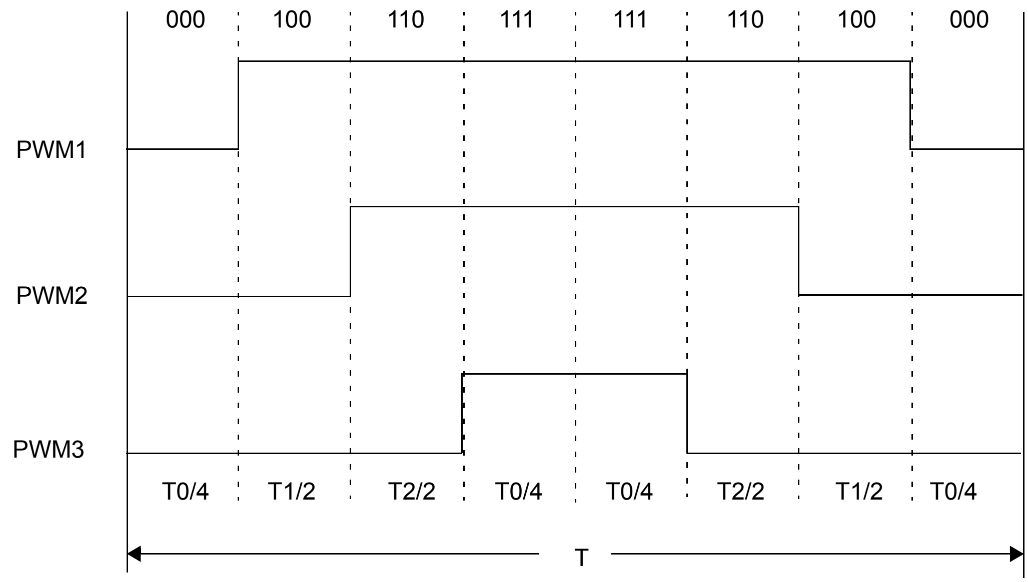
The values for T1 and T2 can be extracted with no extra calculations by using a modified Inverse Clarke transformation. By reversing v_α and v_β, a reference axis is generated that is shifted by 30 degrees from the SVM Star. As a result, for each of the six segments one axis is exactly opposite to that segment and the other two axis symmetrically bound the segment. The values of the vector components along those two bounding axis are equal to T1 and T2. See the `CalcRef.s` and `SVGen.s` files in **"Appendix B. Source Code"** for details of the calculations.

You can see from Figure 9 that for the PWM period T, the vector T1 is output for T1/T and the vector T2 is output for T2/T. During the remaining time the null vectors are output. The dsPIC® DSC device is configured for center aligned PWM, which forces symmetry about the center of the period. This configuration produces two pulses line-to-line during each period. The effective switching frequency is doubled, reducing the ripple current while not increasing the switching losses in the power devices.

TABLE 1: SPACE VECTOR MODULATION INVERTER STATES

C	B	A	V_{ab}	V_{bc}	V_{ca}	V_{ds}	V_{qs}	Vector
0	0	0	0	0	0	0	0	U(000)
0	0	1	VDC	0	-VDC	2/3VDC	0	U_0
0	1	1	0	VDC	-VDC	VDC/3	VDC/3	U_{60}
0	1	0	-VDC	VDC	0	-VDC/3	VDC/3	U_{120}
1	1	0	-VDC	0	VDC	-2VDC/3	0	U_{180}
1	0	0	0	-VDC	VDC	-VDC/3	- VDC/3	U_{240}
1	0	1	VDC	-VDC	0	VDC/3	- VDC/3	U_{300}
1	1	1	0	0	0	0	0	U(111)

FIGURE 9: PWM FOR PERIOD T



CODE DESCRIPTION

The vector control source code was developed in MPLAB® using the Microchip MPLAB C30 tool suite. The main application is written in C and all the primary vector control functions are written in assembly and optimized for speed of execution.

Conventions

A description of the functions is contained in the header of each source file. The equivalent C code for the function is also included in the header for reference. The C lines of code are used as comments in the optimized assembly code so that code flow can easily be followed.

At the beginning of each function the pertinent variables are moved to specific working (W) registers that are used by the DSP and math instructions. The variables are moved back to their respective register locations at the end of the code function. Most of these variables are grouped into structures of related parameters to provide efficient access from the C or assembly code.

Each W register used in an assembly module has been assigned a descriptive name that tells what value the register holds during the calculation. The re-naming of the W registers makes the code easier to follow and avoids register usage conflicts.

Variable Definition and Scaling

Most variables are stored in 1.15 fractional format, which is one of the inherent math modes in the dsPIC DSC devices. A signed fixed-point integer is represented as follows:

- MSB is the sign bit
- range -1 to +.9999
- 0x8000 = -1
- 0000 = 0
- 0x7FFF = .9999

All values are normalized using the Per Unit system (PU).

$V_{PU} = V_{ACT}/V_B$

Then scaled so that the base quantity = .125

This allows for values of 8 times the base value.

V_B = 230V, V_{ACT} =120V, V_{PU} = 120/230 =.5PU,

Scaling $\rightarrow$ V_B = .125 = 0x0FFF (1.15)

120V = .5 * .125 = 0x07FF (1.15)

Individual Source File Descriptions

This section describes the functions contained in each source file.

Note:	If you are viewing an electronic version of this application note, you can click on the following file names to navigate to the code in **"Appendix B. Source Code"**.

UserParms.h

All user definable parameters are located in the `UserParms.h` file. These parameters include motor data and control loop tuning values. More information on the parameters is provided in the Software Tuning section of this document.

ACIM.c

The `ACIM.c` file is the primary source code file for the application. This file contains the main software loop and all ISR handlers. This file calls all hardware and variable initialization routines.

To accomplish high performance closed-loop control the entire vector control loop must be executed every PWM cycle. This is done in the ISR for the ADC converter. The PWM time base is used to trigger ADC conversions. When the ADC conversion is complete, an interrupt is generated.

When not in the ISR, a main software loop is run that handles the user interface. A software count variable is maintained in the ISR so that the user interface is run at periodic intervals. As written, the user interface code is scheduled to run every 50 milliseconds. This parameter can be changed by modifying the `UserParms.h` file.

A software diagnostics mode can be enabled by uncommenting the `#define DIAGNOSTICS` statement in the `UserParms.h` file. The diagnostics mode enables output compare channels OC7 and OC8 as PWM outputs. These outputs can be filtered using simple RC filters and used like a D/A converter to observe the time history of software variables. The diagnostics output simplifies tuning of the PI control loops. More information on the diagnostics output is provided in the Software Tuning section of this document.

Encoder.c

This file contains the function `InitEncoderScaling()` Which is used to calculate the scaling values for mechanical angle and mechanical speed measured with the optical encoder.

InitCurModel.c

This file contains the `InitCurModScaling()` function, which is called from the setup routines in the `ACIM.c` file. This function is used to calculate fixed-point scaling factors that are used in the current model equations from floating point values. The current model scaling factors are a function of the rotor time constant, vector calculation loop period, number of motor poles and the maximum motor velocity in revolutions per second.

CalcRef.s

This file contains the `CalcRefVec()` function, which calculates the scaled 3-phase voltage output vector, (V_{r1}, V_{r2} and V_{r3}), from v_α and v_β. The function implements the Inverse Clarke function, which translates the voltage vector components from a 2-coordinate system back to a 3-coordinate system that can be used by the 3-phase PWM. The method is a modified Inverse Clarke transform where v_α and v_β are swapped compared to the normal Inverse Clarke. The modified method must be used to produce the proper phase alignment of the voltage vector.

CalcVel.s

This file has three functions, `InitCalcVel()`, `CalcVelIrp()` and `CalcVel()`, which are used to determine the motor velocity. The `InitCalcVel()` function initializes key variables associated with the velocity calculations.

The `CalcVelIrp()` function is called at each vector control interrupt period. The interrupt interval, `VelPeriod`, must be less than the minimum time required for 1/2 revolution at maximum speed.

This routine accumulates the change for a specified number of interrupt periods, then copies the accumulation value to the `iDeltaCnt` variable for use by the `CalcVel()` routine to calculate velocity. The accumulation is set back to zero and a new accumulation starts.

The `CalcVel()` routine is only called when new velocity information is available. For the default software values, the `CalcVel()` routine is called every 30 interrupt periods. This interval gives new velocity information every 1.5 msec for a 50 usec interrupt period. The velocity control loop is run each time new velocity information is obtained.

ClarkePark.s

This file contains the function `ClarkePark()` and calculates Clarke and Park transforms. The function uses the sine and cosine values of the flux position angle to calculate the quadrature current values of I_d and I_q. This routine works the same for both integer scaling and 1.15 scaling.

CurModel.s

This file contains the `CurModel()` and `InitCurModel()` functions. The `CurModel()` function executes the rotor current model equation to determine a new rotor flux angle as a function of the rotor velocity and the transformed stator current components. The `InitCurModel()` function is used to clear variables associated with the `CurModel()` routine.

FIGURE 10: VECTOR CONTROL INTERRUPT SERVICE ROUTINE

```
void __attribute__((__interrupt__)) _ADCInterrupt(void)
{
IFS0bits.ADIF = 0;

// Increment count variable that controls execution
// of display and button functions.
iDispLoopCnt++;

// acumulate encoder counts since last interrupt
CalcVelIrp();

if( uGF.bit.RunMotor )
      {
      // Set LED1 for diagnostics
      pinLED1 = 1;
      // Calculate velocity from accumulated encoder counts
      CalcVel();
      // Calculate qIa,qIb
      MeasCompCurr();
      // Calculate qId,qIq from qSin,qCos,qIa,qIb
      ClarkePark();
      // Calculate PI control loop values
      DoControl();
      // Calculate qSin,qCos from qAngle
      SinCos();
      // Calculate qValpha, qVbeta from qSin,qCos,qVd,qVq
      InvPark();
      // Calculate Vr1,Vr2,Vr3 from qValpha, qVbeta
      CalcRefVec();
      // Calculate and set PWM duty cycles from Vr1,Vr2,Vr3
      CalcSVGen();
      // Clear LED1 for diagnostics
      pinLED1 = 0;
      }
}
```

FdWeak.s

The `FdWeak.s` file contains the function for field weakening. The application code, as provided, does not implement field weakening. Field weakening allows a motor to be run at higher than the rated speed. At these higher speeds, the voltage delivered to the motor is kept constant while the frequency is increased.

A field weakening constant is defined in the `UserParms.h` file. This value is derived from the V/Hz constant of the motor. The motor that was used to develop this application has a working voltage of 230 VAC and is designed for an input frequency of 60 Hz. Based on these values, the V/Hz constant is 230/60 = 3.83. The value of 3750 defined for the field weakening constant in `UserParms.h` was empirically derived based on the V/Hz constant of the motor and the absolute scaling of A/D feedback values for the application.

When the motor operates within its rated speed and voltage range, the reference for the I_d control loop is held constant. The field weakening constant in `UserParms.h` is used as the reference value for the control loop. In the normal operating range of the motor, the rotor flux is kept constant.

If field weakening is implemented, the I_d control loop reference should be reduced linearly when the motor is said to 'run out of voltage'. The motor 'runs out of voltage' when the V/Hz ratio for the motor can not be maintained. For example, assume that you are driving a 230 VAC motor with a 115 VAC power source. Since the motor is designed to run at 230 VAC and 60 Hz, the motor would 'run out of voltage' at 30 Hz when operating from a 115 VAC supply. Above 30 Hz, the I_d control loop reference should be linearly reduced as a function of frequency.

You can determine the drive frequency where your ACIM application will run out of voltage by monitoring the inverter DC bus voltage.

When operating in a region where field weakening would be required, the I_d and I_q control loops will saturate, which effectively limits the motor flux. The use of field weakening allows the vector control algorithm to limit its output without saturating the control loops. This is one of the key benefits of field weakening. The operating range of the motor can be extended while closed loop control is maintained.

You can experiment with field weakening in this application by changing the defined reference value in `UserParms.h` file. By lowering this value, you can limit the available voltage that can be delivered to the motor.

InvPark.s

This file contains the `InvPark()` function, which processes the voltage vector values, V_d and V_q, which are generated by the inner PI current control loops. The `InvPark()` function 'un-rotates' the voltage vector values to align them with the stationary reference frame. The function produces the v_α and v_β values. The rotation is accomplished using sine and cosine values of the new rotor flux angle that was previously calculated in the rotor current model equations.

This routine works the same for both integer scaling and 1.15 scaling.

MeasCur.s

This file has two functions, `MeasCompCurr()` and `InitMeasCompCurr()`. The `MeasCompCurr()` function reads S/H channels CH1 and CH2 of the ADC, scales them as signed fractional values using `qKa`, `qKb` and put the results `qIa` and `qIb` of `ParkParm`. A running average of the A/D offset is maintained and is subtracted from the ADC value before scaling.

The `InitMeasCompCurr()` function is used to initialize the A/D offset values at startup.

Scaling and offset variables associated with these functions are kept in the `MeasCurrParm` data structure, which is declared in the `MeasCur.s` file.

OpenLoop.s

This file contains the `OpenLoop()` function that calculates a new rotor flux angle when the application is running open loop. The function calculates the change in rotor flux angle for the desired operating speed. The change in rotor flux angle is then added to the old angle to set the new angle of the voltage vector.

PI.s

This file contains the `CalcPI()` function, which executes a PI controller. The `CalcPI()` function accepts a pointer to a structure that contains the PI coefficients, input and reference signals, output limits and the PI controller output value.

ReadADC0.s

This file contains the `ReadADC0()` and `ReadSignedADC0()` functions. These functions read the data obtained from sample/hold Channel 0 of the ADC, scale the value and store the results.

The `ReadSignedADC0()` function is currently used to read a reference speed value from the potentiometer on the demo board. If speed is obtained from another source, these functions are not required for the application.

SVGen.s

This file has the `CalcSVGen()` function, which calculates the final PWM values as a function of the 3-phase voltage vector.

Trig.s

This file contains the `SinCos()` function, which calculates sine and cosine for a specified angle using linear interpolation on a table of 128 words.

To save data memory space, the 128-word sine wave table is placed in program memory and accessed using the Program Space Visibility (PSV) feature of the dsPIC DSC architecture. PSV allows a portion of program memory to be mapped into data memory space so that constant data can be accessed as if it was in RAM.

This routine works the same for both integer scaling and 1.15 scaling. For integer scaling the angle is scaled such that $0 \le$ angle $< 2\Pi$ corresponds to $0 \le$ angle < 0xFFFF. The resulting Sine and Cosine values are returned, scaled to -32769 to +32767 (i.e., 0x8000 to 0x7FFF).

For 1.15 scaling, the angle is scaled such that $-\Pi \le$ angle $< \Pi$ corresponds to -1 to +0.9999 (i.e., 0x8000 $\le$ angle < 0x7FFF). The resulting sine and cosine values are returned scaled to -1 to +0.9999 (i.e., 0x8000 to 0x7FFF).

DEMO HARDWARE

The vector control application can be run on the dsPICDEM™ MC1 Motor Control Development System. You will need the following hardware:

- Microchip dsPICDEM MC1 Motor Control Development Board
- 9 VDC power supply
- Microchip dsPICDEM MC1H 3-Phase High Voltage Power Module
- Power supply cable for the power module
- 3-Phase AC induction motor with shaft encoder

Note:	An encoder of at least 250 lines per revolution should be used. The upper limit would be 32,768 lines per revolution.

Recommended Motor and Encoder

The following motor and encoder combination was used to develop this application and select the software tuning parameters:

- Leeson Cat# 102684 motor, 1/3 HP, 3450 RPM
- U.S. Digital encoder, model E3-500-500-IHT

The Leeson motor can be obtained from Microchip or an electric motor distributor. The encoder can be ordered from the U.S. Digital web site, www.usdigital.com. This model of encoder is shipped with a mounting alignment kit and a self-sticking encoder body. The encoder can be mounted directly on the front face of the motor, as shown in Figure 12. Any other similar encoder with 500 lines of resolution may be used instead of the U.S. Digital device, if desired.

FIGURE 11: HARDWARE SETUP USING dsPICDEM MOTOR CONTROL DEVELOPMENT SYSTEM

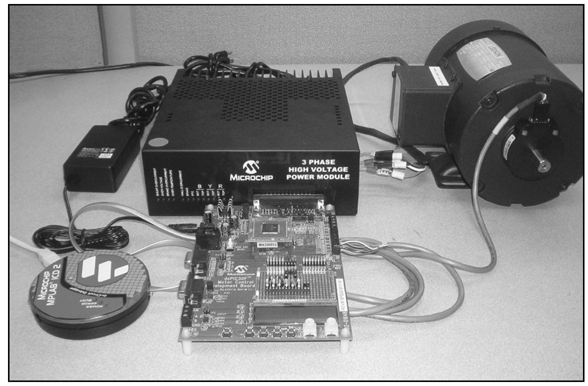
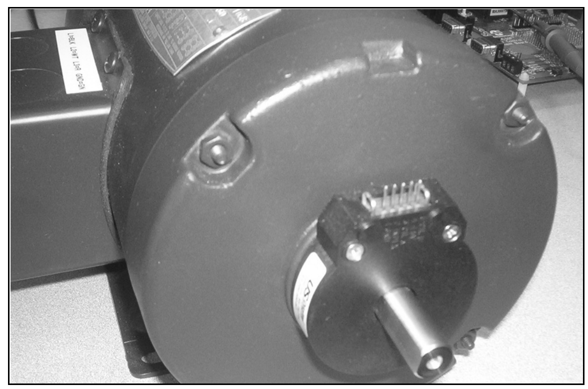


FIGURE 12: LEESON MOTOR WITH MOUNTED INCREMENTAL ENCODER



If You Select Another Motor...

If another motor is selected, you will likely have to experiment with the control loop tuning parameters to get good response from the control algorithm. At a minimum, you will need to determine the rotor electrical time constant in seconds. This information can be obtained from the motor manufacturer. The application will run without the proper rotor time constant, but the response of the system to transient changes will not be ideal.

If the above referenced Leeson motor and a 500-line encoder are used, no adjustment of software tuning parameters should be necessary to get the demo running properly.

Phase Current Feedback

The vector control application requires knowledge of the 3-phase motor currents. This application is designed to use the isolated hall-effect current transducers found on the dsPICDEM MC1H power module. These transducers are active devices that provide a 200 KHz bandwidth, 0-5 volt feedback signal. The hall-effect devices have been used in this application for convenience and safety reasons. The signal from these devices can be connected directly to the dsPIC DSC A/D converter. For your end application, you can choose to measure currents using shunt resistors installed in each leg of the 3-phase inverter. The shunt resistors offer a less expensive solution for current measurement.

Motor Wiring Configuration

Most 3-phase ACIM's, including the Leeson motor, can be wired for 208V or 460V operation. If you are using the dsPICDEM MC1 system to drive your motor, you should wire the motor for 208V operation.

The vector control application does not regulate the DC bus voltage. However, a 208V motor will operate correctly from a 120V source with limited speed and torque output.

Jumper Placement

All jumpers on the 3-Phase High Voltage Power Module can be left at the default settings. If you have removed the cover of the power module to make modifications, please refer to the power module user's guide for the default jumper configuration.

The following jumper configuration should be used for the motor control development board.

- The isolated hall-effect current sensors are used to measure the motor phase currents. Ensure LK1 and LK2 (next to the 5V regulator) are placed on pins 1 and 2.
- Switch S2 (located next to the ICD connector) should be set to the 'Analog' position when running the demo code to connect the phase current feedback to the dsPIC DSC analog input pins. (S2 should be placed in the 'ICD' position for device programming).
- All other jumpers should be left in their default placements.

External Connections

- Plug the Motor Control Development Board directly into the 37 pin connector on the Power Module.
- Make sure a dsPIC30F6010 device is installed on the development board.
- Connect the motor leads to the output of the Power Module in the terminals labeled R,Y and B. Connect phase 1 to 'R', phase 2 to 'Y' and phase 3 to 'B'.
- Connect the encoder leads to the Quadrature Encoder Interface (QEI) terminal block on the MCDB. Match up the pin names screened on the MCDB with the signal names on the encoder. Finally connect the 9V power supply to J2 on the MCDB.

Port Usage

Table 2 indicates how the dsPIC DSC device ports are used in this application. This information is provided to help you develop your hardware definition. The I/O pins that are required for the vector control application are shown in bold text. The application uses other pins, such as LCD interface lines, that are not required for the motor control function. These I/O connections may or may not be used in your final design.

TABLE 2: dsPIC DEVICE PORT USAGE SUMMARY

Pin	Functions	Type	Application Usage
Port A			
RA9	VREF-	O	LED1, D6 (Active-high)
RA10	VREF+	O	LED2, D7 (Active-high)
RA14	INT3	O	LED3, D8 (Active-high)
RA15	INT4	O	LED4, D9 (Active-high)
Port B			
RB0	**PGD/EMUD/AN0/CN2**	**AI**	**Phase1 Current/Device Programming Pin**
RB1	**PGC/EMUC/AN1/CN3**	**AI**	**Phase2 Current/Device Programming Pin**
RB2	AN2/SS1/LVDIN/CN4	AI	not used in application
RB3	**AN3/INDX/CN5**	**I**	**QEI Index**
RB4	**AN4/QEA/CN6**	**I**	**QEI A**
RB5	**AN5/QEB/CN7**	**I**	**QEI B**
RB6	AN6/OCFA	AI	not used in application
RB7	AN7	AI	Pot (VR1)
RB8	AN8	AI	not used in application
RB9	AN9	AI	not used in application
RB10	AN10	AI	not used in application
RB11	AN11	AI	not used in application
RB12	AN12	AI	not used in application
RB13	AN13	AI	not used in application
RB14	AN14	AI	not used in application
RB15	AN15/OCFB/CN12	O	not used in application
Port C			
RC1	T2CK	O	LCD R/$\overline{W}$
RC3	T4CK	O	LCD RS
RC13	EMUD1/SOSC2/CN1	—	Alternate ICD2 Communication Pin
RC14	EMUC1/SOSC1/T1CK/CN0	—	Alternate ICD2 Communication Pin
RC15	OSC2/CLKO	—	—
Port D			
RD0	EMUC2/OC1	I/O	LCD D0
RD1	EMUD2/OC2	I/O	LCD D1
RD2	OC3	I/O	LCD D2
RD3	OC4	I/O	LCD D3
RD4	OC5/CN13	O	not used in application
RD5	OC6/CN14	O	not used in application
RD6	OC7/CN15	O	PWM for diagnostics output
RD7	OC8/CN16/UPDN	O	PWM for diagnostics output
RD8	IC1	I	not used in application
RD9	IC2	I	not used in application
RD10	IC3	I	not used in application
RD11	IC4	O	Demo board PWM output buffer enable (Active-low)
RD12	IC5	—	not used in application
RD13	IC6/CN19	O	LCD E
RD14	IC7/CN20	—	not used in application

TABLE 2: dsPIC DEVICE PORT USAGE SUMMARY (CONTINUED)

Pin	Functions	Type	Application Usage
RD15	IC8/CN21	—	not used in application
Port E			
RE0	**PWM1L**	**O**	**Phase1 L**
RE1	**PWM1H**	**O**	**Phase1 H**
RE2	**PWM2L**	**O**	**Phase2 L**
RE3	**PWM2H**	**O**	**Phase2 H**
RE4	**PWM3L**	**O**	**Phase3 L**
RE5	**PWM3H**	**O**	**Phase3 H**
RE6	PWM4L	O	not used in application
RE7	PWM4H	O	not used in application
RE8	FLTA/INT1	I	Power Module Fault Signal (Active-low)
RE9	FLTB/INT2	O	Power Module Fault Reset (Active-high)
Port F			
RF0	C1RX	I	not used in application
RF1	C1TX	O	not used in application
RF2	U1RX	I	not used in application
RF3	U1TX	O	not used in application
RF4	U2RX/CN17	I	not used in application
RF5	U2TX/CN18	O	not used in application
RF6	EMUC3/SCK1/INT0	I	not used in application
RF7	SDI1	I	not used in application
RF8	EMUD3/SDO1	O	not used in application
Port G			
RG0	C2RX	O	not used in application
RG1	C2TX	O	not used in application
RG2	SCL	I/O	not used in application
RG3	SDA	I/O	not used in application
RG6	SCK2/CN8	I	Button 1 (S4) (Active-low)
RG7	SDI2/CN9	I	Button 2 (S5) (Active-low)
RG8	SDO2/CN10	I	Button 3 (S6) (Active-low)
RG9	SS2/CN11	I	Button 4 (S7) (Active-low)

PROJECT SETUP AND DEVICE PROGRAMMING

It is recommended that you use MPLAB IDE v6.50, or later, to create a project and program the device. To program the source code onto the dsPIC DSC device, you have two options:

1. You can import the pre-compiled hex file supplied with the application source code into MPLAB IDE and program the device, or
2. You can create a new project in MPLAB IDE, compile the source code and program the device.

Importing the HEX File

If you do not have the MPLAB C30 compiler installed, you will not be able to compile the application. In this case, just use the supplied hex file. You will need to use the same hardware setup described in the “Demo Hardware” section of this document.

Setting Up a New Project

The MPLAB C30 v. 1.20 compiler was used to build the application source code. To compile the source code, add all of the assembly files (`.s` extension) and C files to a new project. Include a device linker script in your project files. Assuming the C30 compiler was installed to the default location, use linker script file `p30f6010.gld` (this file is located in the `c:\pic30_tools\support\gld directory`). Also, set the assembler and C compiler include path for the build options.

These paths are `c:\pic30_tools\support\inc` and `c:\pic30_tools\support\h`.

Device Frequency

The supplied source code is set up to use a 7.37 MHz crystal and the 8X PLL option on the device oscillator, providing a device operating speed of 14.76 MIPS. If you have a different crystal value installed, you may need to change some of the values in the `UserParms.h` file. Refer to the “Software Tuning” section of this document for more information on the adjustment of values in `UserParms.h` file. Also, you will need to modify the `config.s` file if a different oscillator option is to be used.

SOFTWARE OPERATION

As provided, the demo program has basic features that allow you to evaluate the performance of the system in response to a 2:1 step change in requested speed.

Two modes of control are provided that allow full closed loop operation or operation in a conventional open loop constant Volts/Hertz mode.

The operational modes are controlled by four push buttons.

The speed command reference is obtained from potentiometer VR2, which is a bidirectional control where zero speed is in the center of the potentiometer.

Buttons

BUTTON 1 (S4)

Pressing Button 1 toggles the active state of the system. If it is off it will run, and if it is running it will stop. This button can also be used to clear any hardware faults by restarting the motor.

BUTTON 2 (S5)

Button 2 toggles the system between open-loop and closed-loop mode. By default, the system starts in open-loop mode.

BUTTON 3 (S6)

Button 3 toggles the commanded speed by a factor of 2. It powers up in the half speed mode.

BUTTON 4 (S7)

Button 4 does not have any function in the demo code, but the button processing code is provided so you can add your own functions.

LEDs

LED 1 (D6)

LED 1 is on when the system is running. This signal is modulated by the interrupt routine. The length of the interrupt service routine can be measured by looking at the time this signal is high.

LED 2 (D7)

On when system is in closed-loop mode.

LED 3 (D8)

On when speed is at full value, off when speed is at half value.

LED 4 (D9)

Not used in the application.

FDW/REV (D5)

The RD7 port pin that is connected to D5 is used as an output compare channel (OC8) for the diagnostics function. Therefore, D5 activity does not have any meaning in the application.

If the diagnostics output is not used, D5 can be driven directly from the QEI on the dsPIC DSC device. There is a control bit in the QEICON register that enables RD7 as a direction status output pin. With this feature enabled, D5 will be lid for the forward direction of travel.

LCD

The LCD is the primary means of user feedback. When the program is in the standby mode, the display prompts the user to push S4 to start the motor. When the program is running, the RPM is displayed. The LCD is updated in the main loop, and other display parameters can easily be added.

Troubleshooting

The motor will not run in open-loop mode:

- Check power module fault lights. Reset the dsPIC DSC device if necessary to clear faults.
- Check to make sure power module has power. Check bus voltage LED inside module.

The motor runs in open-loop mode, but will not run closed-loop.

- Ensure S2 is in ‘Analog’ position.
- Make sure LK1 and LK2 are configured properly.
- Check encoder wiring connections.
- There may be a reversal of encoder signals with respect to motor wiring and direction of rotation. If this is suspected, reverse the A and B signals on the encoder wiring connections. The encoder wiring will also depend on whether the encoder is mounted on the front or rear of the motor.

SOFTWARE TUNING

Diagnostics Mode

A diagnostics mode is available that allows you to use spare output compare (OC) channels OC7 and OC8 to observe internal program variables. These channels are used as PWM outputs for diagnostics. These PWM outputs can then be filtered using simple RC filter networks and used like simple DAC outputs to show the time history of internal variables on an oscilloscope.

The OC7 and OC8 channels are available on pins RD6 and RD7 of the dsPIC30F6010 device. These two pins are accessible on header J7 of the dsPICDEM MC1 Motor Control Development Board.

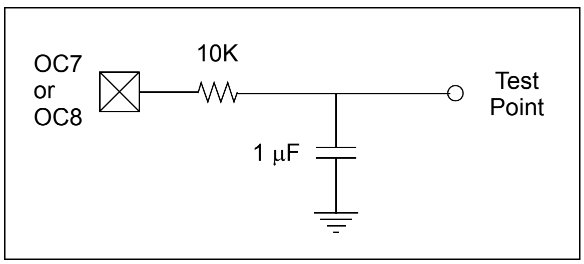
ENABLING DIAGNOSTICS MODE

To enable the diagnostics output, simply uncomment the `#define DIAGNOSTICS` statement in the `UserParms.h` file and re-compile the application.

HARDWARE SETUP FOR DIAGNOSTICS

You will need to add two RC low-pass filter networks to your development board to use the diagnostics. The RC filters should be connected to device pins RD6 and RD7. A 10 kohm resistor and a 1µF capacitor will work well for most situations. If you do not have the exact values, anything close to these values should work fine.

FIGURE 13: DIAGNOSTICS CIRCUIT



Adjusting the PID Gains

The P gain of a PID controller sets the overall system response. When first tuning a controller, the I and D gains should be set to zero. The P gain can then be increased until the system responds well to set-point changes without excessive overshoot or oscillations. Using lower values of P gain will 'loosely' control the system, while higher values will give 'tighter' control. At this point, the system will probably not converge to the set-point.

After a reasonable P gain is selected, the I gain can be slowly increased to force the system error to zero. Only a small amount of I gain is required in most systems. Note that the effect of the I gain, if large enough, can overcome the action of the P term, slow the overall control response and cause the system to oscillate around the set-point. If oscillation occurs, reducing the I gain and increasing the P gain will usually solve the problem.

This application includes a term to limit integral windup, which will occur if the integrated error saturates the output parameter. Any further increase in the integrated error will not effect the output. If allowed to accumulate, when the error does decrease the accumulated error will have to reduce (or unwind) to below the value that caused the output to saturate. The Kc coefficient limits this unwanted accumulation. For most situations, it can be set equal to Ki.

All three controllers have a maximum value for the output parameter. These values can be found in the `UserParms.h` file and are currently set to avoid saturation in the `SVGen()` routine.

CONTROL LOOP DEPENDENCIES

There are three PI control loops in this application that are interdependent. The outer loop controls the motor velocity. The two inner loops control the transformed motor currents, I_d and I_q. As mentioned previously, the I_d loop is responsible for controlling flux and the I_q value is responsible for controlling the motor torque.

TORQUE MODE

When adjusting the coefficients for the three control loops, it can be beneficial to separate the outer control loop from the inner loops. The motor can be operated in a torque mode by uncommenting the `#define TORQUE_MODE` statement in the `UserParms.h` file. This will bypass the outer velocity control loop and feed the potentiometer demand value directly to the I_q control loop set-point.

RECOMMENDED CONTROL LOOP TUNING PROCEDURE

If the control loops require adjustment, it is helpful to bypass the velocity control loop as described above. In most situations, the PI coefficients for the I_d and I_q control loops should be set to equal values. Once the motor has good torque response in the torque mode, the velocity control loop can be enabled and adjusted.

Example Scope Plots

The following scope plots demonstrate the use of the diagnostic outputs and proper tuning of the application parameters.

A plot of the transformed quadrature phase current (I_q) vs. the motor mechanical velocity is shown in Figure 14. Assuming the application is properly tuned, the I_q value is proportional to the motor torque. This value can be found in the `ParkParm` data structure. The motor mechanical velocity is in the `EncoderParm` data structure.

The plot shows an example of properly tuned control loops. As you can see, there is little overshoot or ringing in the bottom trace (motor velocity). Also, there is a rapid response in the quadrature current (top trace), followed by a decay with little overshoot or ringing as the motor reaches the new speed.

FIGURE 14: IQ VS. VELOCITY, 500 TO 1000 RPM STEP

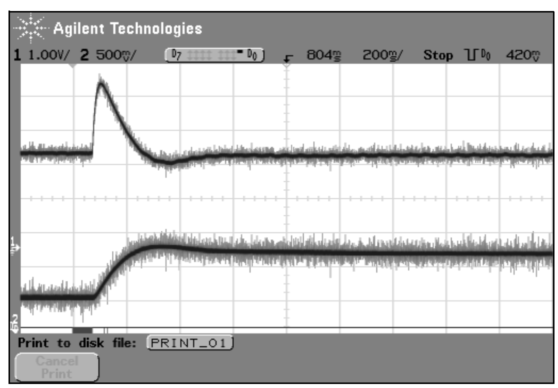


Figure 15 compares the actual AC phase current and the motor velocity during a 1000 RPM to 2000 RPM step change with properly tuned PI loop parameters and the correct motor time constant. The phase current is measured directly from one of the two phase current sensors on the motor control development system. The velocity data is obtained from the `EncoderParm` data structure and sent to one of the PWM diagnostic outputs for display on the scope. In this scope plot you can observe that the velocity moves quickly to the new setpoint with little or no overshoot and ringing. Furthermore, the amplitude of the phase current does not change dramatically during the speed change.

FIGURE 15: PHASE CURRENT VS. VELOCITY, 1000 TO 2000 RPM STEP, TR = 0.078 SEC

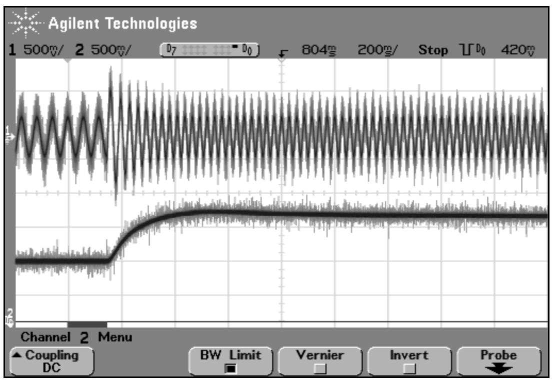


Figure 16 shows the same phase current and velocity data shown in Figure 15. In this case, a step change is made from 1000 RPM to 2000 RPM in open-loop mode. The speed change in open-loop mode requires a higher current amplitude and more time to complete. A comparison of Figure 15 and Figure 16 clearly shows the benefits of vector control. The speed change takes less current to execute in closed-loop mode.

FIGURE 16: PHASE CURRENT VS. VELOCITY, 1000 TO 2000 RPM STEP, OPEN LOOP

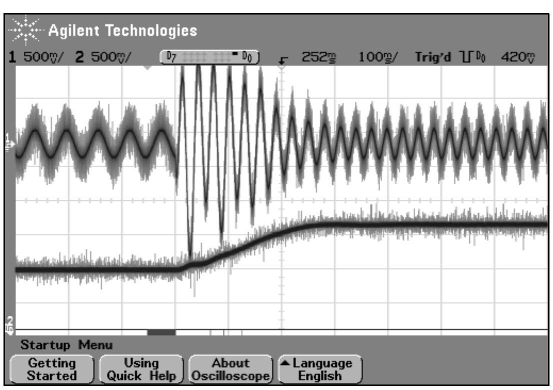
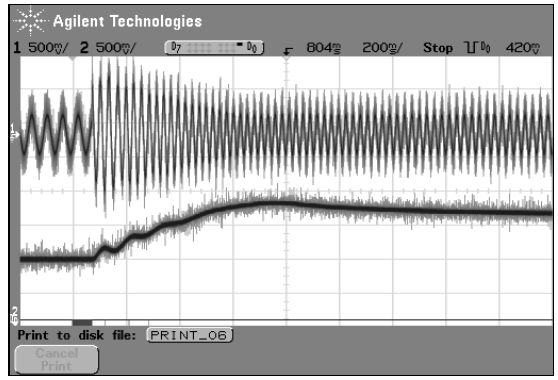


Figure 17 demonstrates a step change with an incorrect rotor time constant value. The step change requires more current and time to execute.

FIGURE 17: PHASE CURRENT VS. VELOCITY, 1000 TO 2000 RPM STEP, TR = 0.039 SEC



APPENDIX A. REFERENCES

1. *Vector Control and Dynamics of AC Drives*, D. W. Novotny, T. A. Lipo, Oxford University Press, 2003, ISBN: 0 19 856439 2.
2. *Modern Power Electronics and AC Drives*, Bimal K. Bose, Pearson Education, 2001, ISBN: 0 13 016743 6.

APPENDIX B. SOURCE CODE

This appendix contains source listings for the files listed below. These are the primary files associated with the vector control algorithm. Other files related to the user interface have not been included in this listing.

If you are viewing an electronic version of this application, you can navigate to a particular file by clicking the file name below.

Header Files

`UserParms.h`

C Files

`ACIM.c`

`Encoder.c`

`InitCurModel.c`

Assembly Files

`CalcRef.s`

`CalcVel.s`

`ClarkePark.s`

`CurModel.s`

`FdWeak.s`

`InvPark.s`

`MeasCur.s`

`OpenLoop.s`

`PI.s`

`ReadADC0.s`

`SVGen.s`

`Trig.s`

UserParms.h

```
//#define TORQUE_MODE
#define DIAGNOSTICS

//****************  Oscillator ************************************
#define dFoscExt          7372800                   // External Crystal or Clock Frequency (Hz)
#define dPLL              8                         // PLL ratio
#define dLoopTimeInSec    0.00005                   // PWM Period - 100 uSec, 10Khz PWM
#define dDeadTimeSec      0.000002                  // Deadtime in seconds
// Derived
#define dFosc             (dFoscExt*dPLL)           // Clock frequency (Hz)
#define dFcy              (dFosc/4)                 // Instruction cycle frequency (Hz)
#define dTcy              (1.0/dFcy)                // Instruction cycle period (sec)
#define dDeadTime         (int)(dDeadTimeSec*dFcy)  // Dead time in dTcys
#define dLoopInTcy        (dLoopTimeInSec/dTcy)     // Basic loop period in units of Tcy
#define dDispLoopTime     0.100                     // Display and button polling loop

//****************  Motor Parameters ******************************
#define diPoles           1                         // Number of pole pairs
#define diCntsPerRev      2000                      // Encoder Lines per revolution
#define diNomRPM          3600                      // Name Plate Motor RPM
#define dfRotorTmConst    0.078                     // Rotor time constant in sec, from mfgr

//**************** Measurement *************************************
#define diIrpPerCalc      30                        // PWM loops per velocity calculation

//************** PI Coefficients ************************************
#define dDqKp             0x2000                    // 4.0     (NKo = 4)
#define dDqKi             0x0100;                   // 0.125
#define dDqKc             0x0100;                   // 0.125
#define dDqOutMax         0x5A82;                   // 0.707   set to prevent saturation

#define dQqKp             0x2000;                   // 4.0     (NKo = 4)
#define dQqKi             0x0100;                   // 0.125
#define dQqKc             0x0100;                   // 0.125
#define dQqOutMax         0x5A82;                   // 0.707   set to prevent saturation

#define dQrefqKp          0x4000                    // 8.0     (NKo = 4)
#define dQrefqKi          0x0800                    // 1.0
#define dQrefqKc          0x0800                    // 1.0
#define dQrefqOutMax      0x3FFF                    // 0.4999  set to prevent saturation

//************** ADC Scaling ***************************************
// Scaling constants:     Determined by calibration or hardware design.
#define dqK               0x3FFF;                   // equivalent to 0.4999
#define dqKa              0x3FFF;                   // equivalent to 0.4999
#define dqKb              0x3FFF;                   // equivalent to 0.4999

//************** Field Weakening ***********************************
// Flux reference value in constant torque range.
// Determined empirically to give rated volts/hertz
#define dqK1              3750;                     //
```

ACIM.c

```
/**********************************************************************
*                                                                     *
*    Author: John Theys/Dave Ross                                     *
*                                                                     *
*    Filename:       ACIM.c                                           *
*    Date:           10/31/03                                         *
*    File Version:   3.00                                             *
*                                                                     *
*    Tools used: MPLAB     -> 6.43                                    *
*                Compiler  -> 1.20.00                                 *
*                                                                     *
*    Linker File:    p30f6010.gld                                     *
*                                                                     *
*                                                                     *
***********************************************************************
*10/31/03  2.00    Released    Motor runs fine, still some loose ends
*
*12/19/03  2.01    Cleaned up structure, created UserParms.h for all user defines.
*
*02/12/043.00-Removed unnecessary files from project.
*          -Changed iRPM to int to correct floating point calc problems.
*          -CalcVel() and velocity control loop only execute after number of loop periods
*              specified by iIrpPerCalc.
*          -Added iDispLoopCount variable to schedule execution of display and button routines
*          -trig.s file changed to use program space for storage of sine data.
*          -Added DiagnosticsOutput() function that uses output compare channels to
*              output control variable information.
*          -Added TORQUE_MODE definition to bypass velocity control loop.
*          -Turned off SATDW bit in curmodel.s file.  The automatic saturation feature prevents
*              slip angle calculation from wrapping properly.
************************************************************************
*   Code Description
*
*   This file demonstrates Vector Control of a 3 phase ACIM using the dsPIC30F.
*   SVM is used as the modulation strategy.
***********************************************************************/

/**************************** GLOBAL DEFINITIONS ***********************/

#define INITIALIZE
#include "Motor.h"
#include "Parms.h"
#include "Encoder.h"
#include "SVGen.h"
#include "ReadADC.h"
#include "MeasCurr.h"
#include "CurModel.h"
#include "FdWeak.h"
#include "Control.h"
#include "PI.h"
#include "Park.h"
#include "OpenLoop.h"
#include "LCD.h"
#include "bin2dec.h"
#include "UserParms.h"

/************************ END OF GLOBAL DEFINITIONS ********************/

unsigned short uWork;
short iCntsPerRev;
short iDeltaPos;
```

```
union   {
        struct
            {
            unsigned DoLoop:1;
            unsigned OpenLoop:1;
            unsigned RunMotor:1;
            unsigned Btn1Pressed:1;
            unsigned Btn2Pressed:1;
            unsigned Btn3Pressed:1;
            unsigned Btn4Pressed:1;
            unsigned ChangeMode:1;
            unsigned ChangeSpeed:1;
            unsigned    :7;
            }bit;
        WORD Word;
        } uGF;                         // general flags

tPIParm    PIParmQ;
tPIParm    PIParmQref;
tPIParm    PIParmD;

tReadADCParm ReadADCParm;

int iRPM;
WORD iMaxLoopCnt;
WORD iLoopCnt;
WORD iDispLoopCnt;

/**********************************************************************/
void __attribute__((__interrupt__)) _ADCInterrupt(void);
void SetupBoard( void );
bool SetupParm(void);
void DoControl( void );
void Dis_RPM( BYTE bChrPosC, BYTE bChrPosR );
void DiagnosticsOutput(void);

/******************** START OF MAIN FUNCTION *************************/

int main ( void )
{
    SetupPorts();
    InitLCD();

    while(1)
        {
        uGF.Word = 0;                  // clear flags

        // init Mode
        uGF.bit.OpenLoop = 1;          // start in openloop

        // init LEDs
        pinLED1 = 0;
        pinLED2 = !uGF.bit.OpenLoop;
        pinLED3 = 0;
        pinLED4 = 0;

        // init board
        SetupBoard();

        // init user specified parms and stop on error
        if( SetupParm() )
            {
            // Error
            uGF.bit.RunMotor=0;
            return;
```

```
    }

// zero out i sums
PIParmD.qdSum = 0;
PIParmQ.qdSum = 0;
PIParmQref.qdSum = 0;

iMaxLoopCnt = 0;

Wrt_S_LCD("Vector Control  ", 0 , 0);
Wrt_S_LCD("S4-Run/Stop     ", 0, 1);

// Enable ADC interrupt and begin main loop timing
IFS0bits.ADIF = 0;
IEC0bits.ADIE = 1;

if(!uGF.bit.RunMotor)
    {
    // Initialize current offset compensation
    while(!pinButton1)                  //wait here until button 1 is pressed
        {
        ClrWdt();

        // Start offset accumulation    //and accumulate current offset while waiting
        MeasCompCurr();

        }
    while(pinButton1);                  //when button 1 is released
    uGF.bit.RunMotor = 1;               //then start motor
    }

// Run the motor
uGF.bit.ChangeMode = 1;
// Enable the driver IC on the motor control PCB
pinPWMOutputEnable_ = 0;

Wrt_S_LCD("RPM=            ", 0, 0);
Wrt_S_LCD("S5-Cls. Lp S6-2x", 0, 1);

//Run Motor loop
while(1)
    {
    ClrWdt();

    // If using OC7 and OC8 to display vector control variables,
    // call the update code.
    #ifdefDIAGNOSTICS
    DiagnosticsOutput();
    #endif

    // The code that updates the LCD display and polls the buttons
    // executes every 50 msec.

    if(iDispLoopCnt >= dDispLoopCnt)
    {
    //Display RPM
    Dis_RPM(5,0);

    // Button 1 starts or stops the motor
        if(pinButton1)
            {
            if( !uGF.bit.Btn1Pressed )
                uGF.bit.Btn1Pressed  = 1;
            }
        else
```

```
        {
        if( uGF.bit.Btn1Pressed )
            {
            // Button just released
            uGF.bit.Btn1Pressed  = 0;
            // begin stop sequence
            uGF.bit.RunMotor = 0;
            pinPWMOutputEnable_ = 1;
            break;
            }
        }

    //while running button 2 will toggle open and closed loop
    if(pinButton2)
        {
        if( !uGF.bit.Btn2Pressed )
            uGF.bit.Btn2Pressed  = 1;
        }
    else
        {
        if( uGF.bit.Btn2Pressed )
            {
            // Button just released
            uGF.bit.Btn2Pressed  = 0;
            uGF.bit.ChangeMode = 1;
            uGF.bit.OpenLoop = ! uGF.bit.OpenLoop;
            pinLED2 = !uGF.bit.OpenLoop;
            }
        }

    //while running button 3 will double/half the speed or torque demand
    if(pinButton3)
        {
        if( !uGF.bit.Btn3Pressed )
            uGF.bit.Btn3Pressed  = 1;
            LATGbits.LATG0 = 0;
        }
    else
        {
        if( uGF.bit.Btn3Pressed )
            {
            // Button just released
            uGF.bit.Btn3Pressed  = 0;
            uGF.bit.ChangeSpeed = !uGF.bit.ChangeSpeed;
            pinLED3 = uGF.bit.ChangeSpeed;
            LATGbits.LATG0 = 1;
            }
        }

    // Button 4 does not do anything
    if(pinButton4)
        {
        if( !uGF.bit.Btn4Pressed )
            uGF.bit.Btn4Pressed  = 1;
        }
    else
        {
        if( uGF.bit.Btn4Pressed )
            {
            // Button just released
            uGF.bit.Btn4Pressed  = 0;
            //*** ADD CODE HERE FOR BUTTON 4 FUNCTION
            }
        }
```



```
            }       // end of display and button polling code

            }       // End of Run Motor loop


        }           // End of Main loop

                    // should never get here
    while(1){}
}

//---------------------------------------------------------------------
// Executes one PI itteration for each of the three loops Id,Iq,Speed
void DoControl( void )
{
short i;

    // Assume ADC channel 0 has raw A/D value in signed fractional form from
    // speed pot (AN7).
    ReadSignedADC0( &ReadADCParm );

    // Set reference speed
    if(uGF.bit.ChangeSpeed)
        CtrlParm.qVelRef = ReadADCParm.qADValue/8;
    else
        CtrlParm.qVelRef = ReadADCParm.qADValue/16;

    if( uGF.bit.OpenLoop )
        {
    `   // OPENLOOP:  force rotating angle,Vd,Vq

        if( uGF.bit.ChangeMode )
            {
            // just changed to openloop
            uGF.bit.ChangeMode = 0;
            // synchronize angles
            OpenLoopParm.qAngFlux = CurModelParm.qAngFlux;

            // VqRef & VdRef not used
            CtrlParm.qVqRef = 0;
            CtrlParm.qVdRef = 0;
            }

        OpenLoopParm.qVelMech = CtrlParm.qVelRef;

        // calc rotational angle of rotor flux in 1.15 format

        // just for reference & sign needed by CorrectPhase
        CurModelParm.qVelMech = EncoderParm.qVelMech;
        CurModel();

        ParkParm.qVq = 0;

        if( OpenLoopParm.qVelMech >= 0 )
            i = OpenLoopParm.qVelMech;
        else
            i = -OpenLoopParm.qVelMech;

        uWork = i <<2;

        if( uWork > 0x5a82 )
            uWork = 0x5a82;

        if( uWork < 0x1000 )
            uWork = 0x1000;
```

```
        ParkParm.qVd = uWork;

        OpenLoop();
        ParkParm.qAngle = OpenLoopParm.qAngFlux;

        }
    else
        // Closed Loop Vector Control
        {

        if( uGF.bit.ChangeMode )
            {
            // just changed from openloop
            uGF.bit.ChangeMode = 0;

            // synchronize angles and prep qdImag
            CurModelParm.qAngFlux = OpenLoopParm.qAngFlux;
            CurModelParm.qdImag = ParkParm.qId;
            }

        // Current model calculates angle
        CurModelParm.qVelMech = EncoderParm.qVelMech;

        CurModel();

        ParkParm.qAngle = CurModelParm.qAngFlux;

        // Calculate qVdRef from field weakening
        FdWeakening();

        // Set reference speed

        // If the application is running in torque mode, the velocity
        // control loop is bypassed.  The velocity reference value, read
        // from the potentiometer, is used directly as the torque
        // reference, VqRef.
        #ifdefTORQUE_MODE
        CtrlParm.qVqRef    = CtrlParm.qVelRef;

        #else
        // Check to see if new velocity information is available by comparing
        // the number of interrupts per velocity calculation against the
        // number of velocity count samples taken.  If new velocity info
        // is available, calculate the new velocity value and execute
        // the speed control loop.
        if(EncoderParm.iVelCntDwn == EncoderParm.iIrpPerCalc)
            {
            // Calculate velocity from acumulated encoder counts
        CalcVel();
        // Execute the velocity control loop
        PIParmQref.qInMeas = EncoderParm.qVelMech;
        PIParmQref.qInRef  = CtrlParm.qVelRef;
        CalcPI(&PIParmQref);
        CtrlParm.qVqRef    = PIParmQref.qOut;
        }
        #endif

        // PI control for Q
        PIParmQ.qInMeas = ParkParm.qIq;
        PIParmQ.qInRef  = CtrlParm.qVqRef;
        CalcPI(&PIParmQ);
        ParkParm.qVq    = PIParmQ.qOut;

        // PI control for D
```

```
        PIParmD.qInMeas = ParkParm.qId;
        PIParmD.qInRef  = CtrlParm.qVdRef;
        CalcPI(&PIParmD);
        ParkParm.qVd    = PIParmD.qOut;


        }
}

//---------------------------------------------------------------------
// The ADC ISR does speed calculation and executes the vector update loop.
// The ADC sample and conversion is triggered by the PWM period.
// The speed calculation assumes a fixed time interval between calculations.
//---------------------------------------------------------------------

void __attribute__((__interrupt__)) _ADCInterrupt(void)
{
        IFS0bits.ADIF = 0;

        // Increment count variable that controls execution
        // of display and button functions.
        iDispLoopCnt++;

        // acumulate encoder counts since last interrupt
        CalcVelIrp();

        if( uGF.bit.RunMotor )
            {

            // Set LED1 for diagnostics
            pinLED1 = 1;

            // use TMR1 to measure interrupt time for diagnostics
            TMR1 = 0;
            iLoopCnt = TMR1;

             MeasCompCurr();

            // Calculate qId,qIq from qSin,qCos,qIa,qIb
            ClarkePark();

            // Calculate control values
            DoControl();

                // Calculate qSin,qCos from qAngle
                SinCos();

                // Calculate qValpha, qVbeta from qSin,qCos,qVd,qVq
                InvPark();

                // Calculate Vr1,Vr2,Vr3 from qValpha, qVbeta
                CalcRefVec();

                // Calculate and set PWM duty cycles from Vr1,Vr2,Vr3
                CalcSVGen();

                // Measure loop time
                iLoopCnt = TMR1 - iLoopCnt;
                if( iLoopCnt > iMaxLoopCnt )
                    iMaxLoopCnt = iLoopCnt;

                // Clear LED1 for diagnostics
                pinLED1 = 0;
                }
}
```

```
//---------------------------------------------------------------------
// SetupBoard
//
// Initialze board
//---------------------------------------------------------------------

void SetupBoard( void )
{
    BYTE b;

    // Disable ADC interrupt
    IEC0bits.ADIE = 0;

    // Reset any active faults on the motor control power module.
    pinFaultReset = 1;
    for(b=0;b<10;b++)
        Nop();
    pinFaultReset = 0;


    // Ensure PFC switch is off.
    pinPFCFire = 0;
    // Ensure brake switch is off.
    pinBrakeFire = 0;
}

//---------------------------------------------------------------------
// Dis_RPM
//
// Display RPM
//---------------------------------------------------------------------

void Dis_RPM( BYTE bChrPosC, BYTE bChrPosR )
{

    if (EncoderParm.iDeltaCnt < 0)
        Wrt_S_LCD("-", bChrPosC, bChrPosR);
    else
        Wrt_S_LCD(" ", bChrPosC, bChrPosR);

    iRPM =
EncoderParm.iDeltaCnt*60/(MotorParm.fLoopPeriod*MotorParm.iIrpPerCalc*EncoderParm.iCntsPerRev);
    Wrt_Signed_Int_LCD( iRPM, bChrPosC+1, bChrPosR);
}
//---------------------------------------------------------------------
bool SetupParm(void)
{
    // Turn saturation on to insure that overflows will be handled smoothly.
    CORCONbits.SATA  = 0;

    // Setup required parameters

    // Pick scaling values to be 8 times nominal for speed and current

    // Use 8 times nominal mechanical speed of motor (in RPM) for scaling
    MotorParm.iScaleMechRPM  = diNomRPM*8;

    // Number of pole pairs
    MotorParm.iPoles        = diPoles ;

    // Encoder counts per revolution as detected by the
    //       dsPIC quadrature configuration.
    MotorParm.iCntsPerRev  = diCntsPerRev;
```



```
    // Rotor time constant in sec
    MotorParm.fRotorTmConst = dfRotorTmConst;

    // Basic loop period (in sec).  (PWM interrupt period)
    MotorParm.fLoopPeriod = dLoopInTcy * dTcy;  //Loop period in cycles * sec/cycle

    // Encoder velocity interrupt period (in sec).
    MotorParm.fVelIrpPeriod = MotorParm.fLoopPeriod;

    // Number of vel interrupts per velocity calculation.
    MotorParm.iIrpPerCalc = diIrpPerCalc;       // In loops

    // Scale mechanical speed of motor (in rev/sec)
    MotorParm.fScaleMechRPS = MotorParm.iScaleMechRPM/60.0;

    // Scaled flux speed of motor (in rev/sec)
    // All dimensionless flux velocities scaled by this value.
    MotorParm.fScaleFluxRPS = MotorParm.iPoles*MotorParm.fScaleMechRPS;

    // Minimum period of one revolution of flux vector (in sec)
    MotorParm.fScaleFluxPeriod = 1.0/MotorParm.fScaleFluxRPS;

    // Fraction of revolution per LoopTime at maximum flux velocity
    MotorParm.fScaleFracRevPerLoop = MotorParm.fLoopPeriod * MotorParm.fScaleFluxRPS;

    // Scaled flux speed of motor (in radians/sec)
    // All dimensionless velocities in radians/sec scaled by this value.
    MotorParm.fScaleFluxSpeed = 6.283 * MotorParm.fScaleFluxRPS;

    // Encoder count rate at iScaleMechRPM
    MotorParm.lScaleCntRate = MotorParm.iCntsPerRev * (MotorParm.iScaleMechRPM/60.0);



// ============= Open Loop ======================

    OpenLoopParm.qKdelta = 32768.0 * 2 * MotorParm.iPoles * MotorParm.fLoopPeriod * 
MotorParm.fScaleMechRPS;
    OpenLoopParm.qVelMech = dqOL_VelMech;
    CtrlParm.qVelRef = OpenLoopParm.qVelMech;

    InitOpenLoop();

// ============= Encoder ===============

    if( InitEncoderScaling() )
        // Error
        return True;

// ============= ADC - Measure Current & Pot ======================

    // Scaling constants: Determined by calibration or hardware design.
    ReadADCParm.qK      = dqK;

    MeasCurrParm.qKa    = dqKa;
    MeasCurrParm.qKb    = dqKb;

    // Inital offsets
    InitMeasCompCurr( 450, 730 );

// ============= Current Model ===============

    if(InitCurModelScaling())
        // Error
        return True;
```

```
// ============= Field Weakening ===============
    // Field Weakening constant for constant torque range
    FdWeakParm.qK1 = dqK1;       // Flux reference value

// ============= PI D Term ===============
    PIParmD.qKp = dDqKp;
    PIParmD.qKi = dDqKi;
    PIParmD.qKc = dDqKc;
    PIParmD.qOutMax = dDqOutMax;
    PIParmD.qOutMin = -PIParmD.qOutMax;

    InitPI(&PIParmD);

// ============= PI Q Term ===============
    PIParmQ.qKp = dQqKp;
    PIParmQ.qKi = dQqKi;
    PIParmQ.qKc = dQqKc;
    PIParmQ.qOutMax = dQqOutMax;
    PIParmQ.qOutMin = -PIParmQ.qOutMax;

    InitPI(&PIParmQ);

// ============= PI Qref Term ===============
    PIParmQref.qKp = dQrefqKp;
    PIParmQref.qKi = dQrefqKi;
    PIParmQref.qKc = dQrefqKc;
    PIParmQref.qOutMax = dQrefqOutMax;
    PIParmQref.qOutMin = -PIParmQref.qOutMax;

    InitPI(&PIParmQref);

// ============= SVGen ===============
    // Set PWM period to Loop Time
    SVGenParm.iPWMPeriod = dLoopInTcy;

// ============= TIMER #1 ======================
    PR1 = 0xFFFF;
    T1CONbits.TON = 1;
    T1CONbits.TCKPS = 1;    // prescale of 8 => 1.08504 uS tick

// ============= Motor PWM ======================

    PDC1 = 0;
    PDC2 = 0;
    PDC3 = 0;
    PDC4 = 0;

    // Center aligned PWM.
    // Note: The PWM period is set to dLoopInTcy/2 but since it counts up and
    // and then down => the interrupt flag is set to 1 at zero => actual
    // interrupt period is dLoopInTcy

    PTPER = dLoopInTcy/2;   // Setup PWM period to Loop Time defined in parms.h

    PWMCON1 = 0x0077;       // Enable PWM 1,2,3 pairs for complementary mode
    DTCON1 = dDeadTime;     // Dead time
    DTCON2 = 0;
    FLTACON = 0;            // PWM fault pins not used
    FLTBCON = 0;
    PTCON = 0x8002;         // Enable PWM for center aligned operation

    // SEVTCMP: Special Event Compare Count Register
    // Phase of ADC capture set relative to PWM cycle: 0 offset and counting up
    SEVTCMP = 2;        // Cannot be 0 -> turns off trigger (Missing from doc)
```



```
    SEVTCMPbits.SEVTDIR = 0;

// ============= Encoder ===============

    MAXCNT = MotorParm.iCntsPerRev;
    POSCNT = 0;
    QEICON = 0;
    QEICONbits.QEIM = 7;    // x4 reset by MAXCNT pulse
    QEICONbits.POSRES = 0;  // Don't allow Index pulse to reset counter
    QEICONbits.SWPAB = 0;   // direction
    DFLTCON = 0;            // Digital filter set to off

// ============= ADC - Measure Current & Pot ======================
// ADC setup for simultanous sampling on
//      CH0=AN7, CH1=AN0, CH2=AN1, CH3=AN2.
// Sampling triggered by PWM and stored in signed fractional form.

    ADCON1 = 0;
    // Signed fractional (DOUT = sddd dddd dd00 0000)
    ADCON1bits.FORM = 3;
    // Motor Control PWM interval ends sampling and starts conversion
    ADCON1bits.SSRC = 3;
    // Simultaneous Sample Select bit (only applicable when CHPS = 01 or 1x)
    // Samples CH0, CH1, CH2, CH3 simultaneously (when CHPS = 1x)
    // Samples CH0 and CH1 simultaneously (when CHPS = 01)
    ADCON1bits.SIMSAM = 1;
    // Sampling begins immediately after last conversion completes.
    // SAMP bit is auto set.
    ADCON1bits.ASAM = 1;


    ADCON2 = 0;
    // Samples CH0, CH1, CH2, CH3 simultaneously (when CHPS = 1x)
    ADCON2bits.CHPS = 2;


    ADCON3 = 0;
    // A/D Conversion Clock Select bits = 8 * Tcy
    ADCON3bits.ADCS = 15;


    /* ADCHS: ADC Input Channel Select Register */
    ADCHS = 0;
    // CH0 is AN7
    ADCHSbits.CH0SA = 7;
    // CH1 positive input is AN0, CH2 positive input is AN1, CH3 positive input is AN2
    ADCHSbits.CH123SA = 0;


    /* ADPCFG: ADC Port Configuration Register */
    // Set all ports digital
    ADPCFG = 0xFFFF;
    ADPCFGbits.PCFG0 = 0;   // AN0 analog
    ADPCFGbits.PCFG1 = 0;   // AN1 analog
    ADPCFGbits.PCFG2 = 0;   // AN2 analog
    ADPCFGbits.PCFG7 = 0;   // AN7 analog

    /* ADCSSL: ADC Input Scan Select Register */
    ADCSSL = 0;

    // Turn on A/D module
    ADCON1bits.ADON = 1;

    #ifdefDIAGNOSTICS
    // Initialize Output Compare 7 and 8 for use in diagnostics.
```

```
    // Compares are used in PWM mode
    // Timer2 is used as the timebase
    PR2 = 0x1FFF;
    OC7CON = 0x0006;
    OC8CON = 0x0006;
    T2CONbits.TON = 1;
    #endif


    return False;
}

#ifdefDIAGNOSTICS
void DiagnosticsOutput(void)
{
int Data;

    if(IFS0bits.T2IF)
        {
        IFS0bits.T2IF = 0;
        Data = (ParkParm.qIq >> 4) + 0xfff;
        if(Data > 0x1ff0) Data = 0x1ff0;
        if(Data < 0x000f) Data = 0x000f;
        OC7RS = Data;
        Data =  (EncoderParm.qVelMech) + 0x0fff;
        if(Data > 0x1ff0) Data = 0x1ff0;
        if(Data < 0x000f) Data = 0x000f;
        OC8RS = Data;
        }

}
#endif
```


Encoder.c

```
// Scaling for encoder routines

#include "general.h"
#include "Parms.h"
#include "Encoder.h"

/**********************************************************
InitEncoderScaling

Initialize scaling constants for encoder rotuines.

    Arguments:
        CntsPerRev: Encoder counts per revolution from quadrature
        ScalingSpeedInRPS: Rev per sec used for basic velocity scaling
        IrpPerCalc: Number of CalcVelIrp interrupts per velocity calculation
        VelIrpPeriod: Period between VelCalcIrp interrupts (in Sec)

For CalcAng:
    Runtime equation:
    qMechAng = qKang * (POSCNT*4) / 2^Nang
    Scaling equations:
        qKang = (2^15)*(2^Nang)/CntsPerRev.
For CalcVelIrp, CalcVel:
    Runtime equation:
        qMechVel = qKvel * (2^15 * Delta / 2^Nvel)
    Scaling equations:
        fVelCalcPeriod = fVelIrpPeriod * iIrpPerCalc
        MaxCntRate = CntsPerRev * ScaleMechRPS
        MaxDeltaCnt = fVelCalcPeriod * MaxCntRate
        qKvel = (2^15)*(2^Nvel)/MaxDeltaCnt
**********************************************************/

bool InitEncoderScaling( void )
{
    float fVelCalcPeriod, fMaxCntRate;
    long  MaxDeltaCnt;
    long  K;

    EncoderParm.iCntsPerRev = MotorParm.iCntsPerRev;

    K = 32768;
    K *= 1 << Nang;
    EncoderParm.qKang = K/EncoderParm.iCntsPerRev;

    EncoderParm.iIrpPerCalc = MotorParm.iIrpPerCalc;
    fVelCalcPeriod = MotorParm.fVelIrpPeriod * MotorParm.iIrpPerCalc;
    fMaxCntRate = EncoderParm.iCntsPerRev * MotorParm.fScaleMechRPS;
    MaxDeltaCnt = fVelCalcPeriod * fMaxCntRate;

    // qKvel = (2^15)*(2^Nvel)/MaxDeltaCnt
    K = 32768;
    K *= 1 << Nvel;
    K /= MaxDeltaCnt;
    if( K >= 32768 )
        // Error
        return True;
    EncoderParm.qKvel = K;

    // Initialize private variables used by CalcVelIrp.
    InitCalcVel();
    return False;
}
```

InitCurModel.c

```
// Scaling for current model routine

#include "general.h"
#include "Parms.h"
#include "CurModel.h"

/**********************************************************
InitCurModelScaling

Initialize scaling constants for current model routine.

Physical constants:
  fRotorTmConst        Rotor time constant in sec

Physical form of equations:
  Magnetizing current (amps):
     Imag = Imag + (fLoopPeriod/fRotorTmConst)*(Id - Imag)

  Slip speed in RPS:
     VelSlipRPS = (1/fRotorTmConst) * Iq/Imag / (2*pi)

  Rotor flux speed in RPS:
     VelFluxRPS = iPoles * VelMechRPS + VelSlipRPS

  Rotor flux angle (radians):
     AngFlux = AngFlux + fLoopPeriod * 2 * pi * VelFluxRPS

Scaled Variables:
  qImag      Magnetizing current scaled by maximum current
  qVelSlip   Mechnical Slip velocity in RPS scaled by fScaleMechRPS
  qAngFlux   Flux angle scaled by pi

Scaled Equations:
  qImag      = qImag + qKcur * (qId - qImag)
  qVelSlip   = Kslip * qIq/qImag
  qAngFlux   = qAngFlux + Kdelta * (qVelMech + qVelSlip)

Scaling factors:
  qKcur = (2^15) * (fLoopPeriod/fRotorTmConst)
  qKdelta = (2^15) * 2 * iPoles * fLoopPeriod * fScaleMechRPS
  qKslip = (2^15)/(2 * pi * fRotorTmConst * iPoles * fScaleMechRPS)



**********************************************************/

bool InitCurModelScaling( void )
{
    CurModelParm.qKcur = 32768.0 * MotorParm.fLoopPeriod / MotorParm.fRotorTmConst;

    CurModelParm.qKdelta = 32768.0 * 2 * MotorParm.iPoles * MotorParm.fLoopPeriod *
MotorParm.fScaleMechRPS;

    CurModelParm.qKslip = 32768.0/(6.2832 * MotorParm.iPoles *
MotorParm.fScaleMechRPS*MotorParm.fRotorTmConst);

    // Maximum allowed slip speed
    CurModelParm.qMaxSlipVel = 32768.0/8;

    // Initialize private variables used by CurrModel
    InitCurModel();
    return False;
}
```

MeasCur.s

```
;*******************************************************************
; MeasCompCurr
;
; Description:
;   Read Channels 1 & 2 of ADC, scale them as signed fractional values
;   using qKa, qKb and put the results qIa and qIb of ParkParm.
;   Running average value of ADC-Ave is maintained and subtracted from
;   ADC value before scaling.
;
;   Specifically the offset is accumulated as a 32-bit signed integer
;       iOffset += (ADC-Offset)
;   and is used to correct the raw ADC by
;       CorrADC = ADCBUFn - iOffset/2^16
;   which gives an offset time constant of ~ MeasurementPeriod*2^16
;
;   Do not call this routine until conversion is completed.
;
;   Scaling constant, qKa and qKb, must be set elsewhere such that
;       qIa = 2 * qKa * CorrADC1
;       qIb = 2 * qKb * CorrADC2
;   The factor of 2 is designed to allow qKa & qKb to be given in 1.15.
;
; Functional prototypes:
;       void MeasCompCurr( void );
;       void InitMeasCompCurr( short iOffset_a, short iOffset_b );
;
; On Start:    Must call InitMeasCompCurr.
; On Entry:    MeasCurrParm structure must contain qKa & qKb.
;              ADC channels 1 & 2must contain signed fractional value.
; On Exit:     ParkParm will contain qIa & qIb.
;
;Parameters:
;   Input arguments:
;       None
;   Return:
;       Void
;   SFR Settings required:
;       CORCON.SATA  = 0
;   If there is any chance that Accumulator will overflow must set
;       CORCON.SATDW = 1
;
;   Support routines required:
;       None
;   Local Stack usage:
;       None
;   Registers modified:
;       w0,w1,w4,w5
;   Timing:
;       29 cycles
;*******************************************************************

        global   _MeasCompCurr
        global   MeasCompCurr

_MeasCompCurr:
MeasCompCurr:

    ;; CorrADC1 = ADCBUF1 - iOffsetHa/2^16
    ;; qIa = 2 * qKa * CorrADC1
        mov.w      _MeasCurrParm+ADC_iOffsetHa,w0
        sub.w      _ADCBUF1,WREG                w0 = ADC - Offset
        clr.w      w1
        btsc       w0,#15
        setm       w1
```

```
        mov.w       w0,w5
        mov.w       _MeasCurrParm+ADC_qKa,w4
        mpy         w4*w5,A
        sac         A,#-1,w4
        mov.w       w4,_ParkParm+Park_qIa

    ;; iOffset += (ADC-Offset)
            add     _MeasCurrParm+ADC_iOffsetLa
            mov.w   w1,w0
            addc    _MeasCurrParm+ADC_iOffsetHa

    ;; CorrADC2 = ADCBUF2 - iOffsetHb/2^16
    ;; qIb = 2 * qKb * CorrADC2
        mov.w       _MeasCurrParm+ADC_iOffsetHb,w0
        sub.w       _ADCBUF2,WREG                 ; w0 = ADC - Offset
        clr.w       w1
        btsc        w0,#15
        setm        w1
        mov.w       w0,w5
        mov.w       _MeasCurrParm+ADC_qKb,w4
        mpy         w4*w5,A
        sac         A,#-1,w4
        mov.w       w4,_ParkParm+Park_qIb

    ;; iOffset += (ADC-Offset)
        add         _MeasCurrParm+ADC_iOffsetLb
        mov.w       w1,w0
        addc        _MeasCurrParm+ADC_iOffsetHb

        return
```

ClarkePark.s

```
;*******************************************************************
; ClarkePark
;
; Description:
;   Calculate Clarke & Park transforms.
;   Assumes the Cos and Sin values are in qSin & qCos.
;
;       Ialpha = Ia
;       Ibeta  = Ia*dOneBySq3 + 2*Ib*dOneBySq3;
;           where Ia+Ib+Ic = 0
;
;       Id =  Ialpha*cos(Angle) + Ibeta*sin(Angle)
;       Iq = -Ialpha*sin(Angle) + Ibeta*cos(Angle)
;
;   This routine works the same for both integer scaling and 1.15 scaling.
;
; Functional prototype:
;
;   void ClarkePark( void )
;
;On Entry: ParkParm structure must contain qSin, qCos, qIa and qIb.
;On Exit:  ParkParm will contain qId, qIq
;
; Parameters:
;   Input arguments:
;       None
;   Return:
;       Void
;   SFR Settings required:
;       CORCON.SATA  = 0
;   If there is any chance that (Ia+2*Ib)/sqrt(3) will overflow must set
```

```
;       CORCON.SATDW = 1
;
; Support routines required:
;       None
; Local Stack usage:
;       None
; Registers modified:
;       w3 -> w7
; Timing:
;       20 cycles
;*******************************************************************
;
           include "general.inc"
; External references
           include "park.inc"
; Register usage
           .equ ParmW,    w3                ; Ptr to ParkParm structure
           .equ Sq3W,     w4                ; OneBySq3
           .equ SinW,     w4                ; replaces Work0W
           .equ CosW,     w5
           .equ IaW,      w6                ; copy of qIa
           .equ IalphaW,  w6                ; replaces Ia
           .equ IbW,      w7                ; copy of qIb
           .equ IbetaW,   w7                ; Ibeta  replaces Ib
; Constants
           equ OneBySq3,  0x49E7            ; 1/sqrt(3) in 1.15 format
;=================== CODE ====================
           section  .text
           global   _ClarkePark
           global   ClarkePark

_ClarkePark:
ClarkePark:
    ;; Ibeta = Ia*OneBySq3 + 2*Ib*OneBySq3;

        mov.w   #OneBySq3,Sq3W              ; 1/sqrt(3) in 1.15 format
        mov.w   _ParkParm+Park_qIa,IaW
        mpy     Sq3W*IaW,A
        mov.w   _ParkParm+Park_qIb,IbW
        mac     Sq3W*IbW,A
        mac     Sq3W*IbW,A
        mov.w   _ParkParm+Park_qIa,IalphaW
        mov.w   IalphaW,_ParkParm+Park_qIalpha
        sac     A,IbetaW
        mov.w   IbetaW,_ParkParm+Park_qIbeta

    ;; Ialpha and Ibeta have been calculated. Now do rotation.

    ;; Get qSin, qCos from ParkParm structure
        mov.w   _ParkParm+Park_qSin,SinW
        mov.w   _ParkParm+Park_qCos,CosW

    ;; Id =  Ialpha*cos(Angle) + Ibeta*sin(Angle)

        mpy     SinW*IbetaW,A               ; Ibeta*qSin -> A
        mac     CosW*IalphaW,A              ; add Ialpha*qCos to A
        mov.w   #_ParkParm+Park_qId,ParmW
        sac     A,[ParmW++]                 ; store to qId, inc ptr to qIq

    ;; Iq = -Ialpha*sin(Angle) + Ibeta*cos(Angle)
        mpy     CosW*IbetaW,A               ; Ibeta*qCos -> A
        msc     SinW*IalphaW,A              ; sub Ialpha*qSin from A
        sac     A,[ParmW]                   ; store to qIq
        return
        .end
```

CurModel.s

```
;*******************************************************************
;Routines: CurModel
;*******************************************************************
;Common to all routines in file
           .include "general.inc"
           .include "curmodel.inc"
           .include "park.inc"
;*******************************************************************
; CurModel
;
; Description:
;
; Physical constants:
;   fRotorTmConst          Rotor time constant in sec
;
;Physical form of equations:
;   Magnetizing current (amps):
;       Imag = Imag + (fLoopPeriod/fRotorTmConst)*(Id - Imag)
;
;   Slip speed in RPS:
;       VelSlipRPS = (1/fRotorTmConst) * Iq/Imag / (2*pi)
;
;   Rotor flux speed in RPS:
;       VelFluxRPS = iPoles * VelMechRPS + VelSlipRPS
;
;   Rotor flux angle (radians):
;       AngFlux = AngFlux + fLoopPeriod * 2 * pi * VelFluxRPS
;
; Scaled Variables:
;   qdImag             Magnetizing current scaled by maximum current (1.31)
;   qVelSlip           Mechnical Slip velocity in RPS scaled by fScaleMechRPS
;   qAngFlux           Flux angle scaled by pi
;
; Scaled Equations:
;   qdImag     = qdImag + qKcur * (qId - qdImag)
;   qVelSlip   = qKslip * qIq/qdImag
;   qAngFlux   = qAngFlux + qKdelta * (qVelMech + qVelSlip)
;
; Scaling factors:
;   qKcur      = (2^15) * (fLoopPeriod/fRotorTmConst)
;   qKdelta    = (2^15) * 2 * iPoles * fLoopPeriod * fScaleMechRPS
;   qKslip     = (2^15)/(2 * pi * fRotorTmConst * iPoles * fScaleMechRPS)
;
; Functional prototype:
;
;   void CurModel( void )
;
; On Entry:    CurModelParm structure must contain qKcur, qKslip, iKpoles,
;                          qKdelta, qVelMech, qMaxSlipVel
; On Exit:     CurModelParm will contain qAngFlux, qdImag and qVelSlip
;
; Parameters:
;   Input arguments:
;       None
;   Return:
;       Void
;   SFR Settings required:
;       CORCON.SATA    = 0
;       CORCON.IF      = 0
;
;   Support routines required:
;       None
;   Local Stack usage:
;       0
```



```
;   Registers modified:
:       w0-w7,AccA
;   Timing:
;       72 instruction cycles
;*******************************************************************
;
;=================== CODE =====================
           .section  .text

;   Register usage for CurModel
           .equ SignW,    w2                            ; track sign changes
           .equ ShiftW,   w3                            ; # shifts before divide
           .equ IqW,      w4                            ; Q current (1.15)
           .equ KslipW,   w5                            ; Kslip constant (1.15)
           .equ ImagW,    w7                            ; magnetizing current (1.15)

           .global    _CurModel
           .global    CurModel

_CurModel:
CurModel:
       ;; qdImag = qdImag + qKcur * (qId - qdImag)      ;; magnetizing current
           mov.w      _CurModelParm+CurMod_qdImag,w6
           mov.w      _CurModelParm+CurMod_qdImag+2,w7
           lac        w7,A
           mov.w      w6,ACCALL

           mov.w      _ParkParm+Park_qId,w4
           sub.w      w4,w7,w4                          ; qId-qdImagH
           mov.w      _CurModelParm+CurMod_qKcur,w5

           mac        w4*w5,A                           ; add Kcur*(Id-Imag) to Imag
           sac        A,w7
           mov.w      ACCALL,w6
           mov.w      w6,_CurModelParm+CurMod_qdImag
           mov.w      w7,_CurModelParm+CurMod_qdImag+2

    ;; qVelSlip = qKslip * qIq/qdImag

    ;; First make qIqW and qdImagW positive and save sign in SignW
           clr        SignW                             ; set flag sign to positive

    ;; if( IqW < 0 ) => toggle SignW and set IqW = -IqW
           mov.w      _ParkParm+Park_qIq,IqW
           cp0        IqW
           bra        Z,jCurModSkip
           bra        NN,jCurMod1
           neg        IqW,IqW
           com        SignW,SignW                       ; toggle sign
jCurMod1:
    ;; if( ImagW < 0 ) => toggle SignW and set ImagW = -ImagW
           cp0        ImagW
           bra        NN,jCurMod2
           neg        ImagW,ImagW
           com        SignW,SignW                       ; toggle sign
jCurMod2:
    ;; Calculate Kslip*|IqW| in Acc A to maintain 1.31
           mov.w      _CurModelParm+CurMod_qKslip,KslipW
           mpy        IqW*KslipW,A

    ;; Make sure denominator is > numerator else skip term
           sac        A,w0                              ; temporary
           cp         ImagW,w0                          ; |qdImag| - |Kslip*qIq|
           bra        LEU,jCurModSkip                   ; skip term:  |qdImag| <= |Kslip*qIq|
```

```
    ;; This will not be required for later releases of the 6010 <SILICON_ERR>
            clr.w       ShiftW

    ;; Calculate how many places ImagW can be shifted without putting
    ;; a one in the msb location (preserves sign)
            ff1l        ImagW,ShiftW
            sub.w       ShiftW,#2,ShiftW              ; # shifts necessary to put 1 in bit 14
    ;; Shift:  ImagW = ImagW << ShiftW
            sl          ImagW,ShiftW,ImagW
    ;; Shift AccA, Requires (-ShiftW) to shift left.
            neg         ShiftW,ShiftW
    ;; |Kslip*qIq| = |Kslip*qIq| << ShiftW
            sftac       A,ShiftW

    ;; Do divide of |qKslip*qIq|/|ImagW|.  We know at this point that the
    ;; results will be positive and < 1.0.  We also know that we have maximum
    ;; precision.

            sac         A,w6
            repeat      #17
            divf        w6,ImagW                      ; w0 = KslipW*IqW/ImagW, w1 = remainder
    ;; Limit maximum slip speed
            mov.w       _CurModelParm+CurMod_qMaxSlipVel,w1
            cp          w1,w0                         ; qMaxSlipSpeed - | Kslip*qIq/qdImag |
            bra         NN,jCurMod4
    ;; result too large: replace it with qMaxSlipSpeed
            mov.w       w1,w0
            bra         jCurMod4

jCurModSkip:
    ;; term skipped entirely - set it = 0
            clr.w       w0

jCurMod4:
    ;; set correct sign
            btsc        SignW,#0
            neg         w0,w0
    ;; For testing
            mov.w       w0,_CurModelParm+CurMod_qVelSlip
    ;; Add mechanical velocity
            mov.w       _CurModelParm+CurMod_qVelMech,w4
            add.w       w0,w4,w4
            mov.w       w4,_CurModelParm+CurMod_qVelFlux
    ;; Load AngFlux to Acc A
            mov.w       _CurModelParm+CurMod_qAngFlux,w1
            lac         w1,A

            mov.w       _CurModelParm+CurMod_qKdelta,w5
            mac         w4*w5,A

            sac         A,w4
            mov.w       w4,_CurModelParm+CurMod_qAngFlux
    return
```

InvPark.s

```
;*******************************************************************
; InvPark
;
;Description:
;   Calculate the inverse Park transform. Assumes the Cos and Sin values
;   are in the ParkParm structure.
;          Valpha =  Vd*cos(Angle) - Vq*sin(Angle)
;          Vbeta  =  Vd*sin(Angle) + Vq*cos(Angle)
;   This routine works the same for both integer scaling and 1.15 scaling.
;
;Functional prototype:
;   void InvPark( void )
;On Entry:     The ParkParm structure must contain qCos, qSin, qVd and qVq.
;On Exit:      ParkParm will contain qValpha, qVbeta.
;
;Parameters:
;   Input arguments:              None
;   Return:                       Void
;   SFR Settings required:        CORCON.SATA  = 0
;   Support routines required:    None
;   Local Stack usage:            None
;   Registers modified:           w3 -> w7, A
;   Timing:                       About 14 instruction cycles
;*******************************************************************
;
       include "general.inc"
;   External references
       include "park.inc"
;   Register usage
       .equ ParmW,    w3          ; Ptr to ParkParm structure
       .equ SinW,     w4
       .equ CosW,     w5
       .equ VdW,      w6          ; copy of qVd
       .equ VqW,      w7          ; copy of qVq

;=================== CODE =====================

       .section   .text
       .global    _InvPark
       .global    InvPark

_InvPark:
InvPark:
    ;; Get qVd, qVq from ParkParm structure
       mov.w      _ParkParm+Park_qVd,VdW
       mov.w      _ParkParm+Park_qVq,VqW
    ;; Get qSin, qCos from ParkParm structure
       mov.w      _ParkParm+Park_qSin,SinW
       mov.w      _ParkParm+Park_qCos,CosW

;; Valpha =  Vd*cos(Angle) - Vq*sin(Angle)
       mpy     CosW*VdW,A         ; Vd*qCos -> A
       msc     SinW*VqW,A         ; sub Vq*qSin from A

       mov.w   #_ParkParm+Park_qValpha,ParmW
       sac     A,[ParmW++]        ; store to qValpha, inc ptr to qVbeta

    ;; Vbeta  =  Vd*sin(Angle) + Vq*cos(Angle)
       mpy     SinW*VdW,A         ; Vd*qSin -> A
       mac     CosW*VqW,A         ; add Vq*qCos to A
       sac     A,[ParmW]          ; store to Vbeta

       return
```

CalcRef.s

```
;*******************************************************************
; CalcRefVec
;
; Description:
;   Calculate the scaled reference vector, (Vr1,Vr2,Vr3), from qValpha,qVbeta.
;   The method is an modified inverse Clarke transform where Valpha & Vbeta
;   are swaped compared to the normal Inverse Clarke.
;
;       Vr1 = Vbeta
;       Vr2 = (-Vbeta/2 + sqrt(3)/2 * Valpha)
;       Vr3 = (-Vbeta/2 - sqrt(3/2) * Valpha)
;
; Functional prototype:
;
; void CalcRefVec( void )
;
; On Entry:The ParkParm structure must contain qCos, qSin, qValpha and qVbeta.
; On Exit: SVGenParm will contain qVr1, qVr2, qVr3
;
; Parameters:
;   Input arguments:
;       None
; Return:
;       Void
; SFR Settings required:
;       CORCON.SATA = 0
; Support routines required:
;       None
; Local Stack usage:
;       None
; Registers modified:
;       w0, w4, w5, w6
; Timing:
;       About 20 instruction cycles
;*******************************************************************
;
        .include "general.inc"

; External references
        .include "park.inc"
        .include "SVGen.inc"
; Register usage
        .equ WorkW,        w0                     ; working
        .equ ValphaW,      w4                     ; qValpha (scaled)
        .equ VbetaW,       w5                     ; qVbeta (scaled)
        .equ ScaleW,       w6                     ; scaling
; Constants
        .equ Sq3OV2,0x6ED9                        ; sqrt(3)/2 in 1.15 format
;=================== CODE =====================

        .section           .text
        .global            _CalcRefVec
        .global            CalcRefVec

_CalcRefVec:
CalcRefVec:
    ;; Get qValpha, qVbeta from ParkParm structure
        mov.w              ParkParm+Park_qValpha,ValphaW
        mov.w              _ParkParm+Park_qVbeta,VbetaW
    ;; Put Vr1 = Vbeta
        mov.w              VbetaW,_SVGenParm+SVGen_qVr1
    ;; Load Sq(3)/2
        mov.w              #Sq3OV2,ScaleW
```



```
;; AccA = -Vbeta/2
    neg.w               VbetaW,VbetaW
    lac                 VbetaW,#1,A
;; Vr2 = -Vbeta/2 + sqrt(3)2 * Valpha)
    mac                 ValphaW*ScaleW,A ; add Valpha*sqrt(3)/2 to A
    sac                 A,WorkW
    mov.w               WorkW,_SVGenParm+SVGen_qVr2
;; AccA = -Vbeta/2
    lac                 VbetaW,#1,A
;; Vr3 = (-Vbeta/2 - sqrt(3)2 * Valpha)
    msc                 ValphaW*ScaleW,A ; sub Valpha*sqrt(3)2 to A
    sac                 A,WorkW
    mov.w               WorkW,_SVGenParm+SVGen_qVr3
    return
    .end
```

CalcVel.s

```
;*******************************************************************
; Routines: InitCalcVel, CalcVel
;
;*******************************************************************
; Common to all routines in file

        .include "general.inc"
        .include "encoder.inc"

;*******************************************************************
; void InitCalcVel(void)
;       Initialize private velocity variables.
;       iIrpPerCalc must be set on entry.
;*******************************************************************

; Register usage for InitCalcVel

        .equ Work0W,  w4   ; Working register
        .equ PosW,    w5   ; current position: POSCNT

;*******************************************************************

        .global   _InitCalcVel
        .global   InitCalcVel
_InitCalcVel:
InitCalcVel:

    ;; Disable interrupts for the next 5 instructions
        DISI      #5

    ;; Load iPrevCnt & zero Delta
    ;; encoder value. Note: To get accurate velocity qVelMech must be
    ;; calculated twice.
        mov.w     POSCNT,PosW         ; current encoder value
        mov.w     PosW,_EncoderParm+Encod_iPrevCnt
        clr.w     _EncoderParm+Encod_iAccumCnt

    ;; Load iVelCntDwn
        mov.w     _EncoderParm+Encod_iIrpPerCalc,WREG
        mov.w     WREG,_EncoderParm+Encod_iVelCntDwn

        return

;*******************************************************************
; CalcVelIrp
;
; Called from timer interrupt at specified intervals.
;
; The interrupt interval, VelPeriod, MUST be less than the minimum time
; required for 1/2 revolution at maximum speed.
;
; This routine will accumulate encoder change for iIrpPerCalc interrupts,
; a period of time  = iIrpPerCalc * VelPeriod, and then copy the accumulation
; to iDeltaCnt for use by the CalcVel routine to calculate velocity.
; The accumulation is set back to zero and a new accumulation starts.
;
;Functional prototype: void CalcVelIrp( void );
;
;On Entry:             EncoderParm must contain iPrevCnt, iAccumCnt, iVelCntDwn
;
;On Exit:              EncoderParm will contain iPrevCnt, iAccumCnt and iDeltaCnt
;                      (if countdown reached zero).
;
```

```
;Parameters:
;   Input arguments               None
;
;   Return:
;       Void
;
;   SFR Settings required         None
;
;   Support routines required:    None
;
;   Local Stack usage:            3
;
;   Registers modified:           None
;
;   Timing:                       About  29 instruction cycles (if new iDeltaCnt produced)
;
;=====================================================
; Equivalent C code
;   {
;   register short Pos, Delta;
;
;   Pos = POSCNT;
;
;   Delta = Pos - EncoderParm.iPrevCnt;
;   EncoderParm.iPrevCnt = Pos;
;
;   if( iDelta >= 0 )
;       {
;       // Delta > 0 either because
;       //      1) vel is > 0 or
;       //      2) Vel < 0 and encoder wrapped around
;
;       if( Delta >=  EncoderParm.iCntsPerRev/2 )
;           {
;           // Delta >= EncoderParm.iCntsPerRev/2 => Neg speed, wrapped around
;
;           Delta -= EncoderParm.iCntsPerRev;
;           }
;       }
;   else
;       {
;       // Delta < 0 either because
;       //      1) vel is < 0 or
;       //      2) Vel > 0 and wrapped around
;;
;       if( Delta < -EncoderParm.iCntsPerRev/2 )
;           {
;           // Delta < -EncoderParm.iCntsPerRev/2 => Pos vel, wrapped around
;
;           Delta += EncoderParm.iCntsPerRev;
;           }
;       }
;
;   EncoderParm.iAccumCnt += Delta;
;
;   EncoderParm.iVelCntDwn--;
;   if(EncoderParm.iVelCntDwn)
;       return;
;
;   iVelCntDwn = iIrpPerCalc;
;   qVelMech = qKvel * iAccumCnt * 2^Nvel;
;   EncoderParm.iAccumCnt = 0;
;}
```

```
;================== CODE ====================
; Register usage for CalcVelIrp
       .equ PosW,     w0                ; current position: POSCNT

       .equ WorkW,    w4                ; Working register
       .equ DeltaW,   w6                ; NewCnt - PrevCnt

       .global   _CalcVelIrp
       .global   CalcVelIrp

_CalcVelIrp:
CalcVelIrp:

    ;; Save registers
       push           w0
       push           w4
       push           w6

    ;; Pos = uTestPos;

 .ifdef SIMU
       mov.w     _uTestPos,PosW                      ; encoder value  ??
 .else
       mov.w     POSCNT,PosW                         ; encoder value
 .endif

       mov.w     _EncoderParm+Encod_iPrevCnt,WorkW

    ;; Update previous cnt with new cnt
       mov.w     PosW,_EncoderParm+Encod_iPrevCnt

    ;; Calc Delta = New - Prev
       sub.w     PosW,WorkW,DeltaW
       bra       N,jEncoder5                            ; Delta < 0

    ;; Delta > 0 either because
    ;;      1) vel is > 0 or
    ;;      2) Vel < 0 and wrapped around

       lsr.w     _EncoderParm+Encod_iCntsPerRev,WREG    ; WREG = CntsPerRev/2

    ;; Is Delta < CntsPerRev/2
       sub.w     DeltaW,w0,WorkW                        ; Delta-CntsPerRev/2
       bra       N,jEncoder20                           ; 0 < Delta < CntsPerRev/2, Vel > 0

    ;; Delta >= CntsPerRev/2 => Neg speed, wrapped around
    ;; Delta = Delta - CntsPerRev

       mov.w     _EncoderParm+Encod_iCntsPerRev,w0
       sub.w     DeltaW,w0,DeltaW

    ;; Delta < 0, Vel < 0
       bra       jEncoder20

 jEncoder5:
    ;; Delta < 0 either because
    ;;       1) vel is < 0 or
    ;;       2) Vel > 0 and wrapped around

       lsr.w     _EncoderParm+Encod_iCntsPerRev,WREG   ; WREG = CntsPerRev/2

    ;; Is Delta + CntsPerRev/2 < 0
       add.w     DeltaW,w0,WorkW  ; Delta+CntsPerRev/2
       bra       NN,jEncoder20       ; -CntsPerRev/2 <= Delta < 0, Vel > 0
```



```
    ;; Delta < -CntsPerRev/2 => Pos vel, wrapped around
    ;; Delta = Delta + CntsPerRev

        mov.w       _EncoderParm+Encod_iCntsPerRev,w0
        add.w       DeltaW,w0,DeltaW

    ;; Delta < -CntsPerRev/2, Vel > 0

jEncoder20:

    ;; Delta now contains signed change in position

    ;; EncoderParm.Delta += Delta;
        mov.w       DeltaW,w0
        add.w       _EncoderParm+Encod_iAccumCnt

    ;; EncoderParm.iVelCntDwn--;
    ;; if(EncoderParm.iVelCntDwn) return;

        dec.w       _EncoderParm+Encod_iVelCntDwn
        cp0.w       _EncoderParm+Encod_iVelCntDwn
        bra         NZ,jEncoder40

    ;; Reload iVelCntDwn: iVelCntDwn = iIrpPerCalc;
        mov.w       _EncoderParm+Encod_iIrpPerCalc,WREG
        mov.w       WREG,_EncoderParm+Encod_iVelCntDwn

    ;; Copy iAccumCnt to iDeltaCnt then iAccumCnt = 0
        mov.w       _EncoderParm+Encod_iAccumCnt,DeltaW
        mov.w       DeltaW,_EncoderParm+Encod_iDeltaCnt
        clr.w       _EncoderParm+Encod_iAccumCnt

jEncoder40:

    ;; Restore registers
        pop         w6
        pop         w4
        pop         w0
        return

;*******************************************************************
; CalcVel
;
; Calculate qVelMech from the last iDeltaCnt produced by the
; interrupt routine CalcVelIrp.
;
;Functional prototype:        void CalcVel( void );
;
;On Entry:                    EncoderParm must contain iDeltaCnt, qKvel
;
;On Exit:                     EncoderParm will contain qVelMech
;
;Parameters:
; Input arguments: None
;
; Return:
;   Void
;
; SFR Settings required:      None
;
; Support routines required:  None
;
; Local Stack usage:          None
;
; Registers modified:         None
```

```
;
; Timing:                         About  8 instruction cycles
;
;*******************************************************************

       .global   _CalcVel
       .global   CalcVel
_CalcVel:
CalcVel:
    ;; qVelMech = qKvel * ( Delta / 2^Nvel / 2^15)


    ;; iDeltaCnt is an integer but as Q15 it = (iDeltaCnt/2^15)
        mov.w      _EncoderParm+Encod_iDeltaCnt,DeltaW
        mov.w      _EncoderParm+Encod_qKvel,WorkW

        mpy        WorkW*DeltaW,A                ; dKvel * (Delta/2^15)
        sac        A,#(Nvel-15),WorkW            ; left shift by 15-Nvel


    ;; qVelMech = qKvel * Q15( Delta / 2^Nvel )
        mov.w      WorkW,_EncoderParm+Encod_qVelMech
        return


        .end
```

FdWeak.s

```
;*******************************************************************
; Routines: FdWeak
;
;*******************************************************************

; Common to all routines in file

        .include "general.inc"
        .include "Control.inc"
        .include "FdWeak.inc"

;*******************************************************************
; FdWeak
;
;Description:
;
;Equations:
;
;Scaling factors:
;Functional prototype:
;
; void FdWeak( void )
;
;On Entry: FdWeakParm structure must contain:_FdWeakParm+FdWeak_qK1
;
;On Exit:  FdWeakParm will contain :         _CtrlParm+Ctrl_qVdRef
;
;Parameters:
; Input arguments: None
;
; Return:
;   Void
;
; SFR Settings required:
;       CORCON.SATA            = 0
;       CORCON.IF              = 0
;
; Support routines required:  None
; Local Stack usage:          0
; Registers modified:         ??w4,w5,AccA
; Timing:                     ??8 instruction cycles
;
;*******************************************************************
;
;=================== CODE =====================
        .section              .text

; Register usage for FdWeak

        .global               _FdWeakening
        .global               FdWeakening

_FdWeakening:
FdWeakening:

        mov.w                 _FdWeakParm+FdWeak_qK1,w0
        mov.w                 w0,_CtrlParm+Ctrl_qVdRef
        return

        .end
```

OpenLoop.s

```
;*******************************************************************
; Routines: OpenLoop
;*******************************************************************
; Common to all routines in file

        .include "general.inc"
        .include "openloop.inc"
;*******************************************************************
; OpenLoop
;
;Description:
;Equations:
;       qDeltaFlux = Kdelta * qVelMech
;       qAngFlux = qAngFlux + Kdelta * qVelMech              ;; rotor flux angle
;
;       qKdelta = (2^15) * 2 * iPoles * fLoopPeriod * fScaleMechRPS
;           where qVelMech is the mechanical velocity in RPS scaled by fScaleMechRPS
;           and the iPoles is required to get Flux vel from Mech vel
;           and the 2 is to scale +/- 2*pi into +/- pi
;Functional prototype:
;
; void OpenLoop( void )
;
;On Entry:   OpenLoopParm structure must contain
;
;On Exit:    OpenLoopParm will contain
;
;Parameters:
; Input arguments:            None
;
; Return:
;   Void
;
; SFR Settings required:
;       CORCON.SATA           = 0
;       CORCON.IF             = 0
;
; Support routines required:  None
; Local Stack usage:          0
; Registers modified:         ??w4,w5,AccA
; Timing:                  ??8 instruction cycles
;*******************************************************************
;
;=================== CODE =====================
        .section  .text

; Register usage for OpenLoop

        .equ Work0W,   w4                         ; Working register
        .equ Work1W,   w5                         ; Working register

        .global        _OpenLoop
        .global        OpenLoop
```

```
_OpenLoop:
OpenLoop:
        mov.w           _OpenLoopParm+OpLoop_qVelMech,Work0W
        mov.w           _OpenLoopParm+OpLoop_qKdelta,Work1W
        mpy             Work0W*Work1W,A
        sac             A,Work0W
        mov.w           Work0W,_OpenLoopParm+OpLoop_qDeltaFlux

    ;; qAngFlux = qAngFlux + qDeltaFlux
        mov.w           OpenLoopParm+OpLoop_qAngFlux,Work1W
        add.w           Work0W,Work1W,Work0W
        mov.w           Work0W,_OpenLoopParm+OpLoop_qAngFlux
        return

;*******************************************************************
; void InitOpenLoop(void)
;         Initialize private OpenLoop variables.
;*******************************************************************

; Register usage for InitOpenLoop


;*******************************************************************

        .global         _InitOpenLoop
        .global         InitOpenLoop
_InitOpenLoop:
InitOpenLoop:

        clr.w           _OpenLoopParm+OpLoop_qAngFlux
        clr.w           _OpenLoopParm+OpLoop_qDeltaFlux
        return

        .end
```

PI.s

```
;*******************************************************************
; PI
;
;Description:  Calculate PI correction.
;
;void CalcPI( tPIParm *pParm)
;{
;   Err  = InRef - InMeas
;   U  = Sum + Kp * Err
;   if( U > Outmax )
;       Out = Outmax
;   else if( U < Outmin )
;       Out = Outmin
;   else
;       Out = U
;   Exc = U - Out
;   Sum = Sum + Ki * Err - Kc * Exc
;}
;
;void InitPI( tPIParm *pParm)
;{
;   Sum = 0
;   Out = 0
;}
;
;----------------------------
; Representation of PI constants:
; The constant Kp is scaled so it can be represented in 1.15 format by
; adjusting the constant by a power of 2 which is removed when the
; calculation is completed.
;
; Kp is scaled Kp = qKp * 2^NKo
;
; Ki & Kc are scaled Ki = qKi, Kc = qKc
;
;
;Functional prototype:
;
; void InitPI( tPIParm *pParm)
; void CalcPI( tPIParm *pParm)
;
;On Entry: PIParm structure must contain qKp,qKi,qKc,qOutMax,qOutMin,
;                                        InRef,InMeas
;On Exit:  PIParm will contain qOut
;
;Parameters:
; Input arguments: tPIParm *pParm
;
; Return:
;   Void
;
; SFR Settings required:
;       CORCON.SATA= 0
;       CORCON.IF  = 0
;
; Support routines required:  None
; Local Stack usage:          0
; Registers modified:         w0-w6,AccA
;
; Timing:
;  31 instruction cycles max, 28 cycles min
;*******************************************************************
```

```
;
       .include "general.inc"

; External references
       .include "PI.inc"

; Register usage

       .equ BaseW0,   w0                      ; Base of parm structure

       .equ OutW1,    w1                      ; Output
       .equ SumLW2,   w2                      ; Integral sum
       .equ SumHW3,   w3                      ; Integral sum

       .equ ErrW4,    w4                      ; Error term: InRef-InMeas
       .equ WorkW5,   w5                      ; Working register
       .equ Unlimit   W6,w6                   ; U: unlimited output
       .equ WorkW7,   w7                      ; Working register
;=================== CODE =====================

       .section   .text

       .global    _InitPI
       .global    InitPI
_InitPI:
InitPI:
       mov.w      w1,[BaseW0+PI_qOut]
           return

       .global    _CalcPI
       .global    CalcPI

_CalcPI:
CalcPI:
    ;; Err  = InRef - InMeas

       mov.w      [BaseW0+PI_qInRef],WorkW7
       mov.w      [BaseW0+PI_qInMeas],WorkW5
       sub.w      WorkW7,WorkW5,ErrW4

    ;; U  = Sum + Kp * Err * 2^NKo
       lac        [++BaseW0],B                ; AccB = Sum
       mov.w      [--BaseW0],WorkW5
       mov.w      WorkW5,ACCBLL

       mov.w      [BaseW0+PI_qKp],WorkW5
       mpy        ErrW4*WorkW5,A
       sftac      A,#-NKo                     ; AccA = Kp*Err*2^NKo
       add        A                           ; Sum = Sum + Kp*Err*2^NKo
       sac        A,UnlimitW6                 ; store U before tests

    ;; if( U > Outmax )
    ;;  Out = Outmax
    ;; else if( U < Outmin )
    ;;  Out = Outmin
    ;; else
    ;;  Out = U

       mov.w      [BaseW0+PI_qOutMax],OutW1
       cp         UnlimitW6,OutW1
       bra        GT,jPI5                     ; U > Outmax; OutW1 = Outmax
```

```
        mov.w       [BaseW0+PI_qOutMin],OutW1
        cp          UnlimitW6,OutW1
        bra         LE,jPI5                  ; U < Outmin; OutW1 = Outmin

        mov.w       UnlimitW6,OutW1          ; OutW1 = U
jPI5:
        mov.w   OutW1,[BaseW0+PI_qOut]

    ;; Ki * Err
        mov.w       [BaseW0+PI_qKi],WorkW5
        mpy         ErrW4*WorkW5,A

    ;; Exc = U - Out
        sub.w       UnlimitW6,OutW1,UnlimitW6

    ;; Ki * Err - Kc * Exc
        mov.w       [BaseW0+PI_qKc],WorkW5
        msc         WorkW5*UnlimitW6,A

    ;; Sum = Sum + Ki * Err - Kc * Exc
        add         A

        sac         A,[++BaseW0]             ; store Sum
        mov.w       ACCALL,WorkW5
        mov.w       WorkW5,[--BaseW0]
        return

        .end
```

ReadADC0.s

```
;*******************************************************************
; ReadADC0 and ReadSignedADC0
;
;Description:
;  Read Channel 0 of ADC, scale it using qK and put results in qADValue.
;  Do not call this routine until conversion is completed.
;
;  ReadADC0 range is qK*(0.0 ->0.9999).
;  ReadSignedADC0 range is qK*(-1.0 ->0.9999).
;
;  Scaling constant, qK, must be set elsewhere such that
;      iResult = 2 * qK * ADCBUF0
;  The factor of 2 is designed to allow qK to be given in 1.15.
;
;
;Functional prototype:
;
; void ReadADC0( tReadADCParm* pParm )          : Calculates unsigned value 0 -> 2*qK
; void ReadSignedADC0( tReadADCParm* pParm )    : Calculates signed value -2*qK -> 2*qK
;
;On Entry:     ReadADCParm structure must contain qK. ADC channel 0
;              must contain signed fractional value.
;   ;On Exit:  ReadADCParm will contain qADValue
;
;Parameters:
; Input arguments: None
;
; Return:
;   Void
;
; SFR Settings required:
;       CORCON.SATA    = 0
;   If there is any chance that Accumulator will overflow must set
;       CORCON.SATDW   = 1
;
; Support routines required: None
; Local Stack usage: None
; Registers modified: w0,w4,w5
; Timing: 13 cycles
;
;*******************************************************************
;
        .include "general.inc"

; External references
        .include "ReadADC.inc"

; Register usage
        .equ ParmBaseW,w0  ; Base of parm structure
        .equ Work0W,   w4
        .equ Work1W,   w5

;=================== CODE =====================

        .section   .text
        .global    _ReadADC0
        .global    ReadADC0
```

```
_ReadADC0:
ReadADC0:

    ;; iResult = 2 * qK * ADCBUF0

        mov.w       [ParmBaseW+ADC_qK],Work0W
        mov.w       _ADCBUF0,Work1W

    ;; change from signed fractional to fractional, i.e. convert
    ;; from -1->.9999 to 0 -> 0.9999
        btg         Work1W,#15
        lsr.w       Work1W,Work1W

        mpy         Work0W*Work1W,A
        sac         A,#-1,Work0W
        mov.w       Work0W,[ParmBaseW+ADC_qADValue]
        return


        .global     _ReadSignedADC0
        .global     ReadSignedADC0

_ReadSignedADC0:
ReadSignedADC0:

    ;; iResult = 2 * qK * ADCBUF0

        mov.w       [ParmBaseW+ADC_qK],Work0W
        mov.w       _ADCBUF0,Work1W

        mpy         Work0W*Work1W,A
        sac         A,#-1,Work0W
        mov.w       Work0W,[ParmBaseW+ADC_qADValue]
        return

        .end
```

SVGen.s

```
;*******************************************************************
; SVGen
;
; Description:  Calculate and load SVGen PWM values.
;
; Functional prototype:
;   void CalcSVGen( void )
;
; On Entry:SVGenParm structure must contain qVr1, qVr2, qVr3
; On Exit: PWM registers loaded
;
; Parameters:
;   Input arguments:
;       None
;   Return:
;       Void
;   SFR Settings required:
;       CORCON.SATA  = 0
;       CORCON.IF    = 0
;   Support routines required:
;       None
;   Local Stack usage:
;       0
;   Registers modified:
;       w0, w2, w3, w4, w5, w6, AccA
;   Timing:
;       34 instruction cycles
;*******************************************************************
; C-Version of code
;
; void CalcRefVec( void )
; {
;   if( Vr1 >= 0 )
;       {
;       // (xx1)
;       if( Vr2 >= 0 )
;           {
;           // (x11)
;           // Must be Sector 3 since Sector 7 not allowed
;           // Sector 3: (0,1,1)  0-60 degrees
;           T1 = Vr2
;           T2 = Vr1
;           CalcTimes();
;           dPWM1 = Ta
;           dPWM2 = Tb
;           dPWM3 = Tc
;           }
;       else
;           {
;           // (x01)
;           if( Vr3 >= 0 )
;               {
;               // Sector 5: (1,0,1)  120-180 degrees
;               T1 = Vr1
;               T2 = Vr3
;               CalcTimes();
;               dPWM1 = Tc
;               dPWM2 = Ta
;               dPWM3 = Tb
;               }
```

```
;           else
;               {
;               // Sector 1: (0,0,1)  60-120 degrees
;               T1 = -Vr2;
;               T2 = -Vr3;
;               CalcTimes();
;               dPWM1 = Tb
;               dPWM2 = Ta
;               dPWM3 = Tc
;               }
;           }
;       }
;   else
;       {
;       // (xx0)
;       if( Vr2 >= 0 )
;           {
;           // (x10)
;           if( Vr3 >= 0 )
;               {
;               // Sector 6: (1,1,0)  240-300 degrees
;               T1 = Vr3
;               T2 = Vr2
;               CalcTimes();
;               dPWM1 = Tb
;               dPWM2 = Tc
;               dPWM3 = Ta
;               }
;           else
;               {
;               // Sector 2: (0,1,0)  300-0 degrees
;               T1 = -Vr3
;               T2 = -Vr1
;               CalcTimes();
;               dPWM1 = Ta
;               dPWM2 = Tc
;               dPWM3 = Tb
;               }
;           }
;       else
;           {
;           // (x00)
;           // Must be Sector 4 since Sector 0 not allowed
;           // Sector 4: (1,0,0)  180-240 degrees
;           T1 = -Vr1
;           T2 = -Vr2
;           CalcTimes();
;           dPWM1 = Tc
;           dPWM2 = Tb
;           dPWM3 = Ta
;
;           }
;       }
; }
;
; void CalcTimes(void)
; {
;   T1 = PWM*T1
;   T2 = PWM*T2
;   Tc = (PWM-T1-T2)/2
;   Tb = Ta + T1
;   Ta = Tb + T2
; }
;*******************************************************************
;
```

```
       .include "general.inc"

; External references
       .include "Park.inc"
       .include "SVGen.inc"
       .include "CurModel.inc"
; Register usage
       .equ WorkW,        w1                      ; Working register
       .equ T1W,          w2
       .equ T2W,          w3

       .equ WorkDLoW,     w4                      ; double word (multiply results)
       .equ Vr1W,         w4
       .equ TaW,          w4
       .equ WorkDHiW,     w5                      ; double word (multiply results)
       .equ Vr2W,         w5
       .equ TbW,          w5
       .equ Vr3W,         w6
       .equ TcW,          w6

       .equ dPWM1,        PDC1
       .equ dPWM2,        PDC2
       .equ dPWM3,        PDC3
;=================== CODE =====================

       .section           .text
       .global            _CalcSVGen
       .global            CalcSVGen

 _CalcSVGen:
 CalcSVGen:
   ;; Get qVr1,qVr2,qVr3
       mov.w              _SVGenParm+SVGen_qVr1,Vr1W
       mov.w              _SVGenParm+SVGen_qVr2,Vr2W
       mov.w              _SVGenParm+SVGen_qVr3,Vr3W
   ;; Test Vr1
       cp0                Vr1W
       bra                LT,jCalcRef20         ; Vr1W < 0
   ;; Test Vr2
       cp0                Vr2W
       bra                LT,jCalcRef10         ; Vr2W < 0
   ;; Must be Sector 3 since Sector 7 not allowed
   ;; Sector 3: (0,1,1)  0-60 degrees
   ;; T1 = Vr2
   ;; T2 = Vr1
       mov.w              Vr2W,T2W
       mov.w              Vr1W,T1W
       rcall              CalcTimes
   ;; dPWM1 = Ta
   ;; dPWM2 = Tb
   ;; dPWM3 = Tc
       mov.w          TaW,dPWM1
       mov.w          TbW,dPWM2
       mov.w          TcW,dPWM3
       return
```

```
jCalcRef10:
   ;; Test Vr3
       cp0            Vr3W
       bra            LT,jCalcRef15         ; Vr3W < 0
   ;; Sector 5: (1,0,1)  120-180 degrees
   ;; T1 = Vr1
   ;; T2 = Vr3
       mov.w          Vr1W,T2W
       mov.w          Vr3W,T1W
       rcall          CalcTimes
   ;; dPWM1 = Tc
   ;; dPWM2 = Ta
   ;; dPWM3 = Tb
       mov.w          TcW,dPWM1
       mov.w          TaW,dPWM2
       mov.w          TbW,dPWM3
       return

jCalcRef15:
   ;; Sector 1: (0,0,1)  60-120 degrees
   ;; T1 = -Vr2
   ;; T2 = -Vr3
       neg.w          Vr2W,T2W
       neg.w          Vr3W,T1W
       rcall          CalcTimes
   ;; dPWM1 = Tb
   ;; dPWM2 = Ta
   ;; dPWM3 = Tc
       mov.w          TbW,dPWM1
       mov.w          TaW,dPWM2
       mov.w          TcW,dPWM3
       return

jCalcRef20:
   ;; Test Vr2
       cp0            Vr2W
       bra            LT,jCalcRef30         ; Vr2W < 0
   ;; Test Vr3
       cp0            Vr3W
       bra            LT,jCalcRef25         ; Vr3W < 0
   ;; Sector 6: (1,1,0)  240-300 degrees
   ;; T1 = Vr3
   ;; T2 = Vr2
       mov.w          Vr3W,T2W
       mov.w          Vr2W,T1W
       rcall          CalcTimes
   ;; dPWM1 = Tb
   ;; dPWM2 = Tc
   ;; dPWM3 = Ta
       mov.w          TbW,dPWM1
       mov.w          TcW,dPWM2
       mov.w          TaW,dPWM3
       return
jCalcRef25:
   ;; Sector 2: (0,1,0)  300-360 degrees
   ;; T1 = -Vr3
   ;; T2 = -Vr1
       neg.w          Vr3W,T2W
       neg.w          Vr1W,T1W
       rcall          CalcTimes

   ;; dPWM1 = Ta
   ;; dPWM2 = Tc
   ;; dPWM3 = Tb
       mov.w              TaW,dPWM1
```

```
       mov.w              TcW,dPWM2
       mov.w              TbW,dPWM3
       return
 jCalcRef30:
    ;; Must be Sector 4 since Sector 0 not allowed
    ;; Sector 4: (1,0,0)  180-240 degrees
    ;; T1 = -Vr1
    ;; T2 = -Vr2
       neg.w              Vr1W,T2W
       neg.w              Vr2W,T1W
       rcall              CalcTimes
    ;; dPWM1 = Tc
    ;; dPWM2 = Tb
    ;; dPWM3 = Ta
       mov.w              TcW,dPWM1
       mov.w              TbW,dPWM2
       mov.w              TaW,dPWM3
       return
;*******************************************************************
; CalcTimes
;
; void CalcTimes(void)
; {
;   T1 = PWM*T1
;   T2 = PWM*T2
;   Tc = (PWM-T1-T2)/2
;   Tb = Ta + T1
;   Ta = Tb + T2
; }
;
; Timing: 17instruction cycles
;*******************************************************************
 CalcTimes:

    ;; T1 = PWM*T1
    ;; Since T1 is in 1.15 and PWM in integer we do multiply by
    ;; 2*PWM*T1 as integers and use upper word of results
    ;; Load PWMPeriod
       sl.w               _SVGenParm+SVGen_iPWMPeriod,WREG  ; Mul PWM * 2 to allow for
                                                            : full range of voltage
       mul.us             w0,T1W,WorkDLoW
       mov.w              WorkDHiW,T1W
    ;; T2 = PWM*T2
       mul.us             w0,T2W,WorkDLoW
       mov.w              WorkDHiW,T2W
    ;; Tc = (PWM-T1-T2)/2
       ;mov.w             _SVGenParm+SVGen_iPWMPeriod,WorkW
       mov.w              _SVGenParm+SVGen_iPWMPeriod,WREG
       sub.w              w0,T1W,WorkW                      ;PWM-T1
       sub.w              WorkW,T2W,WorkW                   ; -T2
       asr.w              WorkW,WorkW                       ; /2
       mov.w              WorkW,TcW                         ; store Tc
    ;; Tb = Tc + T1
       add.w              WorkW,T1W,WorkW
       mov.w              WorkW,TbW
    ;; Ta = Tb + T2
       add.w              WorkW,T2W,WorkW
       mov.w              WorkW,TaW
       return
```

Trig.s

```
;*******************************************************************
; Trig
;
; Description:
; Calculate Sine and Cosine for specified angle using linear interpolation
; on a table of 128 words.
;
; This routine works the same for both integer scaling and 1.15 scaling.
;
; For integer scaling the Angle is scaled such that 0 <= Angle < 2*pi
; corresponds to 0 <= Ang < 0xFFFF. The resulting Sin and Cos
; values are returned scaled to -32769 -> 32767 i.e. (0x8000 -> 0x7FFF).
;
; For 1.15 scaling the Angle is scaled such that -pi <= Angle < pi
; corresponds to -1 -> 0.9999 i.e. (0x8000 <= Ang < 0x7FFF). The
; resulting Sin and Cos values are returned scaled to -1 -> 0.9999
; i.e. (0x8000 -> 0x7FFF).
;
; Functional prototype:
;   void SinCos( void )
;
;   On Entry:   ParkParm structure must contain qAngle
;   On Exit:    ParkParm will contain qSin, qCos.  qAngle is unchanged.
;
; Parameters:
;   Input arguments:
;       None
;   Return:
;       Void
;   SFR Settings required:
;       CORCON.IF = 0
;   Support routines required:
;       None
;   Local Stack usage:
;       0
;   Registers modified:
;       w0-w7
;   Timing:
;       About 28 instruction cycles
;*******************************************************************
;
        .include "general.inc"

;   External references
        .include "park.inc"
;   Constants
        .equ TableSize,128
;   Local register usage
        .equ Work0W,               w0             ; Working register
        .equ Work1W,               w1             ; Working register
        .equ RemainderW,           w2             ; Fraction for interpolation: 0->0xFFFF
        .equ IndexW,               w3             ; Index into table
        .equ pTabPtrW,             w4             ; Pointer into table
        .equ pTabBaseW,            w5             ; Pointer into table base
        .equ Y0W,                  w6             ; Y0 = SinTable[Index]
        .equ ParkParmW,            w7             ; Base of ParkParm structure

    ;; Note: RemainderW and Work0W must be even registers

;=================== LOCAL DATA =====================

        .section .ndata, "d"
 SinTable:
```



```
    .word 0,1608,3212,4808,6393,7962,9512,11039
    .word 12540,14010,15446,16846,18205,19520,20787,22005
    .word 23170,24279,25330,26319,27245,28106,28898,29621
    .word 30273,30852,31357,31785,32138,32413,32610,32728
    .word 32767,32728,32610,32413,32138,31785,31357,30852
    .word 30273,29621,28898,28106,27245,26319,25330,24279
    .word 23170,22005,20787,19520,18205,16846,15446,14010
    .word 12540,11039,9512,7962,6393,4808,3212,1608
    .word 0,-1608,-3212,-4808,-6393,-7962,-9512,-11039
    .word -12540,-14010,-15446,-16846,-18205,-19520,-20787,-22005
    .word -23170,-24279,-25330,-26319,-27245,-28106,-28898,-29621
    .word -30273,-30852,-31357,-31785,-32138,-32413,-32610,-32728
    .word -32767,-32728,-32610,-32413,-32138,-31785,-31357,-30852
    .word -30273,-29621,-28898,-28106,-27245,-26319,-25330,-24279
    .word -23170,-22005,-20787,-19520,-18205,-16846,-15446,-14010
    .word -12540,-11039,-9512,-7962,-6393,-4808,-3212,-1608

;=================== CODE ====================
        .section            .text
        .global             _SinCos
        .global             SinCos

_SinCos:
SinCos:
    ;; Base of qAngle, qSin, qCos group in ParkParm structure
        mov.w               #_ParkParm+#Park_qAngle,ParkParmW

    ;; Calculate Index and Remainder for fetching and interpolating Sin
        mov.w               #TableSize,Work0W
        mov.w               [ParkParmW++],Work1W       ; load qAngle & inc ptr to qCos
        mul.uu              Work0W,Work1W,RemainderW   ; high word in IndexW

    ;; Double Index since offsets are in bytes not words
        add.w               IndexW,IndexW,IndexW

    ;; Note at this point the IndexW register has a value 0x00nn where nn
    ;; is the offset in bytes from the TabBase.  If below we always
    ;; use BYTE operations on the IndexW register it will automatically
    ;; wrap properly for a TableSize of 128.

        mov.w               #SinTable,pTabBaseW        ; Pointer into table base

    ;; Check for zero remainder
        cp0.w               RemainderW
        bra                 nz,jInterpolate

    ;; Zero remainder allows us to skip the interpolation and use the
    ;; table value directly

        add.w               IndexW,pTabBaseW,pTabPtrW
        mov.w               [pTabPtrW],[ParkParmW++]   ; write qSin & inc pt to qCos

    ;; Add 0x40 to Sin index to get Cos index.  This may go off end of
    ;; table but if we use only BYTE operations the wrap is automatic.
        add.b               #0x40,IndexW
        add.w               IndexW,pTabBaseW,pTabPtrW
        mov.w               [pTabPtrW],[ParkParmW]     ; write qCos
        return

jInterpolate:

    ;; Get Y1-Y0 = SinTable[Index+1] - SinTable[Index]
        add.w               IndexW,pTabBaseW,pTabPtrW
        mov.w               [pTabPtrW],Y0W             ; Y0
        inc2.b              IndexW,IndexW              ; (Index += 2)&0xFF
```

```
        add.w               IndexW,pTabBaseW,pTabPtrW
        subr.w              Y0W,[pTabPtrW],Work0W     ; Y1 - Y0

    ;; Calcuate Delta = (Remainder*(Y1-Y0)) >> 16
        mul.us              RemainderW,Work0W,Work0W

    ;; Work1W contains upper word of (Remainder*(Y1-Y0))
    ;; *pSin = Y0 + Delta
        add.w               Work1W,Y0W,[ParkParmW++]  ; write qSin & inc pt to qCos

    ;; ================= COS =========================

    ;; Add 0x40 to Sin index to get Cos index.  This may go off end of
    ;; table but if we use only BYTE operations the wrap is automatic.
    ;; Actualy only add 0x3E since Index increment by two above
        add.b               #0x3E,IndexW
        add.w               IndexW,pTabBaseW,pTabPtrW

    ;; Get Y1-Y0 = SinTable[Index+1] - SinTable[Index]
        add.w               IndexW,pTabBaseW,pTabPtrW
        mov.w               [pTabPtrW],Y0W            ; Y0

        inc2.b              IndexW,IndexW         ; (Index += 2)&0xFF
        add.w               IndexW,pTabBaseW,pTabPtrW
        subr.w              Y0W,[pTabPtrW],Work0W  ; Y1 - Y0

    ;; Calcuate Delta = (Remainder*(Y1-Y0)) >> 16
        mul.us              RemainderW,Work0W,Work0W

    ;; Work1W contains upper word of (Remainder*(Y1-Y0))
    ;; *pCos = Y0 + Delta
        add.w               Work1W,Y0W,[ParkParmW] ; write qCos
         return
        .end
```


REVISION HISTORY

Revision A (June 2005)

Initial release of this document.

Revision B (October 2007)

This revision corrects the second equation in Figure 3.

NOTES:

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Trademarks

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Fuzhou
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Shunde
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

10/05/07