Developing an SRM Drive System Using the TMS320F240

APPLICATION REPORT: SPRA420

Michael T. DiRenzo
DSPS R&D Center

Digital Signal Processing Solutions
March 1998

TEXAS INSTRUMENTS

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Figures

Tables

Developing an SRM Drive System Using the TMS320F240

Abstract

This report describes the basic operation of switched reluctance motors (SRMs) and demonstrates how TMS320F240 DSP-based SRM drive from Texas Instruments™ can be used to achieve a wide variety of control objectives.

The first part of the report offers a detailed review of the operation and characteristics of SRMs. The advantages and disadvantages of this type of motor are cited.

The second part of the report provides examples and strategies for overcoming the limitations cited for SRMs. The examples have complete hardware and software details for developing an SRM drive system using the TMS320F240. Both conventional and position sensorless operations are described, along with the theoretical basis for designing the various control algorithms. The examples can be used as baseline designs, which can be easily modified to accommodate a specific application.

Product Support

Related Documentation

The following list specifies titles and literature numbers of corresponding TI documentation.

- *TMS320C24x DSP Controllers Reference Set, Volume 1: CPU, System, and Instruction Set*, Literature number SPRU160B
- *TMS320C24x DSP Controllers Reference Set, Volume 2: Peripheral Library and Specific Devices*, Literature number SPRU161B
- *TMS320C2xx User's Guide*, Literature number SPRU127B

World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Email

For technical issues or clarification on switching products, please send a detailed email to dsph@ti.com. Questions receive prompt attention and are usually answered within one business day.

Introduction

Electric machines can be broadly classified into two categories on the basis of how they produce torque – electromagnetically or by variable reluctance.

In the first category, motion is produced by the interaction of two magnetic fields, one generated by the stator and the other by the rotor. Two magnetic fields, mutually coupled, produce an electromagnetic torque tending to bring the fields into alignment. The same phenomenon causes opposite poles of bar magnets to attract and like poles to repel. The vast majority of motors in commercial use today operate on this principle. These motors, which include DC and induction motors, are differentiated based on their geometries and how the magnetic fields are generated. Some of the familiar ways of generating these fields are through energized windings, with permanent magnets, and through induced electrical currents.

In the second category, motion is produced as a result of the variable reluctance in the air gap between the rotor and the stator. When a stator winding is energized, producing a single magnetic field, reluctance torque is produced by the tendency of the rotor to move to its minimum reluctance position. This phenomenon is analogous to the force that attracts iron or steel to permanent magnets. In those cases, reluctance is minimized when the magnet and metal come into physical contact. As far as motors that operate on this principle, the switched reluctance motor (SRM) falls into this class of machines.

In construction, the SRM is the simplest of all electrical machines. Only the stator has windings. The rotor contains no conductors or permanent magnets. It consists simply of steel laminations stacked onto a shaft. It is because of this simple mechanical construction that SRMs carry the promise of low cost, which in turn has motivated a large amount of research on SRMs in the last decade. The mechanical simplicity of the device, however, comes with some limitations. Like the brushless DC motor, SRMs can not run directly from a DC bus or an AC line, but must always be electronically commutated. Also, the saliency of the stator and rotor, necessary for the machine to produce reluctance torque, causes strong non-linear magnetic characteristics, complicating the analysis and control of the SRM. Not surprisingly, industry acceptance of SRMs has been slow. This is due to a combination of perceived difficulties with the SRM, the lack of commercially available electronics with which to operate them, and the entrenchment of traditional AC and DC machines in the marketplace. SRMs do, however, offer some advantages along with potential low cost. For example, they can be very reliable machines since each phase of the SRM is largely independent physically, magnetically, and electrically from the other motor phases. Also, because of the lack of conductors or magnets on the rotor, very high speeds can be achieved, relative to comparable motors.

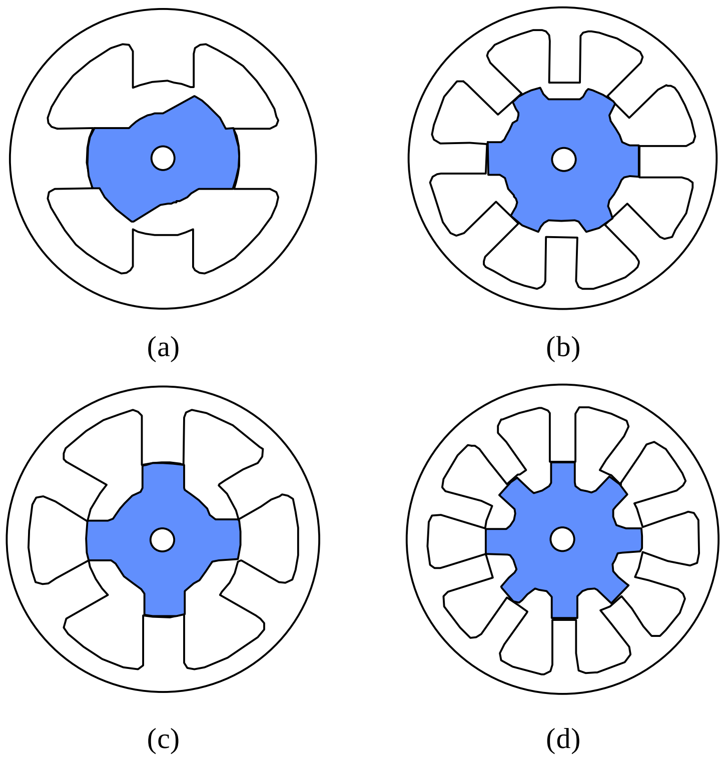
Disadvantages often cited for the SRM; that they are difficult to control, that they require a shaft position sensor to operate, they tend to be noisy, and they have more torque ripple than other types of motors; have generally been overcome through a better understanding of SRM mechanical design and the development of algorithms that can compensate for these problems.

Motor Characteristics

The basic operating principle of the SRM is quite simple; as current is passed through one of the stator windings, torque is generated by the tendency of the rotor to align with the excited stator pole. The direction of torque generated is a function of the rotor position with respect to the energized phase, and is independent of the direction of current flow through the phase winding. Continuous torque can be produced by intelligently synchronizing each phase's excitation with the rotor position.

By varying the number of phases, the number of stator poles, and the number of rotor poles, many different SRM geometries can be realized. A few examples are shown in Figure 1.

Figure 1. Various SRM Geometries



(a) 2-phase, 4 rotor poles/2 stator poles, (b) 4-phase, 8/6, (c) 3-phase, 6/4, (d) 5-phase, 10/8.

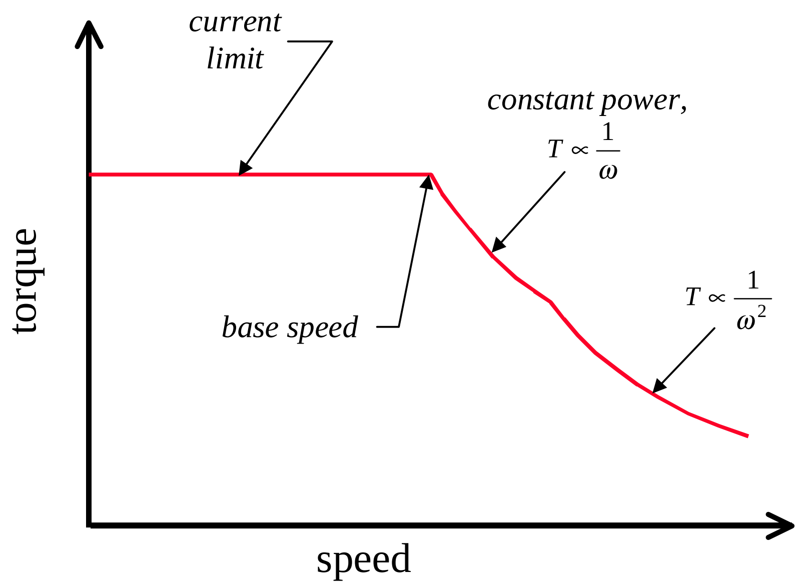
Note that although true of these examples, the number of phases is not necessarily equal to half the number of rotor poles.

Generally, increasing the number of SRM phases reduces the torque ripple, but at the expense of requiring more electronics with which to operate the SRM. At least two phases are required to guarantee starting, and at least three phases are required to insure the starting direction. The number of rotor poles and stator poles must also differ to insure starting.

Torque-Speed Characteristics

The torque-speed operating point of an SRM is essentially programmable, and determined almost entirely by the control. This is one of the features that makes the SRM an attractive solution. The envelope of operating possibilities, of course, is limited by physical constraints such as the supply voltage and the allowable temperature rise of the motor under increasing load. In general, this envelope is described by Figure 2.

Figure 2. SRM Torque-Speed Characteristics



Like other motors, torque is limited by maximum allowed current, and speed by the available bus voltage. With increasing shaft speed, a current limit region persists until the rotor reaches a speed where the back-EMF of the motor is such that, given the DC bus voltage limitation we can get no more current in the winding—thus no more torque from the motor. At this point, called the base speed, and beyond, the shaft output power remains constant, and at it's maximum. At still higher speeds, the back-EMF increases and the shaft output power begins to drop. This region is characterized by the product of torque and the square of speed remaining constant.

Electro-Magnetic Equations

Although SR motor operation appears simple, an accurate analysis of the motor's behavior requires a formal, and relatively complex, mathematical approach. The instantaneous voltage across the terminals of a single phase of an SR motor winding is related to the flux linked in the winding by Faraday's law,

$$v = iR_m + \frac{d\phi}{dt} \tag{1}$$

where, v is the terminal voltage, i is the phase current, R_m is the motor resistance, and ϕ is the flux linked by the winding. Because of the double salient construction of the SR motor (both the rotor and the stator have salient poles) and because of magnetic saturation effects, in general, the flux linked in an SRM phase varies as a function of rotor position, θ, and the motor current. Thus (1) can be expanded as

$$v = iR_m + \frac{\partial\phi}{\partial i}\frac{di}{dt} + \frac{\partial\phi}{\partial\theta}\frac{d\theta}{dt} \tag{2}$$

where, $\frac{\partial\phi}{\partial i}$ is defined as $L(\theta, i)$, the instantaneous inductance, $\frac{\partial\phi}{\partial\theta}$ is $K_b(\theta, i)$, the instantaneous back EMF.

General Torque Equation

Equation (2) governs the transfer of electrical energy to the SRM's magnetic field. In this section, the equations which describe the conversion of the field's energy into mechanical energy are developed. Multiplying each side of (1) by the electrical current, i, gives an expression for the instantaneous power in an SRM,

$$vi = i^2R_m + i\frac{d\phi}{dt} \tag{3}$$

The left-hand side of (3) represents the instantaneous electrical power delivered to the SRM. The first term in the right-hand side (RHS) of (3) represents the ohmic losses in the SRM winding. If power is to be conserved, then the second term in the RHS of (3) must represent the sum of the mechanical power output of the SRM and any power stored in the magnetic field. Thus,

$$i\frac{d\phi}{dt} = \frac{dW_m}{dt} + \frac{dW_f}{dt} \tag{4}$$

where, $\frac{dW_m}{dt}$ is the instantaneous mechanical power, and $\frac{dW_f}{dt}$ is the instantaneous power, which is stored in the magnetic field. Because power, by it's own definition, is the time rate of change of energy, W_m is the mechanical energy and W_f is the magnetic field energy.

It is well known that mechanical power can be written as the product of torque and speed,

$$\frac{dW_m}{dt} = T\omega = T\frac{d\theta}{dt} \tag{5}$$

where, T is torque, and $\omega = \frac{d\theta}{dt}$ is the rotational velocity of the shaft.

Substitution of (5) into (4) gives,

$$i\frac{d\phi}{dt} = T\frac{d\theta}{dt} + \frac{dW_f}{dt} \tag{6}$$

and solving (6) for torque yields the equation,

$$T(\theta,\phi) = i(\theta,\phi)\frac{d\phi}{d\theta} - \frac{dW_f(\theta,\phi)}{d\theta} \tag{7}$$

and for constant flux, (7) simplifies to,

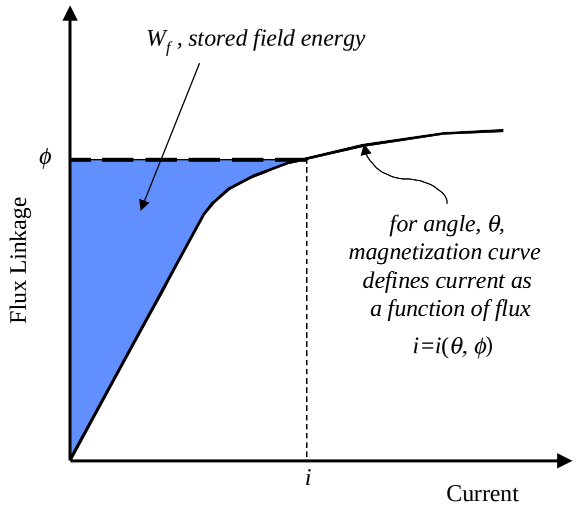
$$T = -\frac{\partial W_f}{\partial \theta} \tag{8}$$

Since it is often desirable to express torque in terms of current rather than flux, it is common to express torque in terms of co-energy, W_c, instead of energy. To introduce the concept of co-energy, first consider a graphical interpretation of field energy. For constant shaft angle, $\frac{d\theta}{dt} = 0$, integration of (6) shows that the magnetic field energy can be given by the equation,

$$W_f = \int_0^{\phi} i(\theta,\phi)d\phi \tag{9}$$

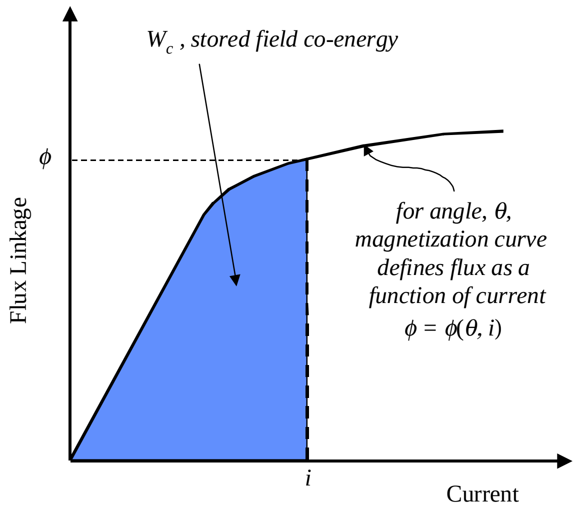
and graphically by the shaded area in Figure 3.

Figure 3. Graphical Interpretation of Magnetic Field Energy


W_f, stored field energy
ϕ
Flux Linkage
for angle, θ, magnetization curve defines current as a function of flux
$i=i(\theta, \phi)$
i
Current

Now, consider Figure 4.

Figure 4. Graphical Interpretation of Magnetic Field Co-Energy


W_c, stored field co-energy
ϕ
Flux Linkage
for angle, θ, magnetization curve defines flux as a function of current
$\phi = \phi(\theta, i)$
i
Current

For the fixed angle, θ, let the magnetization curve define flux as a function of current, instead of current defined as a function of flux. The shaded area below the curve,

$$W_c = \int_0^i \phi(\theta, i)di \tag{10}$$

is defined as the magnetic field co-energy.

From Figure 3 and Figure 4, we see that the area defining the field energy and co-energy can be described by the relation,

$$W_c + W_f = i\phi \tag{11}$$

Differentiating both sides of (11) yields

$$dW_c + dW_f = \phi di + id\phi \tag{12}$$

Solving for the differential field energy in (12) and substituting back into (7) gives,

$$T = \frac{id\phi - \left(\phi di + id\phi - dW_c(\theta, i)\right)}{d\theta} \tag{13}$$

For simplification, the general torque equation, (13), is usually simplified for values of constant current. The differential co-energy can be written in terms of its partial derivatives as,

$$dW_c(\theta, i) = \frac{\partial W_c}{\partial \theta} d\theta + \frac{\partial W_c}{\partial i} di \tag{14}$$

From (13) and (14), it is fairly easy to show that under constant current,

$$T = \frac{\partial W_c}{\partial \theta}, \quad i \text{ constant} \tag{15}$$

Simplified Torque Equation

Often, SRM analysis proceeds under the assumption that, magnetically, the motor remains unsaturated during operation. This assumption can be useful for “first cut” control designs or performance predictions. When magnetic saturation is neglected, the relationship from flux to current is given by,

$$\phi = L(\theta) \cdot i \tag{16}$$

and the motor inductance varies only as a function of rotor angle. Substituting (16) into (10) and evaluating the integral yields,

$$W_c = \frac{i^2}{2} L(\theta) \qquad (17)$$

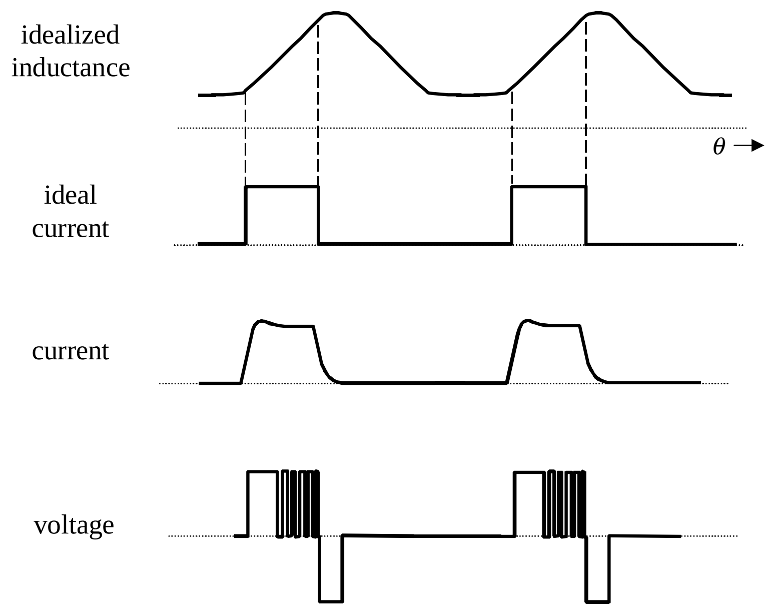
and then substituting (17) into (15) gives the familiar simplified relationship for SRM torque,

$$T = \frac{i^2}{2} \frac{dL}{d\theta} \qquad (18)$$

Control

SRM drives are controlled by synchronizing the energization of the motor phases with the rotor position. Figure 5 illustrates the basic strategy.

Figure 5. Basic Operation of a Current Controlled SRM – Motoring at Low-Speed



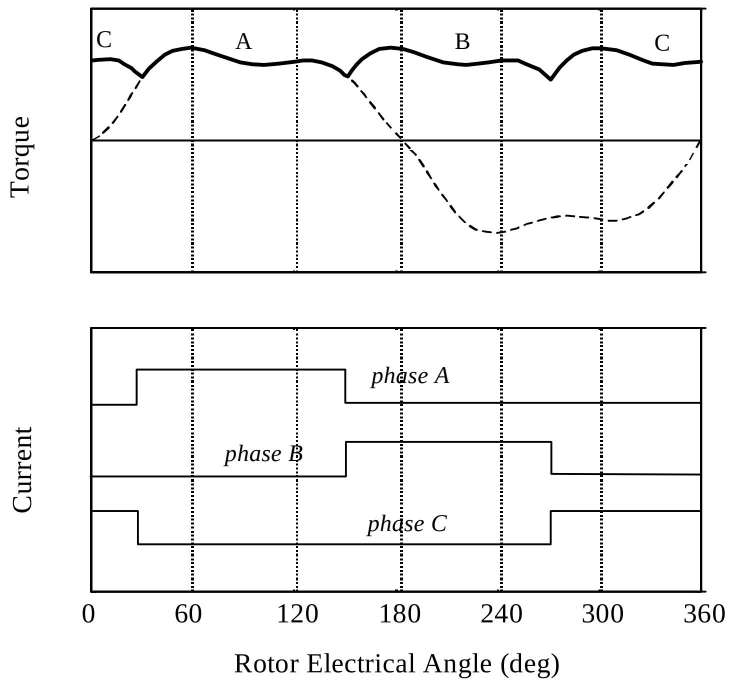
As (18) suggests, positive (or motoring) torque is produced when the motor inductance is rising as the shaft angle is increasing, $\frac{dL}{d\theta} > 0$.

Thus, the desired operation is to have current in the SRM winding during this period of time. Similarly, a negative (or braking) torque is produced by supplying the SRM winding with current while $\frac{dL}{d\theta} < 0$.

The exact choice of the turn-on and turn-off angles and the magnitude of the phase current, determine the ultimate performance of the SRM. The design of commutation angles, sometimes called firing angles, usually involves the resolution of two conflicting concerns – maximizing the torque output of the motor or maximizing the efficiency of the motor. In general, efficiency is optimized by minimizing the dwell angle (the dwell angle is the angle traversed while the phase conducts), and maximum torque is achieved by maximizing the dwell angle to take advantage of all potential torque output from a given phase.

A simple and effective commutation scheme is depicted in Figure 6.

Figure 6. Commutation of a 3-Phase SRM



In the top plot of Figure 6, the dashed line shows the torque that would be generated by phase A, should constant current flow through the phase winding during an entire electrical cycle of the SRM. With the idealized current waveforms of the figure, the resulting net torque from the motor is shown by the solid line. The turn-on and turn-off angles coincide with the region where maximum torque is obtained for the given amount of phase current.

This commutation sequence tends to optimize efficiency. Here, a dwell angle of 120 electrical degrees is used, which is the minimum dwell angle that can be used for a three-phase SRM, without regions of zero torque.

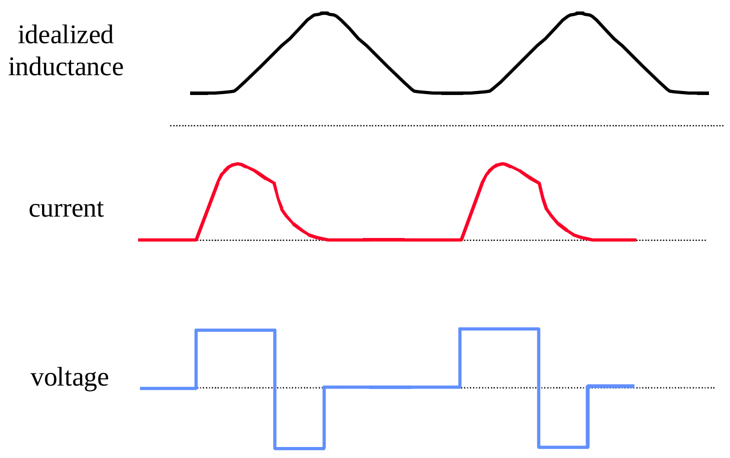
Of interest to note from Figure 6 is that constant current results in non-constant torque. As might be expected, schemes have been proposed by Husain and Ehsani[1], Ilic-Spong, *et al*[2], and Kjaer, *et al*[3] that attempt to linearize SRM output torque by shaping and controlling the phase currents through some non-linear function that depends upon the motor characteristics. This application, although not covered in this report, is well suited for DSP implementation.

Figure 6 illustrates the effect that the choice of commutation angles can have upon the SRM performance. Equally important is the magnitude of the current that flows in the winding. Commonly, the phase current is sensed and controlled in a closed-loop manner, and as seen in the voltage curve of Figure 5, the control is typically implemented using PWM techniques.

SRM control is often described in terms of "low-speed" and "high-speed" regimes. Low-speed operation is typically characterized by the ability to arbitrarily control the current to any desired value. Figure 5 illustrates waveforms typical of low-speed SRM operation. As the motor's speed increases, it becomes increasingly difficult to regulate the current because of a combination of the back EMF effects and a reduced amount of time for the commutation interval. Eventually a speed is reached where the phase conducts (remains on) during the entire commutation interval. This mode of operation, depicted by Figure 7, is called the single-pulse mode.

[1] I. Husain and M. Ehsani, " Torque Ripple Minimization in Switched Reluctance Motor Drives by PWM Current Control," *Proc. APEC'94*, 1994, pp. 72-77.
[2] M. Ilic-Spong, T. J. E. Miller, S. R. MacMinn, and J. S. Thorp, "Instantaneous Torque Control of Electric Motor Drives," *IEEE Trans. Power Electronics*, Vol. 2, pp. 55-61, Jan. 1987.
[3] P. C. Kjaer, J. Gribble, and T. J. E. Miller, "High-grade Control of Switched Reluctance Machines," *IEEE Trans. Industry Electronics*, Vol. 33, pp. 1585-1593, Nov. 1997.

Figure 7. Single-Pulse Mode – Motoring, High Speed



When this occurs, the motor speed can be increased by increasing the conduction period (a greater dwell angle) or by advancing the firing angles, or by a combination of both. By adjusting the turn-on and turn-off angles so that the phase commutation begins sooner, we gain the advantage of producing current in the winding while the inductance is low, and also of having additional time to reduce the current in the winding before the rotor reaches the negative torque region. Control of the firing angles can be accomplished a number of ways, and is based on the type of position feedback available and the optimization goal of the control, as discussed in publications by Becerra, *et al*,[4] and Miller.[5] When position information is more precisely known, a more sophisticated approach can be used. One approach is to continuously vary the turn-on angle with a fixed dwell.

Near turn-on, (2) can be approximated as

$$v = \frac{\partial \phi}{\partial i}\frac{di}{dt} = L_u \cdot \frac{di}{dt} \tag{19}$$

Multiplying each side of (19) by the differential, $d\theta$, and solving for $d\theta$, gives,

$$d\theta = \frac{L_u \cdot di}{v} \cdot \frac{d\theta}{dt} \tag{20}$$

[4] R. Becerra, M. Ehsani, and T. J. E. Miller, “Commutation of SR Motors,” *IEEE Trans. Power Electronics*, Vol. 8, July 1993, pp. 257-262.
[5] T. J. E. Miller, “Switched Reluctance Motors and Their Control,” Magna Physics Publishing, Hillsboro, OH, and Oxford, 1993.

and using first order approximations yields an equation for calculating advance angle,

$$\theta_{adv} = \frac{L_u \cdot i_{cmd}}{V_{bus}} \cdot \omega \tag{21}$$

where i_{cmd} is the desired phase current and V_{bus} is the DC bus voltage.

Example – SRM Drive with Position Feedback

This section describes an example application of an SRM drive with position feedback. The SRM is a 3-phase 12/8 machine that is speed and current controlled.

Hardware Description

SRM Characteristics

The characteristics of the SRM used in this application report are given by Table 1.

Table 1. SRM Parameters

number of phases, m	3
number of stator poles, N_S	12
number of rotor poles, N_R	8
nominal phase resistance, R_m	8.1 Ω
nominal aligned inductance, L_a	240 mH
nominal unaligned inductance, L_u	60 mH
phase current (max)	4 A
DC bus voltage, V_{bus}	170 VDC

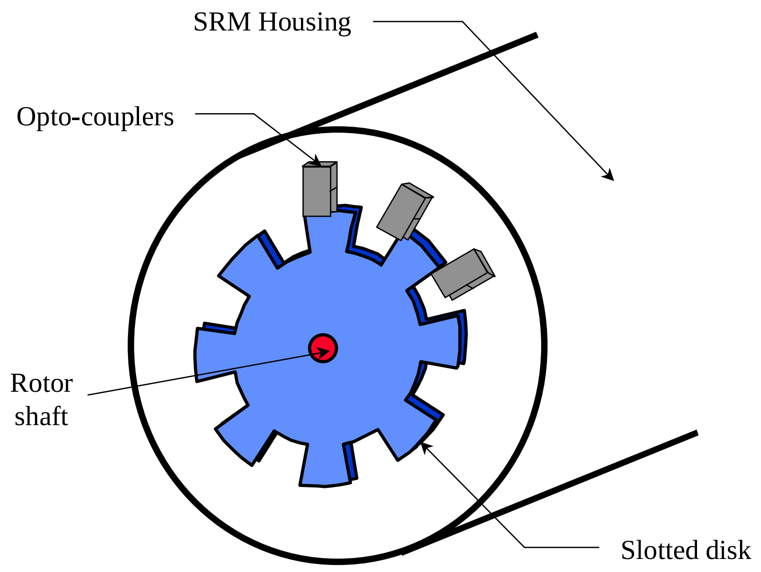
Control Hardware

The control hardware used in this application report is the TMS320F240 evaluation module (EVM).

Position Sensor

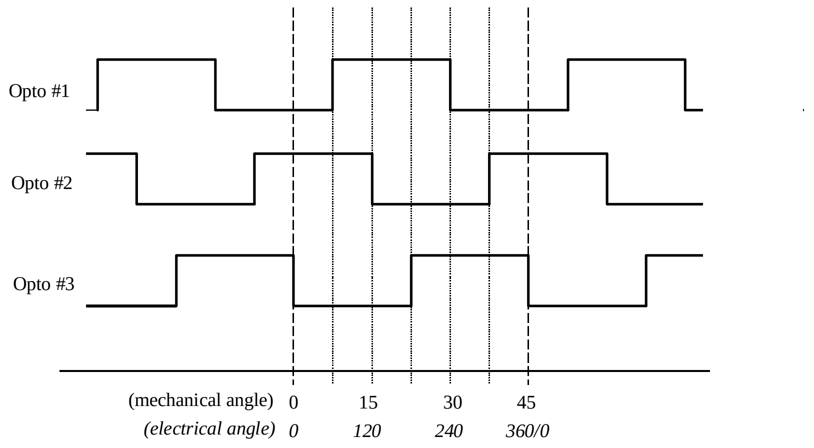
Shaft position information is provided using an 8-slot, slotted disk connected to the rotor shaft and three opto-couplers mounted to the stator housing as shown in Figure 8.

Figure 8. SRM Shaft Position Sensor



The opto-couplers are nominally located 30° apart from each other along the circumference of the disk. This configuration and geometry produces the output waveforms shown in Figure 9.

Figure 9. Opto-Coupler Output Signals vs. Rotor Angle

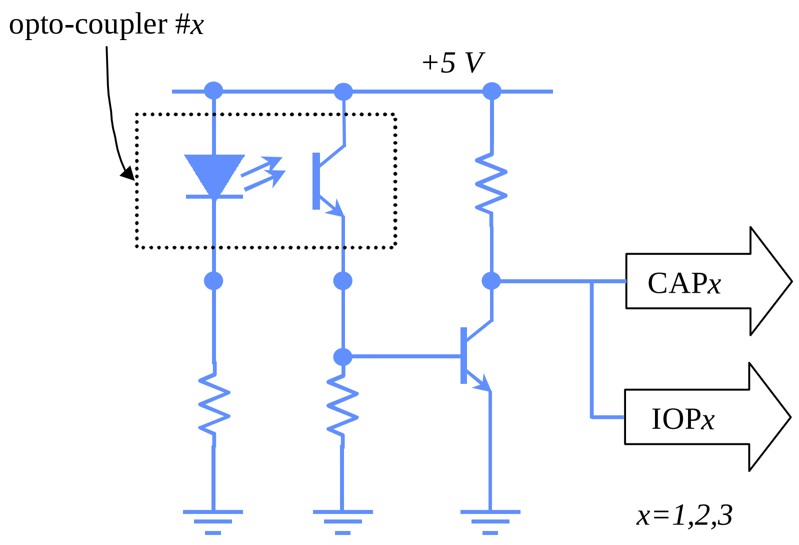


This configuration generates an opto-coupler edge for every 7.5° of mechanical rotation. For every 45° of mechanical rotation the signal pattern repeats, corresponding to one electrical cycle of the SRM, of which there are 8 per shaft revolution.

In this report both mechanical angle and electrical angle are referenced. Mechanical angle is useful when considering velocity control of the SRM, and electrical angle is convenient when considering commutation. Electrical angle is related to mechanical angle by the number of rotor poles, N_R. In Figure 9, the angles are arbitrarily defined with respect to some convenient point. Here, 180° electrical is defined as *the aligned position for phase A of the motor*. This is easily verified by energizing phase A and then monitoring the opto-coupler output waveforms on an oscilloscope to observe that the rotor is at the point where opto-coupler #3 switches state, while opto-coupler #2 is low and opto-coupler #1 is high. For a 3-phase SRM, phases B and C are related to the position of phase A by adding 120 and 240 electrical degrees, respectively.

A fundamentally identical position sensor can be implemented by replacing the opto-couplers with Hall-effect sensors and embedding permanent magnets within the teeth of the slotted disk. The opto-couplers are connected to the F240 EVM as shown in Figure 10.

Figure 10. Opto-Coupler Connections to the TMS320F240 EVM



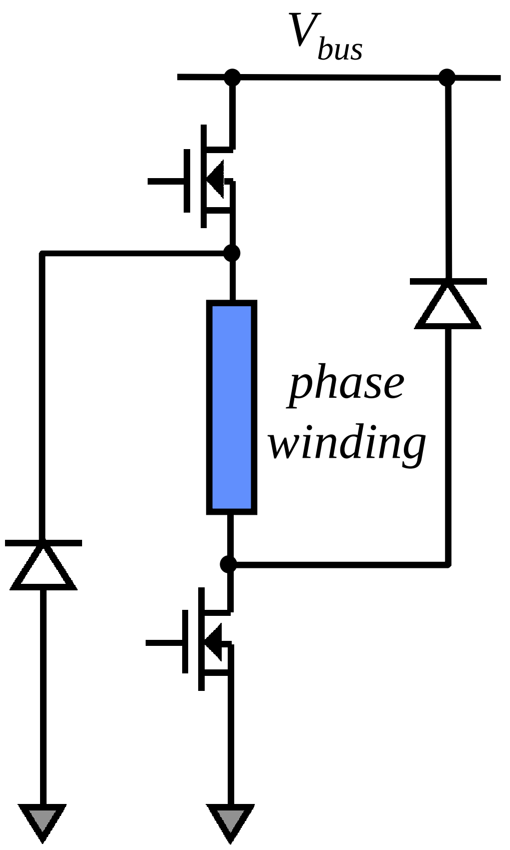
Here, each opto-coupler output is connected to both a capture input and a digital I/O input. As will be explained in further detail below, the capture inputs are used once the motor is running, and the digital I/O inputs are used for estimating initial rotor position and for starting the SRM.

Power Electronics Hardware

The amount of current flowing through the SRM windings is regulated by switching on or off power devices, such as MOSFETs or IGBTs, which connect each SRM phase to a DC bus. The power inverter topology is an important issue in SRM control because it largely dictates how the motor can be controlled.

There are numerous options available, and invariably the decision will come down to trading off the cost of the driver components against having enough control capability (independent control of phases, current feedback, etc.) built into the driver. A popular configuration, and the one used in this application report, uses 2 switches and 2 diodes per phase. This topology is depicted in Figure 11.

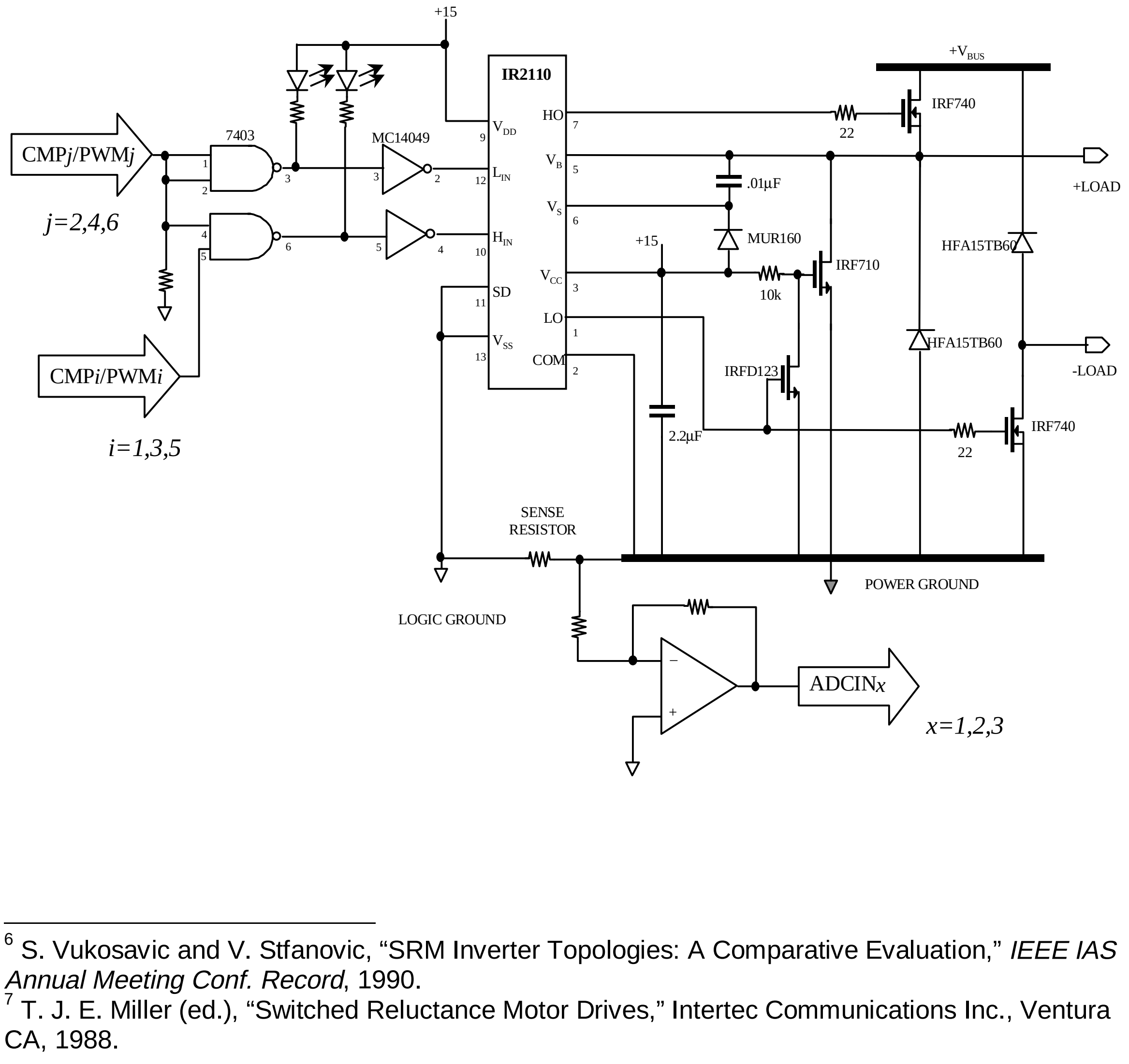
Figure 11. Two-Switch Per Phase Inverter



Publications by Vukosavic and Stfanovic[6] and Miller[7] offer several other configurations that require fewer switches per phase, although with some penalty on control flexibility and maintaining phase independence. A gate drive IC device, such as the IR2110, is used to turn on and off the semiconductor switches. In the topology of Figure 11, the low-side switch is usually held on during a commutation interval, while the top switch is used to implement the control. For independent current control of each phase, a low-ohm sense resistor is placed between the source of the low-side n-channel power MOSFET and ground.

A schematic diagram of the inverter used in this application report, including the gate drive circuit and the connections to the EVM, is given in Figure 12.

Figure 12. Schematic Diagram of SRM Inverter Using the IR2110 and Connections to the EVM



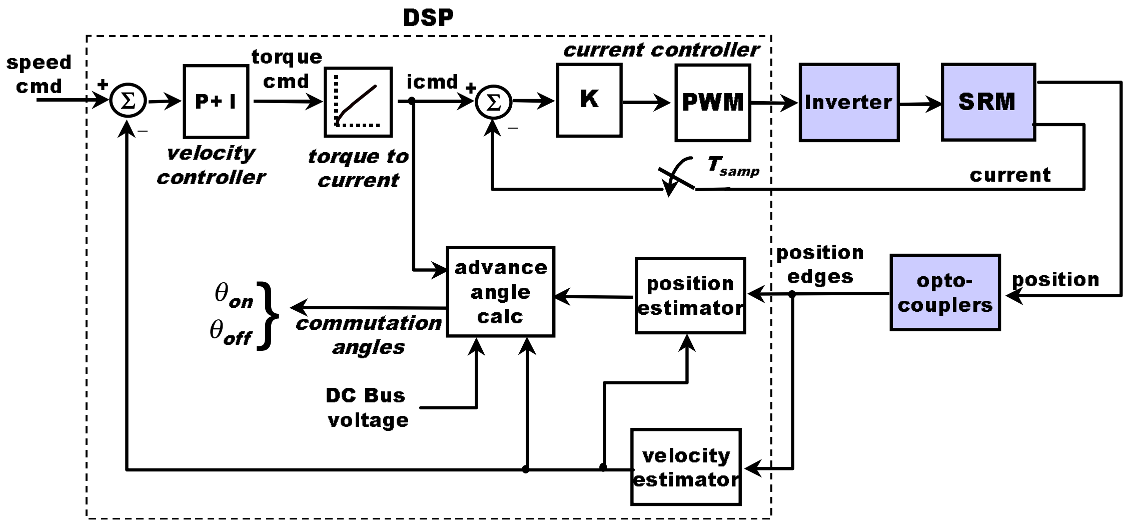
[6] S. Vukosavic and V. Stfanovic, “SRM Inverter Topologies: A Comparative Evaluation,” *IEEE IAS Annual Meeting Conf. Record*, 1990.
[7] T. J. E. Miller (ed.), “Switched Reluctance Motor Drives,” Intertec Communications Inc., Ventura CA, 1988.

The diagram shows the components used for a single phase. Each phase uses two IRF740 n-channel power MOSFETs for the switching elements in the output stage. The IRF740 is rated at 400 VDC, 10 A. The drain to source on resistance of these devices is 0.55 Ω. The free-wheeling diodes used in the power stage are HFA15TB60s, fast recovery diodes. The HFA15TB60 has a reverse recovery time of 60 ns, and is rated at 600 VDC, 15 A. Logic is implemented at the input to the gate drive IC such that the top power MOSFET can be turned on only when the bottom MOSFET is also on. The reasons for this limitation, and other circuit details, are discussed more thoroughly in a publication by Clemente and Dubhashi.[8]

Software Description

The software described in this application report is written in C and is designed for operating a 3-phase 12/8 SRM in closed loop current control and closed loop speed control. A block diagram of the algorithms implemented is given in Figure 13.

Figure 13. Block Diagram of the SRM Controller.



Velocity is estimated by monitoring the elapsed time between opto-coupler edges, which are a known distance apart. A velocity compensation algorithm determines the torque required to bring the motor velocity to the commanded value.

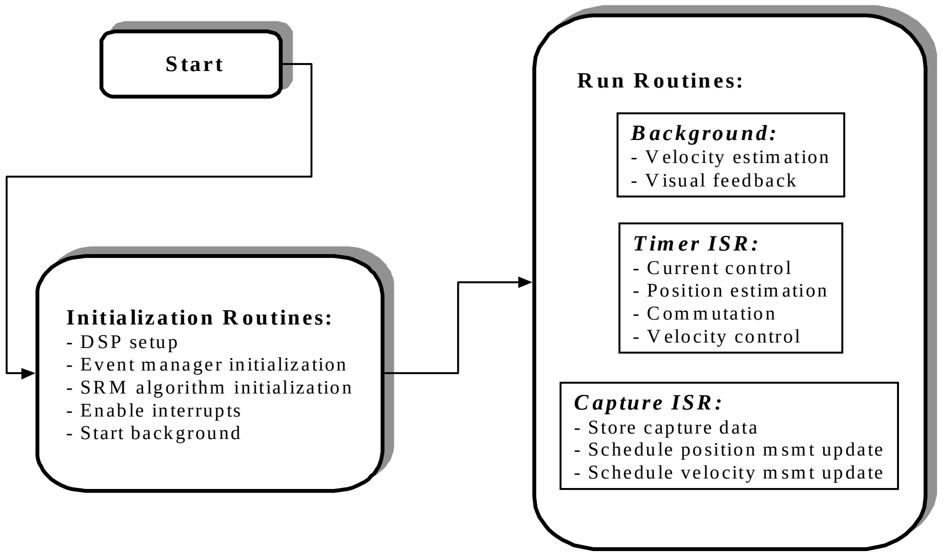
[8] S. Clemente and A. Dubhashi, "HV Floating MOS-Gate Driver IC," *International Rectifier Application Note AN-978A*, International Rectifier, El Segundo, CA, 1990.

A commutation algorithm converts the torque command into a set of phase current commands, and the current in each phase is individually regulated using a fixed-frequency PWM scheme. Further details on each of the algorithms are provided in subsequent sections of this report.

Program Structure

Figure 14 shows the structure of the SRM control software for the TMS320F240 DSP.

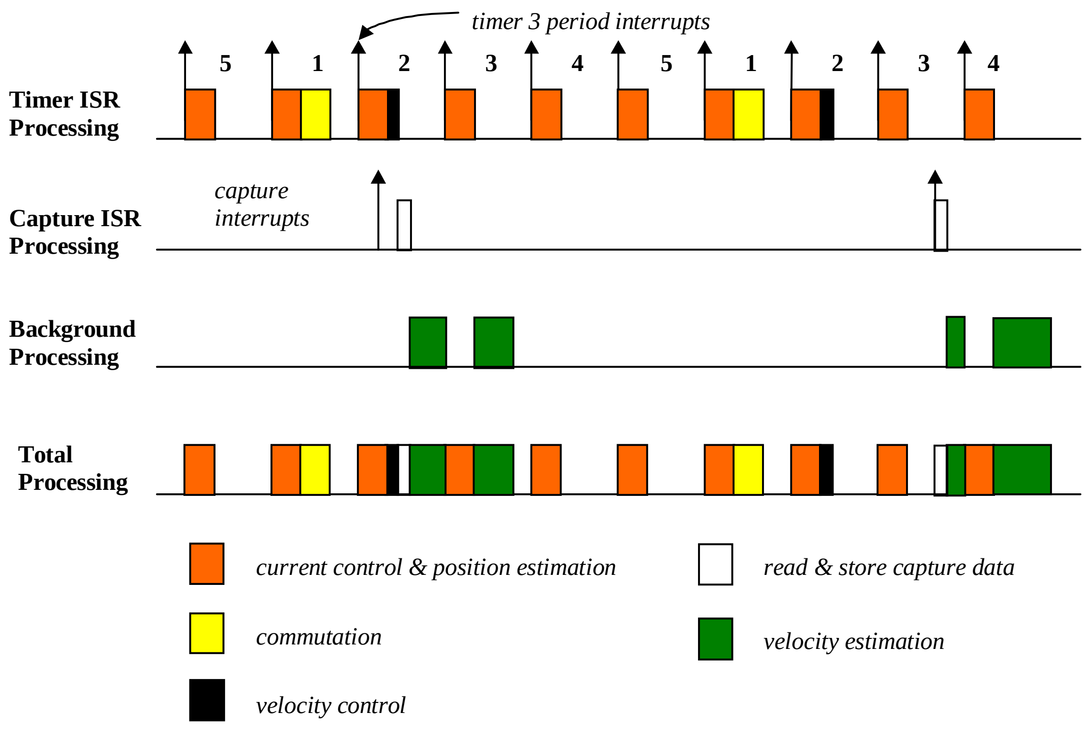
Figure 14. TMS320F240 SRM Control Program Structure



At the highest level, the software consists of initialization routines and run routines. Upon completion of the necessary initialization, the background task is started. The background is simply an infinite loop, although when required, lower priority processing including velocity estimation and a visual feedback routine is executed. The velocity estimation involves double-precision division arithmetic, thus it is executed in background mode so that the timeline is not violated. This algorithm is initiated in the capture interrupt service routine. The visual feedback function simply toggles an LED on the EVM board to provide a signal to the user that the code is running.

All of the time critical motor control processing is done via interrupt service routines. The timer ISR is executed at each occurrence of the maskable CPU interrupt INT3. This interrupt corresponds to the event manager group B interrupts, of which we enable only the timer #3 period interrupt, TPINT3. The frequency, F, at which this routine is executed is specified by loading the timer 3 period register with the desired value. The SRM control algorithms which are implemented during the timer ISR are the current control, shaft position estimation, commutation, and velocity control. As illustrated in Figure 15, only the current control and shaft position estimation are executed at the frequency, F.

Figure 15. Processor Timeline Showing Typical Loading and Execution of SRM Control Algorithms



Because of their lower bandwidth requirements, velocity control and commutation are performed at a frequency of $F/5$. Considering the timer ISR as being sliced into fifths with a pattern repeating every five slices, commutation is run only in the 1st slice and the velocity loop only in the 2nd. Current control and position estimation are performed in each slice.

The capture interrupt service routine is executed at each occurrence of the maskable CPU interrupt INT4. This CPU interrupt corresponds to the event manager group C interrupts, of which we enable the three capture event interrupts, CAPINT1-3. This ISR executes asynchronously to the timers on board the DSP and the frequency of execution is dependent on the SRM shaft speed according to the equation,

$$\text{capture ISR frequency (Hz)} = \text{shaft speed (rpm)} \times \frac{360\,(\text{deg})}{(\text{rev})} \times \frac{1}{7.5\,(\text{deg})} \times \frac{1\,(\text{min})}{60\,(\text{sec})} \tag{22}$$

The capture ISR is used to determine which capture interrupt has occurred, read the appropriate capture FIFO register, and then store the data. Although no algorithm is explicitly executed in this ISR, flags are set which initiate velocity and position estimation actions. As described above, the velocity estimate update calculation is performed in the background. The position estimation algorithm, which executes during the timer ISR, is notified that a new position measurement has been received.

Table 2 summarizes the processing requirements for each of the major software functions for the SRM controller.

Table 2. Benchmark Data for the Various SRM Drive Software Modules

S/W Block	Module	#cycles	execution time @ 50 ns	execution frequency	relative time @ 5 kHz
velocity estimation	background	3620	181.0 µs	800 Hz [1]	29.0 µs
visual feedback	background	60	3.0 µs	2 Hz	0.0 µs
current control	timer ISR	948	47.4 µs	5000 Hz	47.4 µs
position estimation	timer ISR	258	12.9 µs	5000 Hz	12.9 µs
commutation	timer ISR	1296	64.8 µs	1000 Hz	13.0 µs
velocity control	timer ISR	444	22.2 µs	1000 Hz	4.4 µs
misc. overhead	timer ISR	140	7.0 µs	5000 Hz	7.0 µs
capture ISR	capture ISR	500	25.0 µs	800 Hz[1]	4.0 µs
C context switch	RTS.LIB	120	6.0 µs	5800 Hz	7.0 µs
Total					124.7 µs

[1] at a shaft speed of 1000 rpm.

The data in Table 2 shows that when the timer ISR frequency, F, is chosen as 5 kHz, that the overall processor loading is equal to

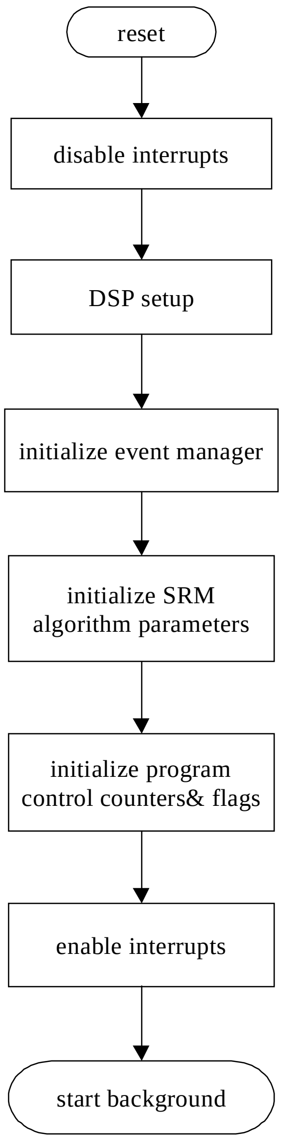
$$\text{processor loading} = \frac{124.7\ \mu s}{200.0\ \mu s} = 62.4\%\,, \qquad (23)$$

when running the DSP at a 20 MHz clock frequency. The code size is 2456 words and 167 words are required for variable/data storage. Thus, the total memory requirement is less than 3 K words. Complete code listings are given in Appendix A.

Initialization Routines

A flowchart describing the initialization routines is given in Figure 16.

Figure 16. Initialization Flowchart



The DSP is configured so that the watchdog timer is disabled. The TMS320F240 EVM has a 10 MHz crystal, which is used in conjunction with the PLL module of the DSP to yield a 20 MHz CPUCLK.

The event manager initialization configures the timer units, the capture units, the compare units, and the A/D converters. Also, the CAP1-CAP4 and IOPB0-IOPB3 pins, whose functions are software programmable, are configured to operate as capture pins and digital output pins, respectively.

Each of the timers are programmed to operate in the continuous up count mode. Timer #1 provides the timebase for the fixed-frequency PWM control of the phase current. Timer #2 provides the timebase for the capture events, and timer #3 is used to provide a CPU interrupt at a fixed rate. The compare units are configured to the PWM mode, where PWMs 1,3, and 5 (used for switching the high-side power MOSFET) are configured as active high.

The SRM algorithm initialization defines the parameters of the position estimation state machine and sets the initial conditions of the motor, for example, setting the shaft velocity estimate to zero. Also, during this routine, the logic states of the opto-couplers are read from the digital I/O pins, and this information used to estimate the rotor position.

Upon initializing several flags and counters which are used for program flow control, the infinite loop background routine is called, and the normal operation of controlling the SRM drive begins.

See the comments in the code listings found in Appendix A for further information on the program initialization.

Current Controller

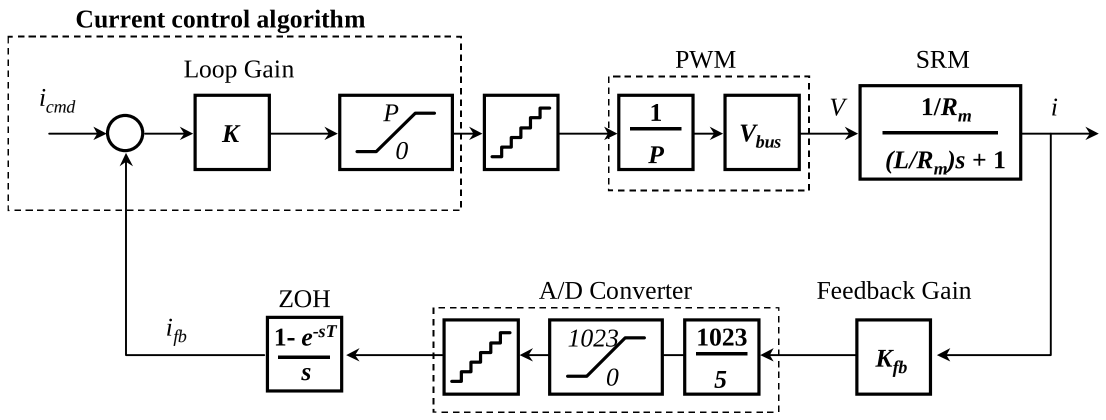
Current is regulated by fixed-frequency PWM signals with varying duty cycles. The TMS320F240 accomplishes this using compare units and output logic circuits. The compare units are programmed for PWM mode, to use timer #1 as a time base. The desired output logic polarity is controlled by the ACTR register. The PWM frequency is specified by loading the period register of timer #1, T1PER, with a value, P, defined by,

$$P = \frac{CPUCLK\ frequency}{PWM\ frequency} - 1 \tag{24}$$

For the F240, the CPUCLK frequency is 20 MHz. The percentage duty cycle for the x^{th} phase is controlled by loading the appropriate compare register, CMPR*x*, with an appropriate value between 0 and P (0 = 0%, P/2 = 50%, P = 100%). A PWM frequency of 20 kHz is used. The value is significantly higher than the bandwidth of the current loop and also at a frequency which is inaudible.

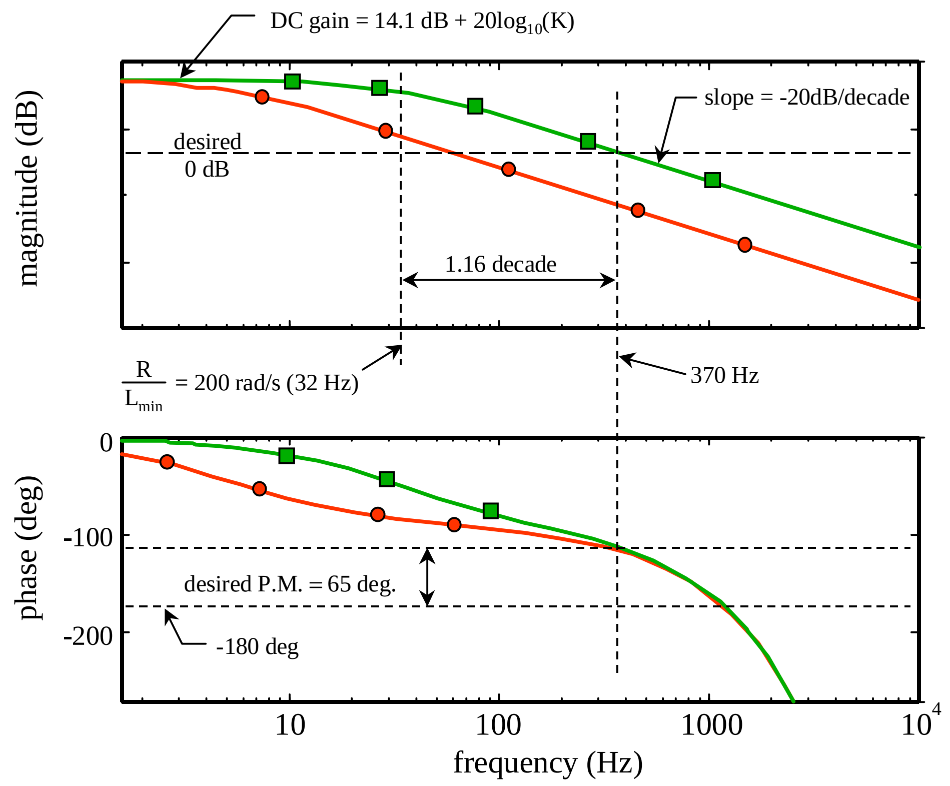
The percentage duty cycle command is calculated by the current loop compensation algorithm, which is designed using linear analysis. The analysis begins with an approximate model of the current loop, given by Figure 17.

Figure 17. Approximate SRM Current Loop Model.



Using the SRM data of , and with P = 999, K_{fb} = 1.17 V/A, the open-loop frequency response, $G(\omega)$, of the SRM current loop, from i_{cmd} to i_{fb}, is given in Figure 18 for values of phase inductance at both the aligned rotor position ($L = L_a$) and the unaligned rotor position ($L = L_u$).

Figure 18. Frequency Response Plots for the SRM Current Loop at the Unaligned Position (Squares) and at the Aligned Position (Circles)



Because of the digital implementation of the current loop, additional phase loss, beyond the 90° due to the motor pole, is contributed by the sample and hold process and the processing delay inherent in the loop. These dynamics essentially limit the current loop bandwidth to an open loop crossover frequency near 370 Hz. The time delay due to the zero-order hold (ZOH) is equal to ½ of the sampling period, in this case ½ of 200 µsec, or 100 µsec. Since the phase loss at any frequency, ω, due to a pure time delay, τ, is given by the expression,

$$\theta_{loss} = \omega\tau \tag{25}$$

using (25) we calculate that the phase loss due to the ZOH sampling at 370 Hz is equal to

$$\theta_{loss} = 2\pi \times 370 \times (100 \times 10^{-6}) = 0.232 \text{ rad} = 13.3^{\circ} \tag{26}$$

Assuming that the processing delay is equal to 50% of a loop cycle, or another 100 µsec, then the net effect of digital implementation yields about 26° of phase loss at 370 Hz. When combined with the 90° due to the motor pole, the phase loss through the loop is approximately 116°, at 370 Hz. If the loop gain, K, is chosen such that the 0 dB point of the open-loop magnitude occurs at 370 Hz, then the resulting phase margin in the loop will be about 64°. This amount of phase margin provides a very stable loop design. The DC gain of the loop is given by,

$$\frac{K \cdot V_{bus} \cdot K_{fb} \cdot 1023}{P \cdot 5 \cdot R} = 5.092K \tag{27}$$

which when written in decibels is equal to,

$$\text{DC gain} = 14.1\,dB + 20\log_{10}(K) \tag{28}$$

For frequencies where $\omega > (R/L)$, the magnitude of the loop response is equal to,

$$\left|G(\omega)\right| = 14.1\,dB + 20\log_{10}(K) - 2 \times \left(\frac{\omega}{R/L}\right) \tag{29}$$

In (29), letting $L = L_u$ provides the most conservative choice, resulting in a stable design for all rotor positions. Setting the left-hand side of (29) to 0 dB, while $\omega = 2\pi(370)$ rad/s, and solving for K, yields the value of K which insures the desired open-loop crossover point for the current loop. In this case $K = 2.8$.

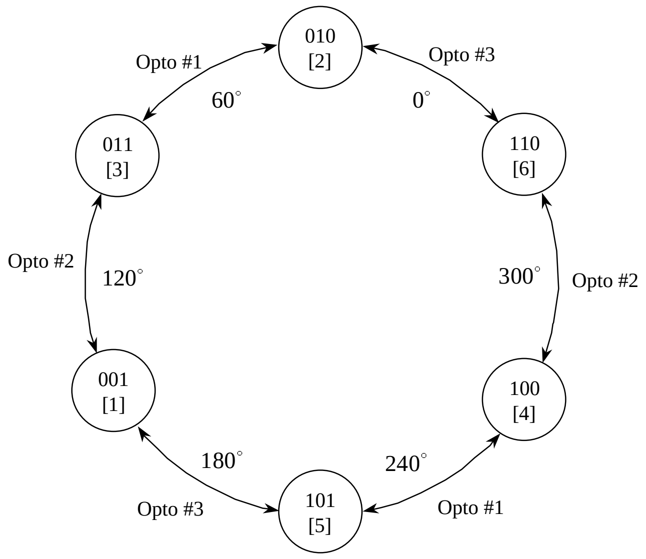
Often, a PI controller is used. In this example adding an integrator to the control law will not make much difference in the loop performance, except only at very low speeds, because the integrator action must be slower than the motor pole to stabilize the loop. In this example using a 3-phase, 12/8 SRM, the motor pole is located near 32 Hz. The SRM operating speed required to produce the equivalent of 32 Hz commands to the SRM current loops is 240 rpm. Thus, in this case, only at operating speeds lower than 240 rpm would any integrator action be helpful.

The current loop gain is set using the ILOOP_GAIN constant in the file CONSTANT.H. For this value, Q3 scaling is used, thus setting ILOOP_GAIN = 22 results in $K = 2.75$, which is sufficiently close to the desired value of 2.8, for this application.

Position Estimation

Recall that Figure 9 showed six possible combinations of the opto-coupler output states per electrical cycle of the SRM. The transitions of the outputs define specific angles. This information can readily be described by a state machine, such as Figure 19.

Figure 19. State Transition Diagram for the SRM Position Pickoff



The state, *[ ],* is defined by 'zyx', where z is the logic state of opto-coupler #3, y of opto-coupler #2, and x is the state of opto-coupler #1.

Position measurements are made by using this state machine and identifying which opto-coupler transition occurs, using the DSP's capture units.

The opto-couplers and slotted disk provide position measurements at six discrete points per electrical cycle of the SRM. Many commutation schemes, however, require continuous position information to optimize performance. Thus, to provide a position estimate between measurements, the equation,

$$\hat{\theta}(k) = \hat{\theta}(k-1) + \hat{\omega}_f(\bar{k}) \times \frac{1}{f_s} \qquad (30)$$

is used, where f_s is the estimation update rate and $\bar{k}$ represents the time of the most recent capture edge. Equation (30) is implemented, using double precision arithmetic, as follows:

```
      long dp;   /* delta-position in mechanical angle */
      int speed;
      int temp;

      if (anSRM->wEst_10xrpm > 0) {
            dp = anSRM->wEst_10xrpm * K_POSITION_EST + anSRM->dp_remainder;
            anSRM->dp_remainder = dp & 0xffff;
            temp = (int) (dp >> 16);
            anSRM->position = anSRM->position + (temp * NR);
      }
      else {
            speed = -anSRM->wEst_10xrpm;
            dp = speed * K_POSITION_EST + anSRM->dp_remainder;
            anSRM->dp_remainder = dp & 0xffff;
            temp = (int) (dp >> 16);
            anSRM->position = anSRM->position - (temp * NR);

}
```

The constant K_POSITION_EST (Q16), compensates for units (shaft velocity is available in the software as `SRM.wEst_10xrpm` with units of rpm × 10) and is calculated according to the equation,

$$\text{K\_POSITION\_EST} = \frac{1}{10}(\text{rpm}\times 10)\times\frac{1\,(\text{sec})}{f_s}\times\frac{1\,(\text{min})}{60\,(\text{sec})}\times\frac{360^\circ}{(\text{rev})}\times\frac{65535}{360^\circ}\times 2^{16} \quad (31)$$

for, f_s = 5 kHz, K_POSITION_EST = 1432.

During startup, the digital I/O ports determine the state of the rotor and initial position is estimated in the mid-range of the state. For example, a reading of [100], (consistent with Figure 19) yields an initial position estimate of 270 electrical degrees. The capture units provide subsequent measurements, by recognizing the edges, or state transitions.

Velocity Estimation

The three opto-coupler outputs produce an edge every 7.5° of mechanical rotation, and each opto-coupler produces an edge every 22.5° mechanical. At each edge, velocity is calculated according to the equation,

$$\hat{\omega} = \frac{\Delta\theta}{\Delta t} = \frac{60\cdot\Delta\theta\cdot f_{clk}}{N} \quad (32)$$

where,

$\hat{\omega}$ is the velocity estimate (rpm)

$\Delta\theta$ is the distance between opto-coupler edges (rev)

Δt is the time between edges (min)

N is the number of clock counts between edges

f_{clk} is the clock frequency (Hz)

The time between edges is determined from the capture units. The capture units are programmed via the CAPCON register to use timer #2 as a time base, and to trigger on both rising and falling edges. Timer #2 is programmed to count at 1.25 MHz via the T2CON. Although we trade-off resolution in measuring Δt, a clock frequency of 1.25 MHz is chosen, versus a maximum of 20 MHz, so that the 16-bit registers containing the count do not overflow except at very low speeds. Using a 1.25 MHz clock, the counter overflows only at shaft speeds less than 71.5 rpm, considered very low for our application. So that we can operate (although degraded) at speeds lower than about 100 rpm, Δt in (32) is determined by a software counter of the number of 5 kHz timer interrupts that occur between opto-coupler edges.

It can be shown that when instantaneous velocity is estimated by (32) that the quantization of a velocity estimate is given by

$$Q = \frac{d\hat{\omega}}{dN} = \frac{\omega^2}{60 \cdot \Delta\theta \cdot f_{clk}} \tag{33}$$

and, Q is the quantization of velocity (rpm). In our design, $\Delta\theta$ = 1/16 revolution (22.5° mechanical) and f_{clk} = 1.25 MHz. Thus at 1200 rpm, the quantization is 0.31 rpm.

Various filtering can be applied to (32) for smoothing the velocity estimate, depending upon the application. What has proven useful is a combination of FIR and IIR filtering of the form:

$$\hat{\omega}_f(\bar{k}) = \alpha \cdot \hat{\omega}_f(\bar{k}-1) + (1-\alpha) \cdot \sum_{j=(\bar{k}-5)}^{\bar{k}} \hat{\omega}(j) \tag{34}$$

The FIR filter portion of (34) uses six (from $\bar{k}-5$ to $\bar{k}$) instantaneous velocity estimates. Because there are six opto-coupler edges per electrical cycle, once per cycle estimation errors are removed.

The FIR filtering and the determination of the instantaneous velocity estimate is calculated using double precision as follows:

```
DWORD a1,a2,a3,a4,a5,a6;
DWORD sum_cnt;
int  inst_velocity;
```

```
/*------------------------------------------*/
/*  Obtain instantaneous velocity estimate   */
/*------------------------------------------*/
if (mode == 1) {/* use timer #2 as time base */

   /*-------------------------------------------------*/
   /* FIR filter for removing once per electrical cycle */
   /* effects                                          */
   /*-------------------------------------------------*/
   a1 = (DWORD) anSRM->capture_delta[0][0];
   a2 = (DWORD) anSRM->capture_delta[0][1];
   a3 = (DWORD) anSRM->capture_delta[1][0];
   a4 = (DWORD) anSRM->capture_delta[1][1];
   a5 = (DWORD) anSRM->capture_delta[2][0];
   a6 = (DWORD) anSRM->capture_delta[2][1];
   sum_cnt = a1+a2+a3+a4+a5+a6;

   /*----------------------------------------------------*/
   /* apply “velocity = delta_theta/delta_time” algorithm  */
   /*----------------------------------------------------*/
   sum_cnt = K1_VELOCITY_EST/sum_cnt;
   inst_velocity = ((int) sum_cnt) * anSRM->shaft_direction;
}

else {      /* else, use timer ISR count as time base */

   /*----------------------------------------------------*/
   /* apply “velocity = delta_theta/delta_time” algorithm  */
   /*----------------------------------------------------*/
   sum_cnt = K2_VELOCITY_EST/anSRM->delta_count;
   inst_velocity = ((int) sum_cnt) * anSRM->shaft_direction;
}
```

Here, K1_VELOCITY_EST and K2_VELOCITY_EST are constants which incorporate $\Delta\theta$ and units so that the instantaneous velocity estimate has units of (rpm $\times$ 10). The constants are calculated using,

$$\text{K1\_VELOCITY\_EST} = 6 \times 22.5\,(\text{deg}) \times \frac{1.25\text{e}6\,(\text{cnts})}{(\text{sec})} \times \frac{1\,(\text{rev})}{360\,(\text{deg})} \times \frac{60\,(\text{sec})}{(\text{min})} \times 10$$

(35)

$$\text{K2\_VELOCITY\_EST} = 1 \times 7.5\,(\text{deg}) \times \frac{5000\,(\text{cnts})}{(\text{sec})} \times \frac{1\,(\text{rev})}{360\,(\text{deg})} \times \frac{60\,(\text{sec})}{(\text{min})} \times 10$$

The IIR filtering is implemented as:

```
long filt_velocity;

/*----------------------------------------------------*/
/*  IIR filter for smoothing velocity estimate       */
/*----------------------------------------------------*/
filt_velocity = (ALPHA * anSRM->wEst_10xrpm)
```

```
+ (ONE_MINUS_ALPHA * inst_velocity);
anSRM->wEst_10xrpm = (int) (filt_velocity >> 3);
```

The filter coefficient, α, is chosen equal to 0.875, [ALPHA = 7 (Q3)]. Let α approach zero for a higher bandwidth velocity estimate (less smoothing, more noise) and let α approach one for more smoothing, less noise, and lower bandwidth.

Commutation

The commutation strategy ultimately determines the performance of the SRM. Torque-speed range, machine efficiency, torque ripple, and acoustic noise all depend, to some extent, on the commutation algorithm. Design of the commutation algorithm must consider requirements in each of these areas, while trading off cost issues such as the algorithm complexity and the availability or accuracy of various sensors. For a current controlled SRM, commutation can be described as the transformation of the desired net motor torque into a set of desired phase currents. This is described mathematically by the equation,

$$i_{cmd}^{j} = g_j(\cdot) \times T_{cmd} \tag{36}$$

and $j = 1,\ldots,m$ and m is the total number of motor phases. In general, $g(\ )$, is a non-linear function of shaft angle θ, shaft speed ω, the desired torque command T_{cmd}, the DC bus voltage V_{bus}, and the motor instantaneous inductance, L. The most simple choice for $g(\ )$ is given by

$$g(\theta) = \begin{cases} 1, & \theta_{ON} \le \theta < (\theta_{ON} + \delta\theta) \\ 0, & \text{otherwise} \end{cases} \tag{37}$$

where the dwell angle, $\delta\theta$, must be at least equal to 360°/m (electrical), to avoid regions of zero torque production. An example of commutation described by (37) is illustrated by Figure 6. The turn-on angle, θ_{ON}, is typically a few degrees beyond the unaligned position of a phase. Equation (37) is useful for only single-quadrant operation. For four quadrant operation, (37) must be modified, for example,

$$g(\theta, T_{cmd}) = \begin{cases} 1, & [\theta_{ON} \le \theta < (\theta_{ON} + \delta\theta)] \ \& \ T_{cmd} > 0 \\ 1, & [(\theta_{ON} + \pi) \le \theta < (\theta_{ON} + \pi + \delta\theta)] \ \& \ T_{cmd} < 0 \\ 0, & \text{otherwise} \end{cases} \tag{38}$$

where the conduction angles are offset by 180° electrical (π radians) when negative torque is desired. This allows a phase to conduct during the region where $\frac{dL}{d\theta} < 0$. An even more flexible approach, which results in a wider operating range for the SRM, allows the turn-on and dwell angles to vary. For example, (38) is extended to allow θ_{ON} and $\delta\theta$, to be functions of velocity, desired torque, and the DC bus voltage.

Often, for minimizing torque ripple, the commutation is designed such that two phases conduct simultaneously and share the job of producing the desired SRM torque. In this case, (38) is further extended to a function of the form,

$$g(\theta, T_{cmd}) = \begin{cases} \rho(\theta), \ [\theta_{ON} \le \theta < (\theta_{ON} + \delta\theta)] \ \& \ T_{cmd} > 0 \\ \rho(\theta), \begin{bmatrix} (\theta_{ON} + \pi) \le \theta \\ \theta < (\theta_{ON} + \pi + \delta\theta) \end{bmatrix} \& \ T_{cmd} < 0 \\ 0, \quad \text{otherwise} \end{cases} \tag{39}$$

where $\rho(\theta)$ is the sharing function. Sharing functions are not implemented in this application report, however, further information on the choice of sharing functions can be found in a publication by Kjaer, *et al*.[9] Essentially commutation schemes of the form in (39) use knowledge of the motor characteristics to design a non-linear function, $\rho(\theta)$, that produces a linear output torque.

In this example, the commutation coefficients, $g(\,)$, were calculated using (38), where $\theta_{ON} = \pi/6 + \theta_{adv}$ (radians), $\delta\theta = \pi/3$ (radians), and the advance angle, θ_{adv}, is given by (21). This yields a single-quadrant, fixed-dwell, variable turn-on commutation algorithm. This algorithm is implemented as follows:

```
int phase;
WORD electricalAngle;
WORD angle;
int channel;
long advance;

/*--------------------------*/
/* Advance angle calculation */
/*--------------------------*/
advance = (anSRM->wEst_10xrpm * anSRM->desiredTorque);
advance = advance >> 9;

/*-------------------------------------------------------*/
/* Offset for advance angle negative torque, if required */
/*-------------------------------------------------------*/
```

[9] P. C. Kjaer, J. Gribble, and T. J. E. Miller, pp. 1585-1593.

```
    if (anSRM->desiredTorque > 0) {
            electricalAngle = anSRM->position + (int) advance;
    }
    else {
            electricalAngle = anSRM->position + PI_16 - (int) advance;
    }

    for (phase=0; phase< NUMBER_OF_PHASES; phase++) {

            /*------------------------------*/
            /* 120 degree offsets for phase */
            /*------------------------------*/
            angle = electricalAngle - phase * TWOPIBYTHREE_16;
            /*------------------------------------------------------------*/
            /* turn phase on, if between desired angles and switch       */
            /* the mux on the A/D to measure the desired     */
            /* phase current                                 */
            /*------------------------------------------------------------*/
            if ( (angle >= (PIBYSIX_16)) && (angle < (FIVEPIBYSIX_16)) ) {
            anSRM->active[phase] = 1;
            channel = anSRM->a2d_chan[phase];
            switch_mux(channel,channel+8);
            }
            else {
            anSRM->active[phase] = 0;
            }
            }
    }
```

As seen in the code above, the advance angle calculation (which yields an advance angle in units of bits, 65535 bits = 360 electrical degrees) is computed according to the equation,

$$\theta_{adv}(bits) = \frac{i_{cmd}(bits) \times \hat{\omega}(rpm \times 10)}{K}, \quad K = 2^9 = 512 \tag{40}$$

From (21) and (40) we can show that the calculation for K which includes L_{min}, V_{dc}, and accounts for units is given by

$$\frac{1}{K} = L_u(\mathrm{H}) \times \frac{1}{V_{bus}(\mathrm{V})} \times \frac{(\mathrm{rpm} \times 10)}{10} \times \frac{1\,(\mathrm{A})}{239.4\,(\mathrm{bits})} \times \frac{1\,(\mathrm{min})}{60\,(\mathrm{sec})} \times \frac{360^\circ(\mathrm{m})}{(\mathrm{rev})} \times \frac{8^\circ(\mathrm{e})}{1^\circ(\mathrm{m})} \times \frac{65535\,(\mathrm{bits})}{360^\circ(\mathrm{e})} \tag{41}$$

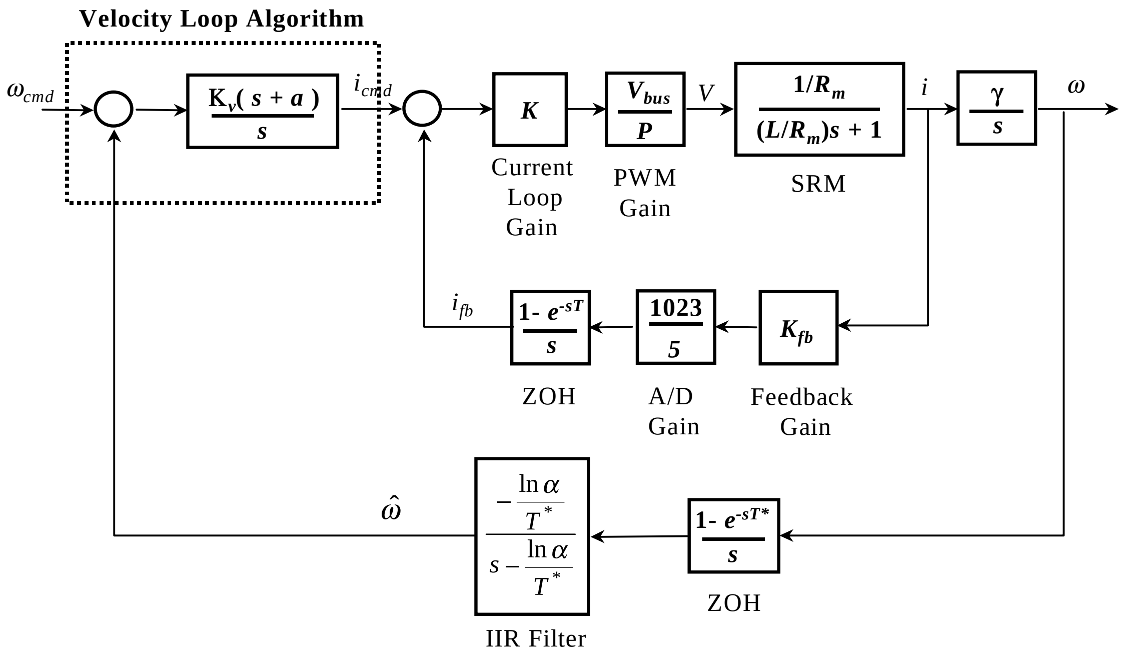
With L_u and V_{bus} given by Table 1, K = 776.25. However, the value of 512 was used because dividing by multiples of 2 is readily accomplished using simple shift instructions. For our application, this simplification provided satisfactory results.

Velocity Controller

Speed is regulated in a closed-loop manner by comparing the desired shaft velocity to the estimated shaft velocity and then compensating the error. We use a PI (proportional plus integral) control action for the velocity loop compensation, so that the steady-state velocity error is zero. The PI coefficients are determined using linear analysis.

Figure 20 shows a simplified model of the velocity loop, where the coefficient, γ, having units of (rad/s)/A, is a non-linear quantity, including the shaft/load inertia and the instantaneous torque constant of the SRM.

Figure 20. Simplified Block Diagram of SRM Velocity Loop Using PI Control



In the figure, K_v controls the loop gain and ‘a’ is the radian frequency of the PI zero.

The non-linearity in γ is due to the non-linear torque/current relationship of the SRM where for non-saturating conditions is given by (18) and, in general, by (15). Thus, the open-loop gain of the velocity loop will vary, approximately, as current squared. This variation can be significant over the operating range of the SRM. From 4 A to 1 A, for example, the gain variation is 16, or 24 dB. Depending on the application, this gain variation may need to be compensated, with a square root law, for example, to stabilize the loop. In this example, the loop compensation was designed with sufficient margin, at the expense of dynamic response, so that this variation can be ignored. The IIR filter used for smoothing the velocity estimate, has a z-transform given by,

$$H(z) = \frac{(1-\alpha)z}{z-\alpha} \tag{42}$$

and is modeled in the Laplace domain as shown in Figure 20.

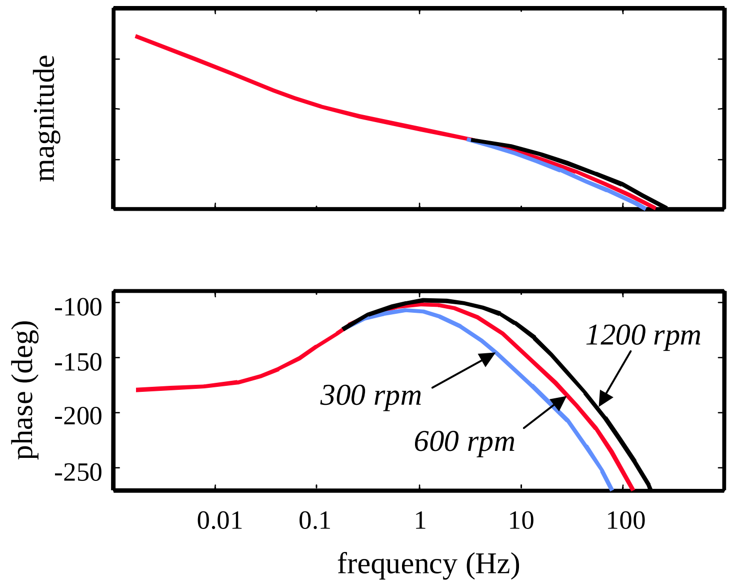
Another interesting feature of this speed loop is that although the velocity loop update rate is a fixed-frequency of 1 kHz, the feedback (provided by the opto-coupler edges) occurs at a variable rate which is a function of rotor speed and given by,

$$f_{samp}\ (\text{Hz}) = 0.8 \times (\text{speed in rpm}) = \frac{1}{T^*} \tag{43}$$

This behavior generates a variable time delay in the velocity loop, due to the zero-order hold, and also makes the dynamics of the IIR filter time-varying.

Using the information in Table 1 and Figure 20 it is possible to obtain the open-loop frequency response, $\frac{\hat{\omega}}{\omega_{cmd}}$ (see Figure 21) for the velocity loop, independent of γ.

Figure 21. Open-Loop Frequency Response of the SRM Velocity Loop at Several Motor Speeds, for a = 0.73 rad/s



The phase loss due to the variable time delay as a function of rotor speed is apparent.

The absolute magnitude as a function of frequency is unknown; however, the shape of the magnitude and the phase are correct. From Figure 21, clearly the desired open-loop crossover frequency is in the 1-4 Hz range for this particular loop. By moving the PI zero, a, beyond 0.73 rad/s, the bandwidth can be extended, while trading off stability margins. If the load inertia and motor torque constant information are known (i.e. γ known), then K_v can be determined analytically; otherwise, the velocity loop gain is set experimentally.

The PI algorithm is implemented as:

```
/*----------------------------*/
/* calculate error signal     */
/*----------------------------*/
speed_error = anSRM->wDes_10xrpm - anSRM->wEst_10xrpm;

/*----------------------------*/
/* integrate error*/
/*----------------------------*/
anSRM->integral_speed_error = anSRM->integral_speed_error +
(long)speed_error;

/*----------------------------*/
/* apply integrator limit     */
/*----------------------------*/
if (anSRM->integral_speed_error > INTEGRAL_LIMIT) {
      anSRM->integral_speed_error = INTEGRAL_LIMIT;
}
if (anSRM->integral_speed_error < -INTEGRAL_LIMIT) {
      anSRM->integral_speed_error = -INTEGRAL_LIMIT;
}

/*----------------------------*/
/* PI filter*/
/*----------------------------*/
integral_error = (int) ((KI*anSRM->integral_speed_error) >> 13);
anSRM->desiredTorque = ((KP*speed_error) >> 1) + integral_error;
```

This implements a PI compensator, with integrator limits, of the form,

$$K_P + \frac{K_I}{s} = \frac{K_v(s+z)}{s} \quad \Rightarrow \begin{cases} K_p = K_v \\ K_I = K_v \times z \end{cases} \qquad (44)$$

Through experimentation, it was determined that K_v = 0.5, provided satisfactory performance. In the software implementation of the integrator, the multiplication by Δt is not performed. Thus, this factor is carried implicitly in K_I. For z = 0.73 rad/s and Δt = 1/1000 sec, K_I = 0.365. This is approximately implemented by setting KI = 3 (scaled Q13 $\times$ 1000) in the file CONSTANT.H.

The integrator limit value is calculated such that the condition is, $\frac{\text{KI} \times \text{INTEGRAL\_LIMIT}}{2^{12}} \leq 1000.$

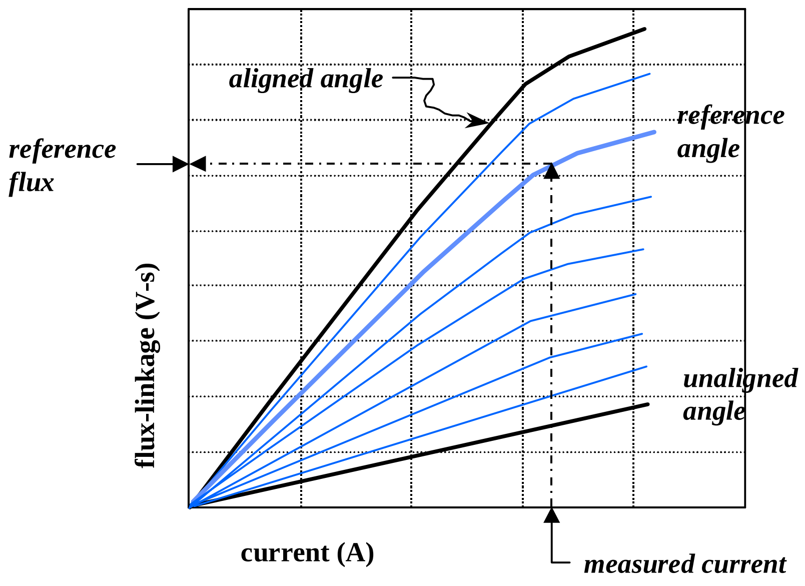
Position Sensorless Control

In this section, the operation of an SRM without using a shaft sensor to report the rotor position is discussed. As in the prior example, SRM commutation typically involves feeding back a rotor position signal to a controller. Position sensors often used include encoders, resolvers, hall sensors, or opto-couplers. For a variety of reasons, including lower cost & better reliability, it is desirable to eliminate the shaft position sensor.

Various schemes have been developed to address the sensor elimination problem.[10] Because of the salient nature of the SR motor's rotor and stator, inductance varies in the motor as a function of rotor position. This variation is sufficiently sensitive such that the motor's inductance profile can be used to estimate rotor position. All position sensor elimination methods for SR motors exploit this relationship in one way or another.

One novel approach described by Lyons, *et al*,[11] uses an estimate of the flux linked by a phase and a model of the SR motor's magnetic characteristics, such as Figure 22, to determine if the rotor angle is approaching, or has gone past, a known reference angle.

Figure 22. SRM Magnetization Curves



[10] W. F. Ray and I. H. Al-Bahadly, "Sensorless Methods for Determining the Rotor Position of Switched Reluctance Motors," *Proc. EPE Conf.*, Vol. 6, 1993, pp. 7-13.
[11] J. Lyons, S. MacMinn, and M. Preston, "Flux/Current Methods for SRM Rotor Position Estimation," *IEEE IAS Annual Meeting Conf. Record,* 1991, pp. 482-487.

The model maps current to flux-linkage at various rotor positions. Thus, given a phase current measurement, the flux-linkage at the reference angle can be determined, as illustrated in Figure 22.

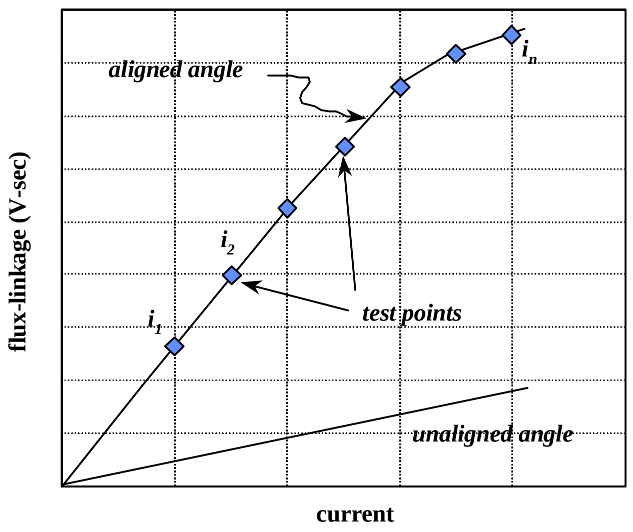
In their sensorless approach, the basic idea is to compare the flux estimate to the reference flux calculated from the model in order to gauge the rotor's position with respect to the reference angle.

In this example, we describe a position sensorless commutation method which extends this idea, and removes the requirement for having an a priori model of the motor characteristics. The fundamental idea is to learn, through an on-line identification algorithm, the flux-current characteristics of each phase of the motor at the aligned position, and then use this information to commutate the motor by approximating the motor's magnetic characteristics at an appropriate reference angle.

Overview of Method

Exciting an arbitrary motor phase, with current large enough to generate torque in excess of the starting friction, will align the rotor with the energized phase. Once at the aligned position, a calibration sequence is cycled through where, as illustrated by Figure 23, desired levels of current are commanded to each motor phase at a series of discrete test points.

Figure 23. Calibration Sequence for SRM at Aligned Position



For the j^{th} test point ($j = 1,\ldots,n$), phase flux is estimated for N cycles via a discretized version of the integral form of Faraday's law,

$$\hat{\phi}_j(N) = \left(\sum_{k=1}^{N} \left[\hat{v}(k) - \hat{i}_j(k)\hat{R}(k) \right] \cdot T \right) + \hat{\phi}(0) \tag{45}$$

where,

n	= total number of test points
$\hat{\phi}(k)$	= estimated phase flux at time, k
$\hat{v}(k)$	= estimated voltage across the phase winding
$\hat{i}(k)$	= measured phase current
$\hat{R}(k)$	= estimated phase resistance
T	= sampling rate
N	= number of intervals in the estimation period
$\hat{\phi}(0)$	= initial flux-linked by the phase, known equal to 0.

At the end of each estimation period ($k = N$), flux/current data pairs $[\hat{\phi}_j(N), \hat{i}_j(N)]$ are recorded and the next current test point is commanded. The procedure is repeated for each of the *n* test points, and then for each of the remaining phases.

Upon completion of the calibration sequence and subsequent curve fitting of the flux/current data, let $g(i)$ be the resulting function describing the estimated magnetic characteristics of the motor at the aligned angle,

$$\hat{\phi}(\theta_{aligned}, i) = g(i) \tag{46}$$

Since flux-linked increases monotonically with angle, moving from the unaligned angle to the aligned angle, the magnetic characteristics of the motor at some other angle, θ_{ref}, can be defined by the expression,

$$\hat{\phi}(\theta_{ref}, i) = \alpha(i) g(i) \tag{47}$$

where,

$$0 \le \alpha(i) \le 1 \ \forall \ i.$$

As in the publication by Lyons, *et al*,[12] a comparison of the estimated flux-linked to the predicted flux-linked at a reference position is the basis for commutating the motor. Thus, during normal operation of the SRM, if flux is estimated using (45) (with N undefined for normal operation) and if the commutation function, α( ), is intelligently designed, then satisfying the condition,

$$\hat{\phi}(k) \geq \alpha[i_{meas}(k)] \times g[i_{meas}(k)] \tag{48}$$

will define appropriate switching points (i.e. commutations) for the SRM.

[12] J. Lyons, S. MacMinn, and M. Preston, pp. 482-487.

Example – SRM Drive without Position Sensor

This section describes an example application of a position sensorless SRM drive.

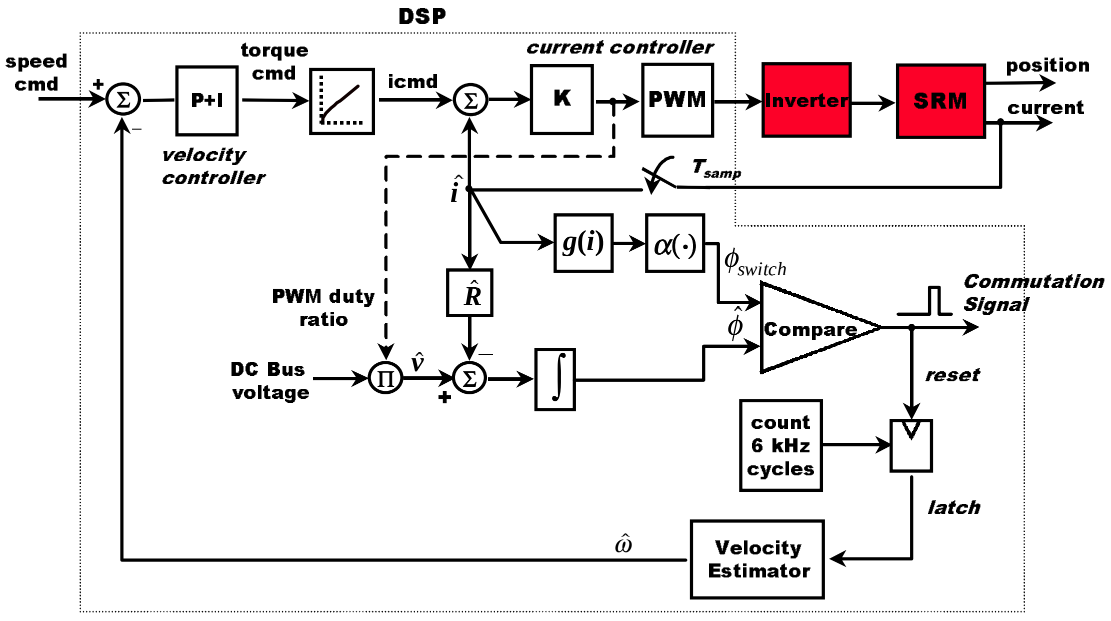
Hardware Description

See the Hardware Description in the Example - SRM Drive With Position Feedback section above. The hardware used in this example is identical to that used in the previous example, with the exception that the connections between the DSP and the opto-couplers do not exist here.

Software Description

The software described in this section is written in C and is designed for operating a 3-phase 12/8 SRM in closed loop current and closed loop speed control, without using a shaft position sensor. A block diagram of the algorithms implemented is given in Figure 24.

Figure 24. Block Diagram of the SRM Sensorless Controller

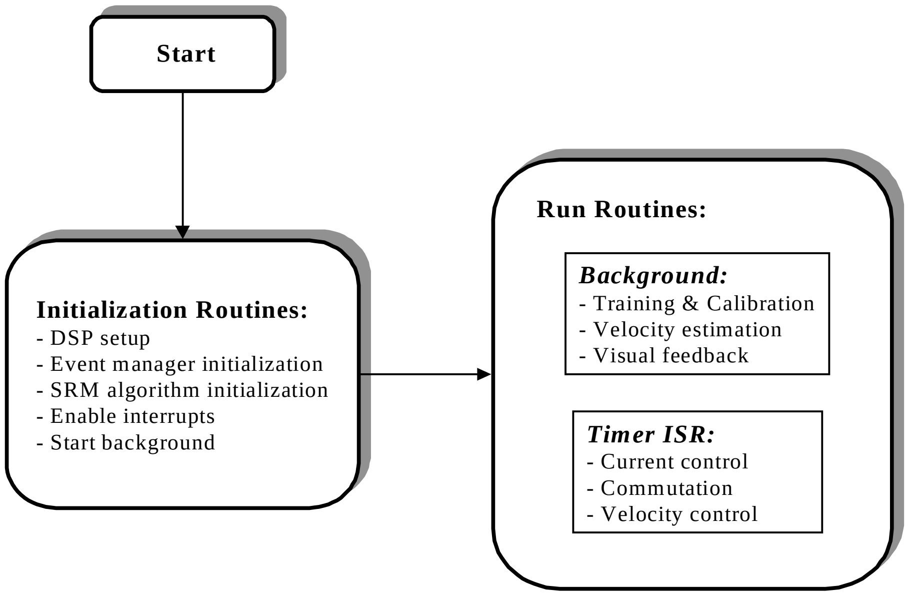


Current in each phase is individually regulated using a fixed-frequency PWM scheme. Velocity is estimated by monitoring the elapsed time between phase commutations, which are a known distance apart. Velocity error is determined by comparing the speed command to the velocity estimate. The velocity controller, consisting of a PI compensator, calculates the motor torque required to bring the velocity error to zero. The commutation algorithm converts the torque command to a phase current command. Also, position sensorless commutation is performed, as described above, by comparing the estimated phase flux to the flux predicted for the reference angle. Further details on each of the algorithms are provided in subsequent sections of this report.

Program Structure

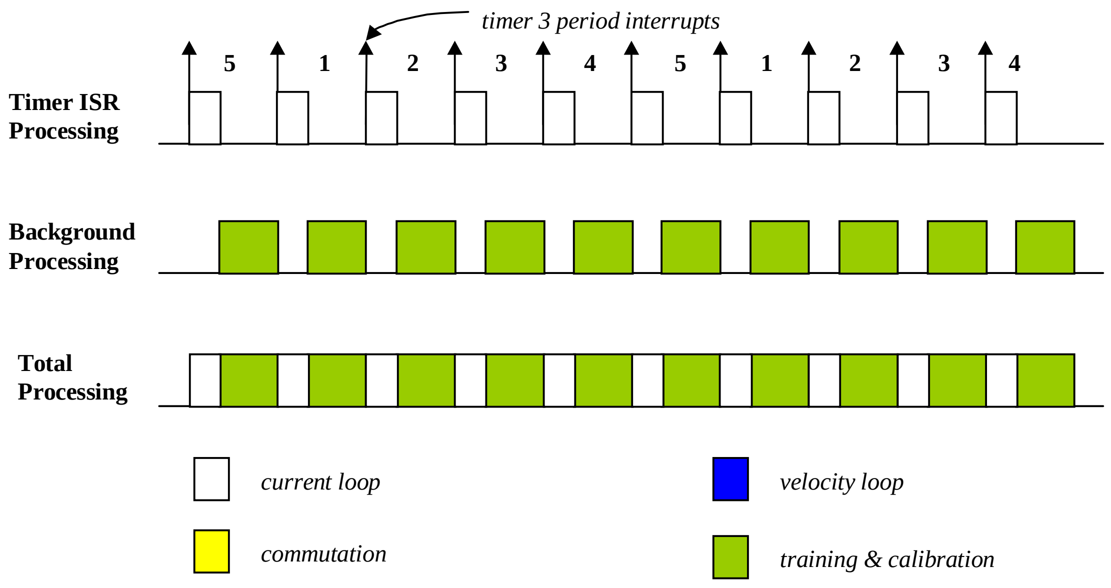
Figure 25 shows the structure of the sensorless SRM control software for the TMS320F240 DSP.

Figure 25. TMS320F240 Position Sensorless SRM Control Program Structure



At the highest level, the software consists of initialization routines and run routines. Initialization is started following each processor reset. Upon completion of the initialization, the background task is started. As in the prior example, the background task is an infinite loop where, when required, lower priority processing tasks (velocity estimation and a visual feedback routine) are executed. In addition to these two operations, in the sensorless drive, a training and calibration routine is performed in the background. The training and calibration routine is immediately executed upon entry into the background task, initiated by the setting of a flag during initialization. Like initialization, this operation is executed only one time per software reset, however, because current control is required during the training, it is performed with the run routines. While the training and calibration algorithm is executing, only the current control algorithm is being executed in the timer ISR. As illustrated in Figure 26, the processor is 100% loaded during this time.

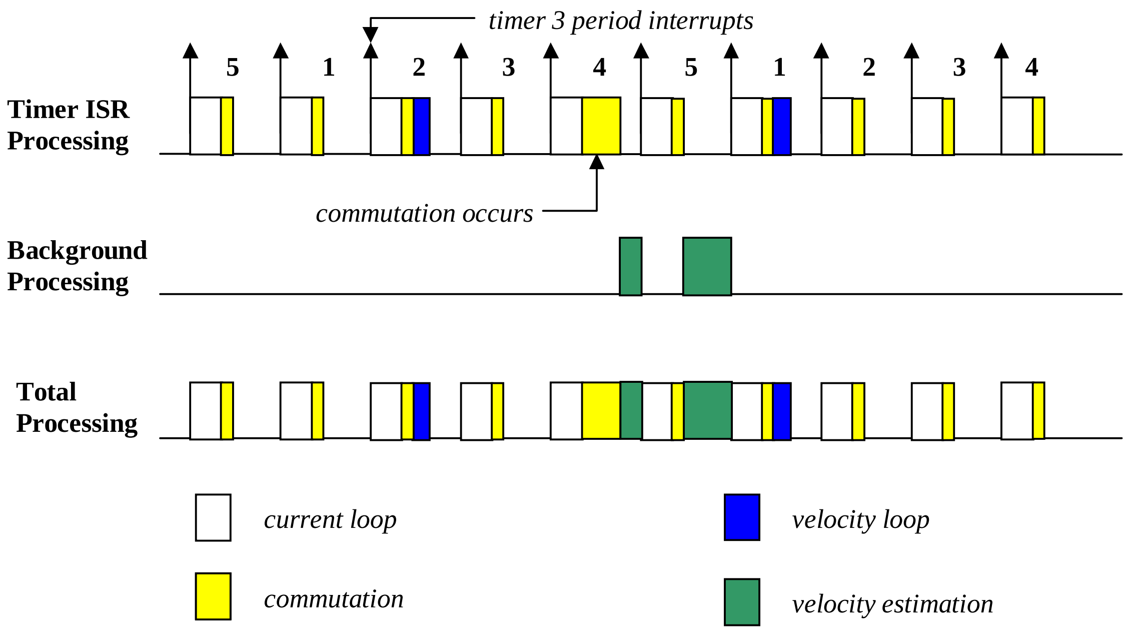
Figure 26. Processor Timeline Showing Loading With Training and Calibration Routine Active



Upon completion of the training/calibration routine, normal motor operation proceeds.

During normal operation of the SRM, the processor timeline is typical of that shown in Figure 27.

Figure 27. Processor Timeline During Normal SRM Operation



The current loop and commutation algorithm are both performed at the frequency, F, while the velocity control algorithm is executed at a frequency $F/5$. As mentioned above, the velocity estimation algorithm executes in background, and a new estimate is calculated following each commutation instance. This frequency is given by,

$$\text{commutation frequency (Hz)} = \text{shaft speed (rpm)} \times \frac{1\,(\text{min})}{60\,(\text{sec})} \times \frac{mN_R}{(\text{rev})} \tag{49}$$

All time critical motor control processing is done via the timer interrupt service routine. The timer ISR is executed at each occurrence of the maskable CPU interrupt INT3. This interrupt corresponds to the event manager group B interrupts, of which only the timer #3 period interrupt, TPINT3 is enabled. The frequency, F, at which this routine is executed is specified by the value loaded into the timer 3 period register.

Table 3. Benchmark Data for the Various Position Sensorless SRM Drive Software Modules

S/W Block	Module	#Cycles	Execution Time @ 50 ns	Average Execution Frequency	Relative Time @ 6 kHz
Velocity Estimation	background	2512	125.6 µs	400 Hz [1]	8.4 µs
Visual Feedback	background	60	3.0 µs	2 Hz	0.0 µs
Current Control[2]	timer ISR	1160	58.0 µs	6000 Hz	58.0 µs
Commutation[3]	timer ISR	248	12.4 µs	5600 Hz [1]	11.6 µs
Commutation[4]	timer ISR	1272	63.6 µs	400 Hz [1]	4.2 µs
Velocity Control	timer ISR	494	24.6 µs	1200 Hz	4.9 µs
Misc. Overhead	timer ISR	140	7.0 µs	6000 Hz	7.0 µs
C Context Switch	RTS.LIB	120	6.0 µs	6000 Hz	6.0 µs
Total					100.1 µs

[1]at a shaft speed of 1000 rpm
[2]including flux estimation
[3]when commutation condition is not met
[4]when commutation condition is met

The data in Table 3 shows that when the timer ISR frequency, F, is chosen as 6 kHz, that the overall processor loading is equal to

$$\text{processor loading} = \frac{100.1\ \mu s}{166.7\ \mu s} = 60.5\%, \qquad (50)$$

when using a 20 MHz dsp clock frequency. The code size is 3599 words, and 326 words are required for variable/data storage. Thus, the total memory requirement is less than 4 K words. Complete code listings are given in Appendix B.

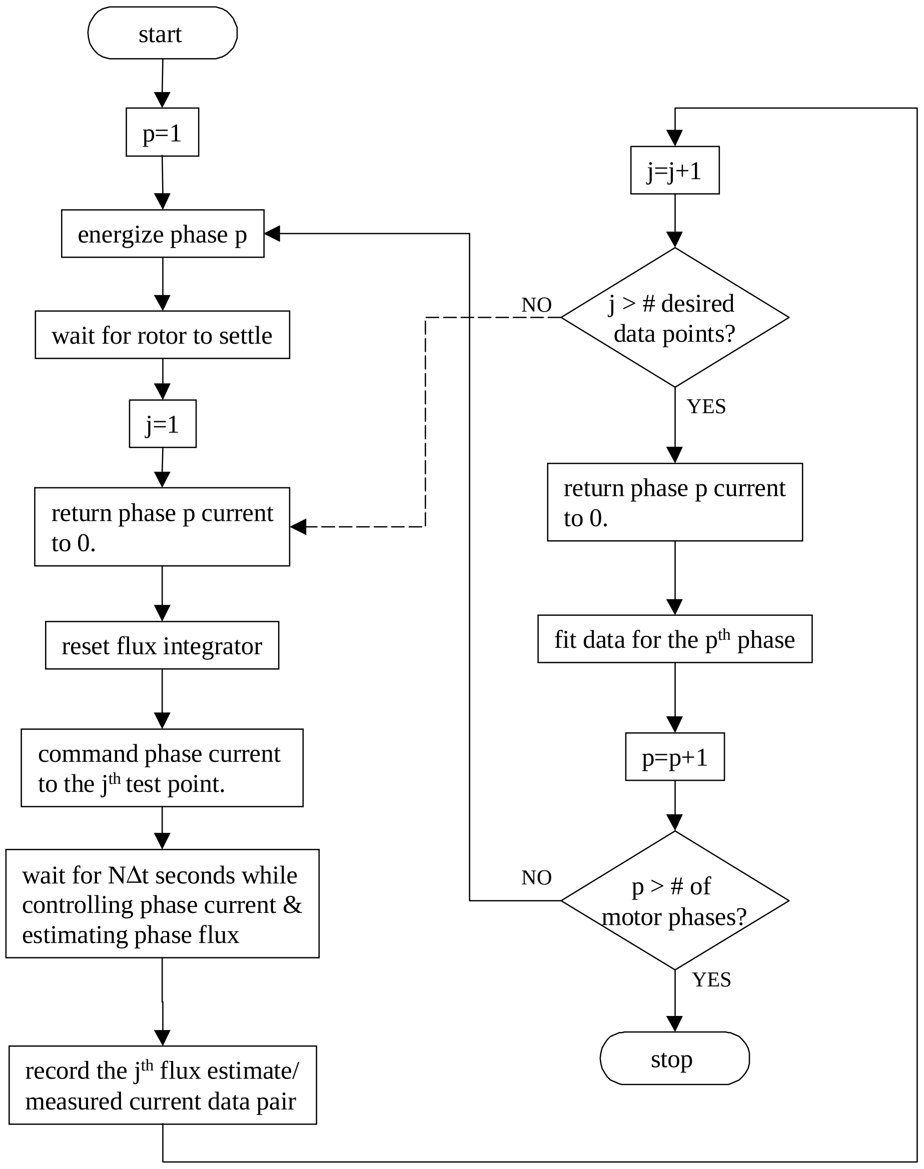
Initialization Routines

The initialization for the sensorless SRM drive is fundamentally identical to that described in the Initialization Routines section and by Figure 16. In each example, the DSP setup is equivalent. Although timer #2 is unused in this example, the same event manager initialization code is used for the sensorless example as was used in the prior example.

Training and Calibration

The sequence used to estimate the SRM's flux-linkage versus current characteristics is described in the flowchart diagram of Figure 28.

Figure 28. SRM Training Algorithm Flowchart


start
p=1
energize phase p
wait for rotor to settle
j=1
return phase p current to 0.
reset flux integrator
command phase current to the jth test point.
wait for NΔt seconds while controlling phase current & estimating phase flux
record the jth flux estimate/ measured current data pair
j=j+1
j > # desired data points?
NO
YES
return phase p current to 0.
fit data for the pth phase
p=p+1
p > # of motor phases?
NO
YES
stop

There are several issues of interest to point out in the algorithm.

First, the SRM is an undamped system, or more precisely, very lightly damped with friction. Thus after energizing a motor phase, the shaft will oscillate about the aligned position. This ringing is a complex function of parameters including the shaft inertia, load inertia, friction, and the angle traveled from rest to the aligned position. To improve estimation accuracy, we wait for a certain amount of time for the rotor to reach the aligned position and settle. For our particular example, we used a value of 6 seconds, which was determined through experimentation and observation. The wait time is specified using the macro WAIT(300), where the argument of the macro specifies the number of 20 msec periods for waiting.

Second, how long should the time, $N\Delta t$, be? $N\Delta t$ is the amount of time that the current is allowed to flow in the motor phase prior to returning the phase current to 0 and making the next measurement. The time should be long enough for the current to reach a steady state value. This in turn is related to the bandwidth of the current control loop. Using an approximate linear analysis similar to that in the Current Controller section, it can be shown that for the SRM described in Table 1, the current will reach approximately 98% of its final value in 12 msec. In this example, a value of 60 msec, specified by WAIT(3), was used. This should be more than adequate.

Third, how many data points should be used? More data points will yield a better estimate at the expense of requiring more memory and taking more measurement/processing time. The number of data points is specified with the NUM_POINTS constant. Here the value NUM_POINTS = 80 was used.

Finally, and most importantly, how should the estimation of the magnetization curve at the aligned position be accomplished? In other words, how should the data be fit to a model? This is the classic estimation problem which, in itself, can be a topic of much discussion. The interested reader is encouraged to review the publications by Anderson and Moore[13] and Lewis.[14]

When considering this question, it is important to realize that when implementing the flux estimation algorithm (45) errors exist. Let the estimates/measurements of phase voltage, current and resistance by given by,

[13] B. Anderson and J. Moore, Optimal Filtering, Prentice-Hall Publishing, Englewood Cliffs, NJ, 1979.
[14] F. Lewis, Applied Optimal Control & Estimation, Prentic Hall Publishing, Englewood Cliffs, NJ, 1992.

$$\begin{cases} \hat{v}(k) = v(k) + \delta v(k) \\ \hat{i}(k) = i(k) + \delta i(k) \\ \hat{R}(k) = R(k) + \delta R(k) \end{cases} \tag{51}$$

where the right-hand side terms of (51) are the true values and associated perturbations of voltage, current, and phase resistance.

By substituting (51) into (45) we obtain,

$$\hat{\phi}_j(N) = \left(\sum_{k=1}^{N} \left[(v(k) + \delta v(k)) - (i_j(k) + \delta i(k))(R + \delta R(k)) \right] \cdot T \right) + \hat{\phi}(0) \tag{52}$$

The development of an estimation algorithm proceeds by specifying an error model for the perturbations and by specifying a process model for the flux-linkage as a function of current. The choice of both models is largely application dependent.

As with most trade-offs, it is a question of accuracy vs. simplicity. For example, let the process model be the non-saturating SRM,

$$\phi(k) = L_a i(k) \tag{53}$$

Further assume that the errors in voltage, current, and resistance are each constant,

$$\begin{cases} \delta v(k) = \delta v \\ \delta i(k) = \delta i \\ \delta R(k) = \delta R \end{cases} \tag{54}$$

Substituting (54) into (52) gives,

$$\hat{\phi}_j(N) = \left(\sum_{k=1}^{N} \left[v(k) - i_j(k) R \right] \cdot T \right) + \left(\sum_{k=1}^{N} \left[(\delta v - \delta i R - \delta i \delta R) - i_j(k) \delta R \right] \cdot T \right) \tag{55}$$

where the initial phase flux is known to equal 0. If the first summation on the right-hand side of (55) represents the true value of flux-linked given by our process model (53) and we assume in the second summation that $i_j(k) = i_j(N)$, for all k, then (55) can be rewritten in the more convenient vector form,

$$\hat{\phi}_j(k) = \mathbf{H}_j^{\mathbf{T}} \mathbf{x} \tag{56}$$

where, $\mathbf{H}_j = \begin{bmatrix} i_j(N) \\ i_j(N)NT \\ NT \end{bmatrix}$, and $\mathbf{x} = \begin{bmatrix} L_a \\ -\delta R \\ (\delta v - \delta i R - \delta i \delta R) \end{bmatrix}$, and $j = 1,\ldots,n$.

Solutions to (56) are well known. Given n linearly independent measurements ($n \geq 3$), a technique such as least squares or Kalman filtering can be used to determine the coefficients of $\mathbf{x}$.

Although this estimation approach extends to more sophisticated models, for simplicity, in this example, we assume that the process model is given by (53), that the voltage and current perturbations are constants , and that the perturbation of phase resistance is zero. Thus, $\mathbf{H}_j = \begin{bmatrix} i_j(N) \\ NT \end{bmatrix}$, and $\mathbf{x} = \begin{bmatrix} L_a \\ (\delta v - \delta i R) \end{bmatrix}$.

The coefficients of $\mathbf{x}$ were found using a least squares method. This solution can be given by,

$$L_a = \frac{n\sum_{j=1}^{n}\hat{\phi}_j i_j - \left(\sum_{j=1}^{n} i_j\right)\left(\sum_{j=1}^{n}\hat{\phi}_j\right)}{n\sum_{j=1}^{n} i_j{}^2 - \left(\sum_{j=1}^{n} i_j\right)^2}, \tag{57}$$

$$(\delta v \text{-} \delta i R) = \frac{1}{n}\sum_{j=1}^{n}\hat{\phi}_j - \frac{L_a}{n}\sum_{j=1}^{n} i_j$$

Because these calculations are performed only during the initialization phase, processing time is not of overwhelming concern. Thus, we used floating-point arithmetic and the library routines found in RTS.LIB in the implementation of (57).

Flux Estimation

As described above, flux is estimated using a discretized version of Faraday’s law, such as (45). Since during the calibration routine we estimated an equivalent bias term, $b = \frac{\delta v - \delta i R}{N}$, this information is used to improve the flux estimate. The flux estimation equation is of the form,

$$\hat{\phi}(k) = \sum_{l=1}^{k}\left[\left(v(l) - i(l)R\right) - b\right]\cdot T \tag{58}$$

Rewriting (58) in terms of the summation of three delta-flux values, multiplying through by 100, and dividing through by the sampling period, T, yields

$$\frac{100\times\hat{\phi}(k)}{T}=\sum_{l=1}^{k}100\times(\Delta\phi_1+\Delta\phi_2)-b \qquad (59)$$

where $\Delta\phi_1 = v(l)\ \text{and}\ \Delta\phi_2 = i(l)R$.

Because the SRM is controlled using a PWM scheme, the instantaneous phase voltage, v, is determined by multiplying the DC bus voltage by the ratio of the commanded pulse width to the PWM period. Thus,

$$\Delta\phi_1 = \text{VBUS}\times(\%\ \text{duty ratio})\,, \qquad (60)$$

where VBUS is assumed to be a constant in this example. In the software, the duty ratio command is contained in the variable SRM.dutyRatio[phase] and is scaled such that 100% = 999. For $\Delta\phi_1$ to have units of $\frac{(\text{V - sec})}{T\ (\text{sec})}$, (60) must be implemented as

$$\Delta\phi_1 = \frac{\texttt{VBUS}\times\texttt{SRM.dutyRatio[phase]}}{999} \qquad (61)$$

and because division by 999 is not readily accomplished, the constant VBUS is rescaled such that,

$$\Delta\phi_1 = \frac{\texttt{VBUS'}\times\texttt{SRM.dutyRatio[phase]}}{1024} \qquad (62)$$

where VBUS’ = 1.025*VBUS. For V_{bus} = 170 V as described in Table 1, the VBUS constant defined in SL_CONST.H must be 174.

In calculating $\Delta\phi_2$, the constant R_PHASE is the phase resistance in ohms, and the SRM phase current is contained in the variable SRM.iFB[phase] and scaled such that 1 Ampere = 239.6 bits. For $\Delta\phi_2$ to have units of $\frac{(\text{V - sec})}{T\ (\text{sec})}$, the calculation of $\Delta\phi_2$ must be implemented as

$$\Delta\phi_2 = \frac{\texttt{R\_PHASE}\times\texttt{SRM.iFB[phase]}}{239.6} \qquad (63)$$

which can be implemented with Q16 scaling, as

$$\Delta\phi_2 = \frac{\texttt{R\_PHASE} \times \texttt{SRM.iFB[phase]} \times 273}{65536} \tag{64}$$

Multiplying the delta-flux terms by 100 provides an improved scaling for the parameter estimation algorithm, yielding integer estimates L_a and b, with greater resolution and accuracy.

The flux estimate integration is implemented using double precision arithmetic as follows:

```
int phase;
long df1, df2;         /* delta-flux 1 and 2*/
int temp1, temp2;
long dflux;

phase = anSRM->Active;

/*-------------------------------------*/
/* update flux linkage estimate         */
/*-------------------------------------*/
df1 = VBUS * anSRM->dutyRatio[phase] + anSRM->df1_remainder;
anSRM->df1_remainder = df1 & 0x3ff;
temp1 = (int) (df1 >> 10);
df2 = R_PHASE * anSRM->iFB[phase] * 273 + anSRM->df2_remainder;
anSRM->df2_remainder = df2 & 0xffff;
temp2 = (int) (df2 >> 16);
dflux = 100*(temp1-temp2) - anSRM->bias[phase];

anSRM->fluxEstimate[phase] = anSRM->fluxEstimate[phase] + dflux;

if (anSRM->fluxEstimate[phase] < anSRM->minFlux ) {
      anSRM->fluxEstimate[phase] = anSRM->minFlux;
}
```

During calibration, the bias term is zero and the minimum flux limit is a large negative number. Upon completion of the calibration routine and the parameter estimation, the bias terms are defined and the minimum flux limit is set to zero.

Current Controller

The current controller used for the position sensorless SRM drive is very similar to that used for the SRM drive with the position sensor and is described in Section 4.2.3. The software differs in this case only in that it allows only a single phase to be active at a time, and the flux estimation algorithm is executed during the current loop algorithm. The linear analysis is exactly the same as the SRM drive using a position sensor. Following that procedure the phase loss due to the ZOH sampling at 370 Hz can be calculated as,

$$\theta_{loss} = 2\pi \times 370 \times (83.3 \times 10^{-6}) = 0.194\ \text{rad} = 11.1^\circ \tag{65}$$

Assuming that the processing delay is equal to 50% of a loop cycle, or another 83.3 µsec, then the net effect of digital implementation yields about 22° of phase loss at 370 Hz. When combined with the 90° due to the motor pole, the phase loss through the loop is approximately 112°, at 370 Hz. If the loop gain, K, is chosen such that the 0 dB point of the open-loop magnitude occurs at 370 Hz, then the resulting phase margin in the loop will be about 68°. This amount of phase margin provides a very stable loop design. The DC gain of the loop is given by,

$$\frac{K \cdot V_{bus} \cdot K_{fb} \cdot 1023}{P \cdot 5 \cdot R} = 5.092K \tag{66}$$

which when written in decibels is equal to,

$$\text{DC gain} = 14.1\, dB + 20\log_{10}(K) \tag{67}$$

For frequencies where $\omega > (R/L)$, the magnitude of the loop response is equal to,

$$|G(\omega)| = 14.1\, dB + 20\log_{10}(K) - 2 \times \left(\frac{\omega}{R/L}\right) \tag{68}$$

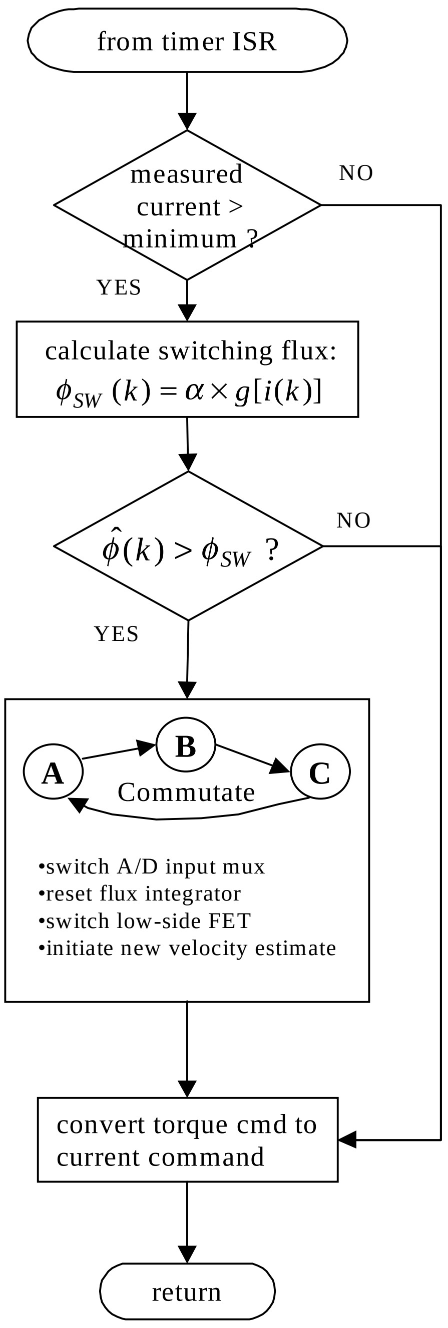
Setting the left-hand side of (68) to 0 dB, while $\omega = 2\pi(370)$ rad/s, and solving for K, yields the value of K which insures the desired open-loop crossover point for the current loop. In this case K = 2.8.

The current loop gain is set using the ILOOP_GAIN constant in the file SL_CONST.H. For this value, the Q3 scaling is used, thus setting ILOOP_GAIN = 22 results in K = 2.75, which is sufficiently close to the desired value of 2.8.

Commutation

The commutation algorithm is outlined in the flowchart diagram of Figure 29.

Figure 29. SRM Position Sensorless Commutation Algorithm Flowchart


from timer ISR
measured current > minimum ?
NO
YES
calculate switching flux:
$\phi_{SW}(k) = \alpha \times g[i(k)]$
$\hat{\phi}(k) > \phi_{SW}$?
NO
YES
A
B
C
Commutate
•switch A/D input mux
•reset flux integrator
•switch low-side FET
•initiate new velocity estimate
convert torque cmd to current command
return

This algorithm is executed at each occurrence of the timer ISR.

The motor is commutated when the estimated phase flux equals of exceeds the switching flux,

$$\hat{\phi}(k) > \alpha[i(k)] \times g[i(k)] \tag{69}$$

Clearly the selection of α has significant impact on the performance of the SRM. Choosing α is entirely analogous to choosing conduction angles when designing a commutation strategy using position feedback. As α approaches unity, the switching point for a phase moves in such a way as to retard the turn-on of the next phase and the turn-off of the active phase. Likewise, as α approaches zero, the turn-on of the next phase advances. Because α provides only a single degree of freedom, the dwell angle effectively stays constant, unlike conventional controllers where both turn-on and dwell angles are adjusted. This is a limitation of the approach. Conceivably a more complicated approach allowing multiple phases to conduct simultaneously could be developed based on the techniques described in this report.

For best performance, the choice of α should consider both the desired phase current and the shaft speed of the motor, much in the same way that these quantities are used to implement phase advance and variable dwell angle control in conventional SRM commutation strategies. Thus, more generally, the commutation equation is written as,

$$\hat{\phi}(k) > \alpha(i_{cmd},\omega)\times g(i) \tag{70}$$

Commutation is restricted to occur only when the measured phase current exceeds a minimum threshold. As can be seen in Figure 22, there is considerable loss of sensitivity in estimating position from flux when the phase current is small. In this circumstance, small errors in current translate to large errors in position, and consequently errors in the desired switching instant. This is a second limitation of this type of position sensorless commutation scheme. To help guard against this problem, the threshold is used. The limitation makes this commutation scheme most suitable for applications where the motor naturally operates under loaded conditions, like fans.

In this example, a constant value of $\alpha = 0.625$ was chosen. Using the Q3 scaling, ALPHA = 5 as defined in the file SL_CONST.H

Velocity Estimation

Velocity is estimated by counting the number of times that the timer ISR is executed, between commutation decisions. At each commutation, velocity is calculated according to the equation,

$$\hat{\omega} = \frac{\Delta\theta}{\Delta t} = \frac{60\cdot\Delta\theta\cdot f_{clk}}{N} \tag{71}$$

where,

$\hat{\omega}$ = the velocity estimate (rpm)

$\Delta\theta$ = the distance between commutations (rev)

Δt = the time between edges (min)

N = the number of timer ISR executions between commutations

f_{clk} = the timer ISR execution frequency (Hz)

It can be shown that when instantaneous velocity is estimated by (71) that the quantization of a velocity estimate is given by

$$Q = \frac{d\hat{\omega}}{dN} = \frac{\omega^2}{60 \cdot \Delta\theta \cdot f_{clk}} \qquad (72)$$

and, Q is the quantization of velocity (rpm). In this example, $\Delta\theta = \frac{1}{mN_R}$ is 1/24 revolution and f_{clk} is equal to 6 kHz. For example, at 1000 rpm the quantization is 66.7 rpm – which at 6.7% of the shaft speed is a fairly considerable amount. Although filtering of the instantaneous estimates can significantly smooth the velocity estimation, the filter bandwidth also serves to limit the achievable velocity loop bandwidth.

Applications that require fast dynamic response along with good speed accuracy are not well-suited for this type of approach. Again, applications like fans and blowers are best suited for this type of scheme.

The instantaneous velocity estimate is calculated using double precision as follows:

```
DWORD sum_cnt;
int  inst_velocity;

if (anSRM->delta_count > 7) {    /* protect from divide by 0 and  */
                                 /* estimate out of range         */

/*------------------------------------------------*/
/* apply velocity = delta_theta/delta_time algorithm */
/*------------------------------------------------*/
sum_cnt = K_VELOCITY_EST/anSRM->delta_count;
inst_velocity = ((int) sum_cnt);
}
```

Here, K_VELOCITY_EST is a constant that incorporates $\Delta\theta$ and units so that the instantaneous velocity estimate has units of (rpm $\times$ 10). This constant is calculated by,

$$\text{K\_VELOCITY\_EST} = 15\,(\text{deg}) \times \frac{6000\,(\text{cnts})}{(\text{sec})} \times \frac{1\,(\text{rev})}{360\,(\text{deg})} \times \frac{60\,(\text{sec})}{(\text{min})} \times 10 \qquad (73)$$

To smooth the velocity estimate for quantization a low pass, IIR filter is used. The IIR filtering is implemented,

```
/*--------------------------------------------------*/
/* IIR filter for smoothing velocity estimate*/
/*--------------------------------------------------*/
filt_velocity = (BETA * anSRM->wEst_10xrpm)
             + (ONE_MINUS_BETA * inst_velocity);
anSRM->wEst_10xrpm = (int) (filt_velocity >> 4);
```

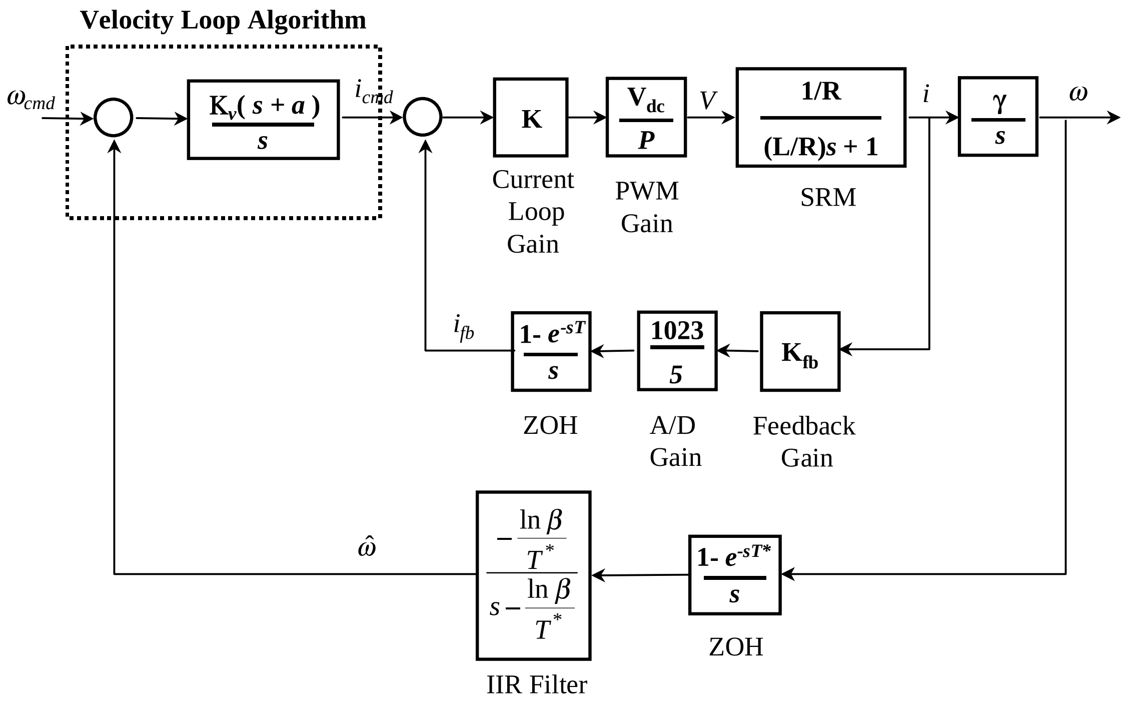
The filter coefficient, β, is chosen equal to 0.9375, [BETA = 15 (Q4)]. Let β approach zero for a higher bandwidth velocity estimate (less smoothing, more noise) and let β approach one for more smoothing, less noise, and lower bandwidth. Here, because of the relatively large amount of quantization in the velocity estimate, β was chosen near 1.0.

Velocity Controller

The velocity controller used for the position sensorless SRM drive is very similar to that used for the SRM drive with the position sensor and described in Velocity Controller under the Example - SRM Drive With Position Feedback section above. Speed is regulated in a closed-loop manner, by comparing the desired shaft velocity to the estimated shaft velocity and then compensating the error. A PI (proportional plus integral) control action is used for the velocity loop compensation, to drive steady-state velocity error is zero. The PI coefficients are determined using linear analysis.

Figure 30 shows a simplified model of the velocity loop, where the coefficient, γ, having units of (rad/s)/A, is a non-linear quantity, including the shaft/load inertia and the instantaneous torque constant of the SRM.

Figure 30. Simplified Block Diagram of the SRM Velocity Loop Using PI Control



In the figure, K_v controls the loop gain and 'a' is the radian frequency of the PI zero.

The non-linearity in γ is due to the non-linear torque/current relationship of the SRM. Thus, the open-loop gain of the velocity loop will vary, approximately, as current squared. The IIR filter, which has a z-transform given by,

$$H(z)=\frac{(1-\beta)z}{z-\beta} \tag{74}$$

is modeled in the Laplace domain as shown in Figure 30.

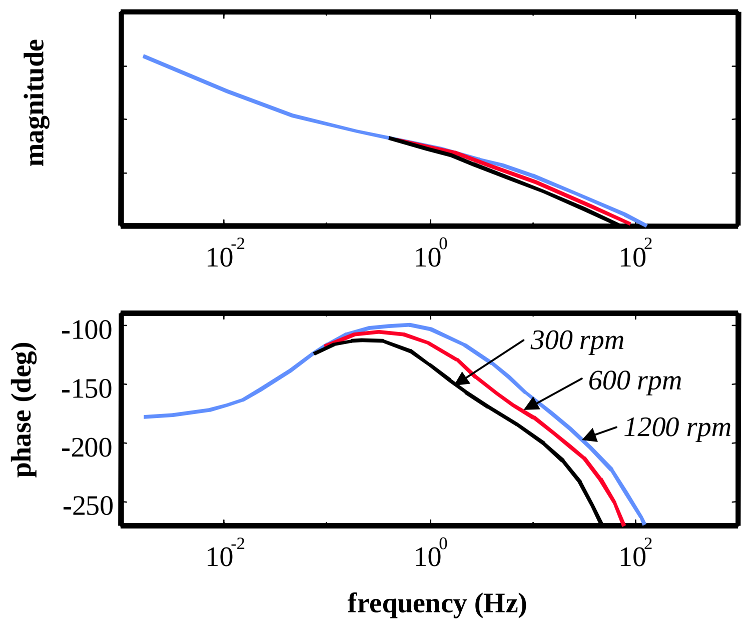
An interesting feature of the speed loop is that although the velocity loop update rate is a fixed-frequency of 1.2 kHz, the feedback, which is provided at each phase commutation, occurs at a variable rate. This feedback frequency, is a function of rotor speed and can be given by,

$$f_{samp}\ (\text{Hz}) = 0.4\times(\text{speed in rpm}) = \frac{1}{T*} \tag{75}$$

This results in an effective variable time delay in the velocity loop, due to the zero-order hold, and also makes the dynamics of the IIR filter, used for smoothing the velocity estimate, variable.

Using the information in Table 1 and Figure 30 it is possible to obtain open-loop frequency response plots (see Figure 31) for the velocity loop, independent of γ.

Figure 31. Open-Loop Frequency Response of SRM Velocity Loop at Several Motor Speeds, for α = 0.293 rad/s



The variable loop dynamics as a function of rotor speed are apparent. The absolute magnitude, as a function of frequency, is unknown, however the shape of the magnitude and the phase are correct. From Figure 30, clearly the desired open-loop crossover frequency is in the 0.2-1.0 Hz range. By moving a about 0.293 rad/s, the bandwidth vs. stability margin tradeoffs can be made. If the load inertia and motor torque constant information are known (i.e. γ known), then K_v can be determined analytically; otherwise, the velocity loop gain is set experimentally.

The velocity control loop compensation algorithm is implemented as:

```
/*-------------------------*/
/* calculate error signal  */
/*-------------------------*/
speed_error = anSRM->wDes_10xrpm - anSRM->wEst_10xrpm;

/*---------------------------------------------*/
/* reduce loop bandwidth at low shaft speed     */
/*---------------------------------------------*/
if (anSRM->wEst_10xrpm < SPEED_THRESHOLD) {
       speed_error = speed_error >> 2;
}

/*-------------------------*/
/* integrate error         */
/*-------------------------*/
anSRM->integral_speed_error = anSRM->integral_speed_error
+ (long)speed_error;

/*-------------------------*/
/* apply integrator limit  */
/*-------------------------*/
if (anSRM->integral_speed_error > INTEGRAL_LIMIT) {
       anSRM->integral_speed_error = INTEGRAL_LIMIT;
}
if (anSRM->integral_speed_error < -INTEGRAL_LIMIT) {
       anSRM->integral_speed_error = -INTEGRAL_LIMIT;
}

/*-------------------------*/
/* PI filter               */
/*-------------------------*/
integral_error = (int) ((KI*anSRM->integral_speed_error) >> 14);
anSRM->desiredTorque = ((KP*speed_error) >> 2) + integral_error;

/*-------------------------------*/
/* insure a minimum phase current  */
/*-------------------------------*/
if (anSRM->desiredTorque < MIN_TORQUE_COMMAND) {
       anSRM->desiredTorque = MIN_TORQUE_COMMAND;

}
```

This implements a PI compensator, with integrator limits, of the form,

$$K_P + \frac{K_I}{s} = \frac{K_v(s+z)}{s} \quad \Rightarrow \begin{cases} K_p = K_v \\ K_I = K_v \times z \end{cases} \tag{76}$$

Through experimentation, it was determined that $K_v = 0.25$, provided satisfactory performance. In the software implementation of the integrator, the multiplication by Δt is not performed. Thus, this factor is carried implicitly in K_I. For $z = 0.293$ rad/s and $\Delta t =$ 1/1200 sec, $K_I = 0.073$ (≈ KI = 1 scaled Q14×1200). The integrator limit value, INTEGRAL_LIMIT, is calculated such that, $\frac{\text{KI} \times \text{INTEGRAL\_LIMIT}}{2^{14}} \le 1000$ is satisfied. To add stability to the loop during the motor startup, the loop gain is reduced by a factor of 4, while the motor accelerates through about 400 rpm. Also, to help avoid the problems described earlier with this sensorless commutation approach at low levels of phase current, the torque command, which is the velocity loop output, is limited to some minimum threshold.

References

Anderson, B. and J. Moore, Optimal Filtering, Prentice-Hall Publishing, Englewood Cliffs, NJ, 1979.

Becerra, R., M. Ehsani, and T. J. E. Miller, “Commutation of SR Motors,” *IEEE Trans. Power Electronics*, Vol. 8, pp. 257-262, July 1993.

Clemente, S. and A. Dubhashi, “HV Floating MOS-Gate Driver IC,” *International Rectifier application note AN-978A*, International Rectifier, El Segundo, CA, 1990.

Husain, I. and M. Ehsani, " Torque Ripple Minimization in Switched Reluctance Motor Drives by PWM Current Control," *Proc. APEC’94*, 1994.

Ilic-Spong, M., T. J. E. Miller, S. R. MacMinn, and J. S. Thorp, “Instantaneous Torque Control of Electric Motor Drives,” *IEEE Trans. Power Electronics*, Vol. 2, Jan. 1987.

Kjaer, P. C., J. Gribble, and T. J. E. Miller, “High-Grade Control of Switched Reluctance Machines,” *IEEE Trans. Industry Electronics*, vol. 33, Nov. 1997.

Lewis, F. Applied Optimal Control & Estimation, Prentic Hall Publishing, Englewood Cliffs, NJ, 1992.

Lyons, J., S. MacMinn, and M. Preston, “Flux/Current Methods for SRM Rotor Position Estimation,” *IEEE IAS Annual Meeting Conf. Record,* 1991.

Miller , T. J. E. (ed.), “Switched Reluctance Motor Drives,” Intertec Communications Inc., Ventura, CA, 1988.

Miller, T. J. E., “Switched Reluctance Motors and Their Control,” Magna Physics Publishing, Hillsboro, OH, and Oxford, 1993.

Ray, W. F. and I. H. Al-Bahadly, “Sensorless Methods for Determining the Rotor Position of Switched Reluctance Motors,” *Proc. EPE Conf.*, Vol. 6, 1993.

Vukosavic, S. and V. Stfanovic, “SRM Inverter Topologies: A Comparative Evaluation,” *IEEE IAS Annual Meeting Conf. Record*, 1990.

Appendix A. Software Listings for a TMS320F240-Based SRM Drive With Position Sensor

This appendix contains the software to implement an SRM drive using a slotted disk type position sensor for a TMS320F240 DSP.

File	Major Modules	Description
TYPEDEFS.H		header file – data type definitions
C240.H		header file – C240 register definitions
CONSTANT.H		header file – SRM constant defintions
SRM.H		header file – SRM variable declarations
MAIN.C	main( )	supervisory program
	c_int3( )	timer ISR
	c_int4( )	capture ISR
SRM.C	Time_Update_Position( )	time update algorithm for position estimation
	Msmt_Update_Position( )	measurement update algorithm for position estimation
	Msmt_Update_Velocity( )	velocity estimation algorithm
	Commutation_Algorithm( )	SRM fixed-dwell, variable turn-on commutation algorithm
	velocityController( )	shaft velocity loop compensation algorithm
	currentController( )	phase current loop compensation algorithm
EVMGR.C		modules for event manager initialization and operating the event manager peripherials
VECTORS.ASM		interrupt vectors
LINK.CMD		linker command file

```
/***********************************************************
*   File: TYPEDEFS.H                                       *
*   TMS320x240 Test Bed Code                               *
*   Texas Instruments, Inc.                                *
*   Copyright (c) 1996 Texas Instruments Inc.              *
*   11/05/96   Version 1.0                                 *
*   Jeff Crankshaw                                         *
***********************************************************/
#ifndef TYPEDEFS_H
#define TYPEDEFS_H

#define FALSE 0
#define TRUE  1

typedef unsigned int   WORD;          /* 16-bit data */
typedef unsigned long  DWORD;         /* 32-bit data */
typedef volatile WORD * PORT;

#define STR(x) #x

#define OUTMAC(address,data)  \
 asm("        LDPK    _"STR(data));  \
 asm("        OUT     _"STR(data) "," STR(address))

#define INMAC(address,data)   \
 asm("        LDPK    _"STR(data));  \
 asm("        IN      _"STR(data) "," STR(address))

#define Int_Read(addr)         * (int *) (addr)
#define Int_Write(addr,data)   * (int *) (addr) = (data)

#endif  /* _TYPEDEFS */


/***********************************************************
*   File: C240.H                                           *
*   TMS320x240 Test Bed Code                               *
*   Texas Instruments, Inc.                                *
*   Copyright (c) 1996 Texas Instruments Inc.              *
*   11/05/96   Version 1.0                                 *
*   Jeff Crankshaw                                         *
*                                                          *
*   TMS320C240 Peripheral Register Addresses               *
*                                                          *
***********************************************************/
#ifndef c240_h
#define c240_h

#include "typedefs.h"

/*--------------------------------------------------------*/
/* definitions of I/O space macros                        */
/*--------------------------------------------------------*/
#define STR(x) #x

#define OUTMAC(address,data)  \
 asm("        LDPK    _"STR(data));  \
 asm("        OUT     _"STR(data) "," STR(address))

#define INMAC(address,data)   \
 asm("        LDPK    _"STR(data));  \
 asm("        IN      _"STR(data) "," STR(address))

#define LED_LOC 000ch  /* F240 EVM I/O space location for LEDs */
```

```
/*---------------------------------------------------------*/
/* definitions of CPU core registers                       */
/*---------------------------------------------------------*/
#define IMR_REG        (( PORT )0x0004 )
#define IFR_REG        (( PORT )0x0006 )

/*---------------------------------------------------------*/
/* External Memory Interface Registers                     */
/*---------------------------------------------------------*/
#define WSGR        0x0ffff
/* Wait State Generator Register                           */

/*---------------------------------------------------------*/
/* System Module Registers                                 */
/*---------------------------------------------------------*/
#define SYSCR         (( PORT )0x07018)      /* System Module Control Register */
#define SYSSR         (( PORT )0x0701A)      /* System Module Status Register */
#define SYSIVR        (( PORT )0x0701E)      /* System Interrupt Vector Register */
#define XINT1_CR      (( PORT )0x07070)      /* Int1 (type A) Control reg */
#define NMI_CR        (( PORT )0x07072)      /* Non maskable Int (type A) Control reg */
#define XINT2_CR      (( PORT )0x07078)      /* Int2 (type C) Control reg */
#define XINT3_CR      (( PORT )0x0707A)      /* Int3 (type C) Control reg */
#define PDPINT_CR     (( PORT )0x0742C)      /* Power Drive Protection Int cntl reg */

/* System Interrupt Vector Register - Address offsets */
#define PHANTOM_INT_VECTOR    0x00
#define NMI_INT_VECTOR        0x02
#define XINT1_INT_VECTOR      0x01
#define XINT2_INT_VECTOR      0x11
#define XINT3_INT_VECTOR      0x1f
#define SPI_INT_VECTOR        0x05
#define SCI_RX_INT_VECTOR     0x06
#define SCI_TX_INT_VECTOR     0x07
#define RTI_INT_VECTOR        0x10
#define PDP_INT_VECTOR        0x20
#define EV_CMP1_INT_VECTOR    0x21
#define EV_CMP2_INT_VECTOR    0x22
#define EV_CMP3_INT_VECTOR    0x23
#define EV_SCMP1_INT_VECTOR   0x24
#define EV_SCMP2_INT_VECTOR   0x25
#define EV_SCMP3_INT_VECTOR   0x26
#define EV_T1PER_INT_VECTOR   0x27
#define EV_T1CMP_INT_VECTOR   0x28
#define EV_T1UF_INT_VECTOR    0x29
#define EV_T1OF_INT_VECTOR    0x2a
#define EV_T2PER_INT_VECTOR   0x2b
#define EV_T2CMP_INT_VECTOR   0x2c
#define EV_T2UF_INT_VECTOR    0x2d
#define EV_T2OF_INT_VECTOR    0x2e
#define EV_T3PER_INT_VECTOR   0x2f
#define EV_T3CMP_INT_VECTOR   0x30
#define EV_T3UF_INT_VECTOR    0x31
#define EV_T3OF_INT_VECTOR    0x32
#define EV_CAP1_INT_VECTOR    0x33
#define EV_CAP2_INT_VECTOR    0x34
#define EV_CAP3_INT_VECTOR    0x35
#define EV_CAP4_INT_VECTOR    0x36
#define AC2_INT_VECTOR        0x04
```

```
/*-----------------------------------------------------------*/
/* Digital I/O  Registers                                    */
/*-----------------------------------------------------------*/
#define OCRA           (( PORT )0x07090)     /* Output Control Reg A */
#define OCRB           (( PORT )0x07092)     /* Output Control Reg B */
#define PADATDIR       (( PORT )0x07098)     /* I/O port A Data & Direction reg. */
#define PBDATDIR       (( PORT )0x0709A)     /* I/O port B Data & Direction reg. */
#define PCDATDIR       (( PORT )0x0709C)     /* I/O port C Data & Direction reg. */


/*--------------------------------------------------------------------------*/
/* Watch-Dog(WD) / Real Time Int(RTI) / Phase Lock Loop(PLL) Registers     */
/*--------------------------------------------------------------------------*/
#define RTICNTR        (( PORT )0x07021)     /* RTI Counter reg */
#define WDTCNTR        (( PORT )0x07023)     /* WD Counter reg  */
#define WDTKEY         (( PORT )0x07025)     /* WD Key reg */
#define RTICR          (( PORT )0x07027)     /* RTI Control reg */
#define WDCR           (( PORT )0x07029)     /* WD Control reg */
#define CKCR0          (( PORT )0x0702B)     /* PLL control reg 1 */
#define CKCR1          (( PORT )0x0702D)     /* PLL control reg 2 */

/*-----------------------------------------------------------*/
/* Analog-to-Digital Converter(ADC) registers                */
/*-----------------------------------------------------------*/
#define ADCTRL1        (( PORT )0x07032)     /* ADC Control & Status reg */
#define ADCTRL2        (( PORT )0x07034)     /* ADC Configuration reg */
#define ADCFIFO1       (( PORT )0x07036)     /* ADC Channel 1 Result Data */
#define ADCFIFO2       (( PORT )0x07038)     /* ADC Channel 2 Result Data */

/*-----------------------------------------------------------*/
/* Serial Peripheral Interface (SPI) Registers               */
/*-----------------------------------------------------------*/
#define SPICCR         (( PORT )0x07040)      /* SPI Config Control Reg */
#define SPICTL         (( PORT )0x07041)      /* SPI Operation Control Reg */
#define SPISTS         (( PORT )0x07042)      /* SPI Status Reg */
#define SPIBRR         (( PORT )0x07044)      /* SPI Baud rate control reg */
#define SPIEMU         (( PORT )0x07046)      /* SPI Emulation buffer reg */
#define SPIBUF         (( PORT )0x07047)      /* SPI Serial Input buffer reg */
#define SPIDAT         (( PORT )0x07049)      /* SPI Serial Data reg */
#define SPIPC1         (( PORT )0x0704D)      /* SPI Port control reg1 */
#define SPIPC2         (( PORT )0x0704E)      /* SPI Port control reg2 */
#define SPIPRI         (( PORT )0x0704F)      /* SPI Priority control reg */

/*---------------------------------------------------------*/
/* Serial Communications Interface (SCI) Registers         */
/*---------------------------------------------------------*/
#define SCICCR         (( PORT )0x07050)      /* SCI Comms Control Reg */
#define SCICTL1        (( PORT )0x07051)      /* SCI Control Reg 1 */
#define SCIHBAUD       (( PORT )0x07052)      /* SCI Baud rate control */
#define SCILBAUD       (( PORT )0x07053)      /* SCI Baud rate control */
#define SCICTL2        (( PORT )0x07054)      /* SCI Control Reg 2 */
#define SCIRXST        (( PORT )0x07055)      /* SCI Receive status reg */
#define SCIRXEMU       (( PORT )0x07056)      /* SCI EMU data buffer */
#define SCIRXBUF       (( PORT )0x07057)      /* SCI Receive data buffer */
#define SCITXBUF       (( PORT )0x07059)      /* SCI Transmit data buffer */
#define SCIPC1         (( PORT )0x0705D)      /* SCI Port control reg1 */
#define SCIPC2         (( PORT )0x0705E)      /* SCI Port control reg2 */
#define SCIPRI         (( PORT )0x0705F)      /* SCI Priority control reg */
```

```
/*-------------------------------------------------------*/
/* Event Manager (EV) Registers                          */
/*-------------------------------------------------------*/
#define GPTCON        (( PORT )0x07400)     /* General Timer Controls */
#define T1CNT         (( PORT )0x07401)     /* T1 Counter Register */
#define T1CMP         (( PORT )0x07402)     /* T1 Compare Register */
#define T1PER         (( PORT )0x07403)     /* T1 Period Register */
#define T1CON         (( PORT )0x07404)     /* T1 Control Register */
#define T2CNT         (( PORT )0x07405)     /* T2 Counter Register */
#define T2CMP         (( PORT )0x07406)     /* T2 Compare Register */
#define T2PER         (( PORT )0x07407)     /* T2 Period Register */
#define T2CON         (( PORT )0x07408)     /* T2 Control Register */
#define T3CNT         (( PORT )0x07409)     /* T3 Counter Register */
#define T3CMP         (( PORT )0x0740a)     /* T3 Compare Register */
#define T3PER         (( PORT )0x0740b)     /* T3 Period Register */
#define T3CON         (( PORT )0x0740c)     /* T3 Control Register */
#define COMCON        (( PORT )0x07411)     /* Compare Unit Control */
#define ACTR          (( PORT )0x07413)     /* Full Compare Unit Output Action Ctrl */
#define SACTR         (( PORT )0x07414)     /* Simple Comp Unit Output Action Ctrl */
#define DBTCON        (( PORT )0x07415)     /* Dead Band Timer Control */
#define CMPR1         (( PORT )0x07417)     /* Full Compare Channel 1 Threshold */
#define CMPR2         (( PORT )0x07418)     /* Full Compare Channel 2 Threshold */
#define CMPR3         (( PORT )0x07419)     /* Full Compare Channel 3 Threshold */
#define SCMPR1        (( PORT )0x0741a)     /* Simple Comp Channel 1 Threshold */
#define SCMPR2        (( PORT )0x0741b)     /* Simple Comp Channel 2 Threshold */
#define SCMPR3        (( PORT )0x0741c)     /* Simple Comp Channel 3 Threshold */
#define CAPCON        (( PORT )0x07420)     /* Capture Unit Control */
#define CAPFIFO       (( PORT )0x07422)     /* FIFO1-4 Status Register */
#define FIFO1         (( PORT )0x07423)     /* Capture Channel 1 FIFO Top */
#define FIFO2         (( PORT )0x07424)     /* Capture Channel 2 FIFO Top */
#define FIFO3         (( PORT )0x07425)     /* Capture Channel 3 FIFO Top */
#define FIFO4         (( PORT )0x07426)     /* Capture Channel 4 FIFO Top */
#define IMRA          (( PORT )0x0742c)     /* Group A Interrupt Mask Register */
#define IMRB          (( PORT )0x0742d)     /* Group B Interrupt Mask Register */
#define IMRC          (( PORT )0x0742e)     /* Group C Interrupt Mask Register */
#define IFRA          (( PORT )0x0742f)     /* Group A Interrupt Flag Register */
#define IFRB          (( PORT )0x07430)     /* Group B Interrupt Flag Register */
#define IFRC          (( PORT )0x07431)     /* Group C Interrupt Flag Register */
#define IVRA          (( PORT )0x07432)     /* Group A Int. Vector Offset Register */
#define IVRB          (( PORT )0x07433)     /* Group B Int. Vector Offset Register */
#define IVRC          (( PORT )0x07434)     /* Group C Int. Vector Offset Register */

#endif


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*                                                                */
/*File:                 CONSTANT.H                                */
/*Target Processor:     TMS320F240                                */
/*Compiler Version:                                               */
/*Assembler Version:                                              */
/*Created:              10/1/97                                   */
/*                                                                */
/*----------------------------------------------------------------*/
/*  Constants for the SRM control algorithms                      */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
```

```
/*-----------------------------------------------*/
/* clock frequencies and time related constants */
/*-----------------------------------------------*/
#define PWM_FREQ              20000          /* PWM frequency (Hz)         */
#define SYSCLK_FREQ           20000000       /* DSP clock frequency (Hz)   */
#define CPU_INT_FREQ          5000           /* timer ISR frequency (Hz)   */
#define ONE_HALF_SECOND       (CPU_INT_FREQ/2)

/*---------------------------------- */
/* current loop algorithm constants   */
/*---------------------------------- */
#define ILOOP_GAIN            22             /* current loop gain:                 */
                                             /*      (Q3: gain = 2.75)             */
#define ILIMIT                1023           /* current limit: (1023 bits =        */
                                             /*   5 V x 0.855 A/V = 4.273 A)       */
#define MAXIMUM_DUTYRATIO     999            /* limit on the PWM duty cycle:       */
                                             /*   100 % =                          */
                                             /*   (SYSCLK_FREQ/PWM_FREQ - 1)       */

/*-------------------------------------------*/
/* velocity loop algorithm constants         */
/*-------------------------------------------*/
#define INTEGRAL_LIMIT        2793472        /* integrator limit                   */
#define KI                    3              /* (Q13*1000): Ki = 0.366             */
#define KP                    1              /* Q1: Kp = 0.5                       */


/*----------------------------------------------------------*/
/* position & velocity estimation algorithm constants       */
/*----------------------------------------------------------*/
#define K_POSITION_EST        1432
#define K1_VELOCITY_EST       281250000
#define K2_VELOCITY_EST       62500
#define ALPHA                 7              /* Q3: alpha = 0.875                  */
#define ONE_MINUS_ALPHA       1              /* Q3: 1-alpha = 0.125                */

/*-------------------------------------------*/
/* motor geometry related                    */
/*-------------------------------------------*/
#define NR                    8              /* number of rotor poles              */
#define NUMBER_OF_PHASES      3

/*-----------------------------------------------*/
/* Electrical Angles: 2*pi (rad) = 65535         */
/*-----------------------------------------------*/
#define PIBYSIX_16            5461
#define PIBYFOUR_16           8192
#define PIBYTHREE_16          10923
#define TWOPIBYTHREE_16       21845
#define THREEPIBYFOUR_16      24576
#define FIVEPIBYSIX_16        27307
#define PI_16                 32768
#define FOURPIBYTHREE_16      43690
#define FIVEPIBYTHREE_16      54613
#define TWOPI_16              65535

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*                                                              */
/*File:               SRM.H                                     */
/*Target Processor:   TMS320F240                                */
/*Compiler Version:                                             */
/*Assembler Version:                                            */
/*Created:            10/1/97                                   */
/*                                                              */
/*--------------------------------------------------------------*/
/*  Variable declarations for the SRM control algorithm         */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

#include "constant.h"
#include "typedefs.h"

/*----------------------------------------------------------*/
/* position estimation state machine data structure         */
/*----------------------------------------------------------*/
typedef struct {
  int   state;
  WORD  position;
  int   direction;
} state_machine;

/*-----------------------------------------------------------------------------*/
/* SRM variables data structure:                                               */
/*-----------------------------------------------------------------------------*/
/*a2d_chan[i]  -> sets which A/D pin is used for the ith phase current
/*desiredTorque       -> torque command (output of velocity loop)
/*integral_speed_error-> velocity loop integrator for PI compensator
/*iDes[i]             -> current command for the ith phase
/*capture_edge[i]     -> timer #2 count value at the occurence of
/*             the most recent ith capture
/*capture_delta[i][2] -> change in the timer #2 count value between
/*             the occurences of the ith capture events.  The two most
/*             recent events are stored.
/*delta_count  -> change in the software counter of the timer ISR
/*             between occurences of any capture event.
/*wEst_10xrpm  -> shaft velocity estimate (units of rpm*10)
/*     wDes_10xrpm    -> desired shaft velocity (units of rpm*10)
/*     active[i]      -> flag indicating whether the ith phase is ON (1 = on)
/*     iFB[i]         -> current feedback measurement for the ith phase
/*     dutyRatio[i]   -> commanded % duty ratio for the high-side FET of
/*             the ith phase
/*position     -> shaft position estimate (electrical degrees)
/*             scaled: 2*pi (rad) = 65535 bits
/*position_state      -> position state of the SRM (defined by opto-couplers)
/*shaft_direction     -> direction which the shaft is rotating.
/*trans_lut[7][4]     -> the position state machine
/*position_initial_guess[7] -> initial position guess, based on state
/*dp_remainder -> 16-bit remainder used in the position estimation alg
/*last_capture -> the most recent capture to occur
/*-------------------------------------------------------------------------*/
typedef struct  {
  int          a2d_chan[NUMBER_OF_PHASES];
  int          desiredTorque;
  long         integral_speed_error;
  WORD         iDes[NUMBER_OF_PHASES];
  WORD                capture_edge[NUMBER_OF_PHASES];
  WORD                capture_delta[NUMBER_OF_PHASES][2];
  WORD         delta_count;
  int                 wEst_10xrpm;
  int          wDes_10xrpm;
  int          active[NUMBER_OF_PHASES];
  WORD         iFB[NUMBER_OF_PHASES];
  int          dutyRatio[NUMBER_OF_PHASES];
  WORD         position;
  int          position_state;
  int          shaft_direction;
  state_machine       trans_lut[7][4];
  WORD         position_initial_guess[7];
  long         dp_remainder;
  int          last_capture;
} anSRM_struct;
```

```
/*------------------------------------------------------------------*/
/*PROTOTYPE DEFINITIONS                                              */
/*------------------------------------------------------------------*/
void eventmgr_init();
void initializeSRM(anSRM_struct *anSRM);
void Commutation_Algorithm(  anSRM_struct *anSRM);
void Time_Update_Position(anSRM_struct *anSRM);
void velocityController(  anSRM_struct *anSRM);
void currentController(  anSRM_struct *anSRM);
void computePositionAndVelocity(anSRM_struct *anSRM);
void Msmt_Update_Velocity(anSRM_struct *anSRM, int mode);
void Msmt_Update_Position(anSRM_struct *anSRM);
void switch_lowside(int phaseactive);
void switch_mux(int adc1, int adc2);
void disable_interrupts();
void dsp_setup();
void initialize_counters_and_flags();
void enable_interrupts();
void start_background();
void check_for_stall();


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
/*File:               MAIN.C
/*Target Processor:   TMS320F240
/*Compiler Version:   6.6
/*Assembler Version:  6.6
/*Created:            10/31/97
/*
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/* This file is the main program for the control of an SRM drive with a
/*position sensor
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

/*-------------------------------------------------------------------*/
/*INCLUDE FILES                                                       */
/*-------------------------------------------------------------------*/
#include "c240.h"
#include "srm.h"

/*----------------------------------------------------------*/
/*GLOBAL VARIABLE DECLARATIONS                               */
/*----------------------------------------------------------*/
int  count;
int  slice;
int  old_count;
int  Update_Velocity;
int  Toggle_LED;
int  Msmt_Update;
anSRM_struct SRM;
int  LEDvalue;

/*--------------------------------------------------------*/
/*MAIN PROGRAM                                             */
/*--------------------------------------------------------*/

void main() {

  disable_interrupts();
  dsp_setup();
  initializeSRM(&SRM);
  eventmgr_init();
  initialize_counters_and_flags();
  enable_interrupts();

  start_background();

}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*BACKGROUND TASKS                                           */
/*--------------------------------------------------------- */
/*Upon completion of the required initialization, the main
/*program starts the background task.  The background is
/*simply an infinite loop.  Time critical motor control
/*processing is done via interrupt service routines and lower
```

```
/*priority processing is done in the background, when they
/*are needed.  Two background operations are defined:
/*
/*1) Update_Velocity - when a capture interrupt occurs,
/*      the ISR stores the capture data and then intiates
/*      this task.  The velocity update is done in
/*      background, because it is doing a floating point
/*      division.
/*2) Toggle_LED - this task toggles an LED on the EVM to
/*      provide visual feedback to the user that the code
/*      is running.  This task is initiated at a fixed
/*      rate set by the ONE_HALF_SECOND value.
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
void start_background()
{

  while (1)

  {
      /*----------------------*/
      /* Velocity update task */
      /*----------------------*/
      if (Update_Velocity) {
        if (Update_Velocity == 1) {   /* use capture data    */
                                      /*    as time base     */
           Msmt_Update_Velocity(&SRM,1);
        }
        else {    /* else shaft is rotating too slowly, capture
                  /* data may be in error by overflow.
                  /* use count of timer ISR's between captures
                  /* as time base.   */
           Msmt_Update_Velocity(&SRM,2);
        }
        Update_Velocity = 0;
     }

     /*-----------------------*/
     /* Visual feedback task  */
     /*-----------------------*/
     if (Toggle_LED) {

        LEDvalue = -LEDvalue;
        if (LEDvalue == 1) {
               asm("   OUT  1, 000ch");
        }
        else {
               asm("   OUT  0, 000ch");
        }
        Toggle_LED = 0;
        SRM.wDes_10xrpm = 6000;       /* motor speed command units = (rpm x 10)    */
                                      /*    just hard-coded here, but setup        */
                                      /*    another background task to allow       */
                                      /*    command from an external input         */
     }

  }  /* infinite loop */

}
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*TIMER ISR                                                         */
/*-----------------------------------------------------------------*/
/*
/*This interrupt service routine is executed at each
/*occurence of the maskable CPU interrupt INT3.  This CPU
/*interrupt corresponds to the event manager group B interrupts,
/*of which we enable only the timer #3 period interrupt, TPINT3.
/*The frequency, F, at which this routine is executed is specified
/*using the CPU_INT_FREQ parameter.
/*
/*The SRM control algorithms which are implemented during the
/*timer ISR are:
/*
/*      1.  Current control (frequency = F)
/*      2.  Rotor position estimation (frequency = F)
/*      3.  Commutation (frequency = F/5)
/*      4.  Velocity control (frequency = F/5)
/*
```

```
/*Additionally, time can be measured (coarsely) by counting
/*the number of executions of this ISR, which runs at a
/*known fixed rate.  This measure of time is used for several
/*reasons, including:
/*
/*- For precaution against over-current, a simple
/*test is made to determine if the rotor has stalled.
/*
/*- Also, the visual feedback task is initiated if the correct
/*amount of time has elapsed.
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void c_int3()
{

  *IFR_REG = 0x0004;                /* clear interrupt flags             */
  *IFRB = 0xff;

  currentController(&SRM);          /* current loop algorithm            */

  if (Msmt_Update) {                /* position estimation               */
     Msmt_Update_Position(&SRM);    /*     if recent capture edge        */
     Msmt_Update = 0;               /*     use this information          */
  }
  else {                            /*     else, propagate pos est       */
     Time_Update_Position(&SRM);    /*     using algorithm               */
  }

  check_for_stall();

  count = count + 1;                /* increment count                   */
  slice = slice + 1;                /* increment slicer                  */

  if (slice == 1) {
     Commutation_Algorithm(&SRM);   /* do commutation in the 1st         */
  }                                 /*     slice.                        */
  else if (slice == 2) {            /* velocity loop algorithm in        */
     velocityController(&SRM);      /*     the 2nd                       */
  }
  else if (slice == 5) {
     slice = 0;                     /* reset slicer                      */
  }

  if (count == ONE_HALF_SECOND) {   /* set flag for toggling the         */
     Toggle_LED = 1;                /*     EVM LED, if time              */
     count = 0;
  }

}

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*CAPTURE ISR                                                     */
/*-------------------------------------------------------------- */
/*
/*This interrupt service routine is executed at each
/*occurence of the maskable CPU interrupt INT4.  This CPU
/*interrupt corresponds to the event manager group C interrupts,
/*of which we enable the three capture event interrupts,
/*CAPINT1-3.  This ISR executes asynchronously and the
/*frequency of execution is dependent on the shaft speed
/*of the SRM.
/*
/*The ISR performs the following processing:
/*
/*     clear interrupt flags;
/*     determine which capture has occured;
/*     read the appropriate capture FIFO register;
/*     store capture data;
/*     set flag for position update using measurement;
/*     set flag for initiating velocity estimate
/*            update in background;
/*     return;
/*
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void c_int4()
{
  int groupc_flags;
  int capture;
  int n;
  int delta_count;
  WORD edge_time;

  *IFR_REG = 0x0008;          /* clear CPU interrupt flag   */

  /*------------------------------------------------------*/
  /* determine which capture interrupt occured and read     */
  /*    the appropriate FIFO                                */
  /*------------------------------------------------------*/
  groupc_flags = *IFRC;              /* read event manger interrupt */
                                     /*      flag register          */

  if (groupc_flags & 0x1){           /* capture #1                 */
     *IFRC = 0xf9;                   /*      clear flag register    */
     capture = 1;                    /*                            */
     edge_time = read_fifo(capture); /*      read FIFO             */
  }
  else if (groupc_flags & 0x2) {     /* capture #2                 */
     *IFRC = 0xfa;
     capture = 2;
     edge_time = read_fifo(capture);
  }
  else if (groupc_flags & 0x4) {     /* capture #3                 */
     *IFRC = 0xfc;
     capture = 3;
     edge_time = read_fifo(capture);
  }
  else {                             /* not a valid capture        */
     *IFRC = 0xff;
     capture = 0;
  }

  /*------------------------------------------------------------*/
  /* if a valid capture occured, store capture data and set flags  */
  /*    foor position and velocity estimate updates.  The most     */
  /*    recent two time intervals between edges is saved           */
  /*    to allow for some filtering of the velocity estimate.      */
  /*    The number of timer ISR's which occur between capture      */
  /*    interrupts is also checked.  When this time exceeds a      */
  /*    certain value, then the capture data could be in error     */
  /*    by an overflow, so the lower resolution delta-time         */
  /*    associated with the ISR count is used in the velocity      */
  /*    estimate calculation.                                      */
  /*------------------------------------------------------------*/
  if (capture > 0) {

     SRM.last_capture = capture;     /* save capture data          */
     n = capture-1;
     SRM.capture_delta[n][1] = SRM.capture_delta[n][0];
     SRM.capture_delta[n][0] = edge_time - SRM.capture_edge[n];
     SRM.capture_edge[n] = edge_time;

     Msmt_Update = 1;                /* position update flag       */

     /*-------------------------------------------------------*/
     /* Set flags & select time base for use with velocity update */
     /*-------------------------------------------------------*/
     delta_count = count - old_count;
     old_count = count;
     if (delta_count < 0) delta_count = delta_count + ONE_HALF_SECOND;

     if (delta_count > 100) {        /* low shaft speed use        */
                                     /*      ISR counter           */
       SRM.delta_count = delta_count;
       Update_Velocity = 2;
     }

     else {                          /* else, shaft speed ok       */
                                     /*      use 1.25MHz clk       */
       SRM.delta_count = delta_count;
       Update_Velocity = 1;
     }
  }

}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
```

```
/*UTILITY SUBROUTINES                                              */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */


/*****************************************************************/
void disable_interrupts()
{
  asm(" SETC   INTM");
}


*****************************************************************/
void dsp_setup() {

  int temp;

  /*------------------------*/
  /* Disable watchdog timer */
  /*------------------------*/
  temp = *WDCR;
  temp = temp | 0x68;
  *WDCR = temp;
  /*--------------------------------------*/
  /* initialize PLL module (10 MHz XTAL1) */
  /*--------------------------------------*/
  *CKCR1 = 0xb1;              /* 20MHz CPUCLK = 10MHz crystal       */
                              /*      and 2x PLL mult ratio         */
  *CKCR0 = 0xc3;              /* low-power mode 0,                  */
                              /*     ACLK enabled,                  */
                              /*     PLL enabled,                   */
                              /*     SYSCLK=CPUCLK/2                */
  *SYSCR = 0x40c0;

}

/*************************************************************************/
void initialize_counters_and_flags() {

  count = 0;                  /* current timer ISR count    */
  slice = 0;                  /* ISR slice count            */
  old_count = 0;              /* timer ISR count at last    */
                              /*     capture edge           */
  Toggle_LED = 0;             /* flag for visual feedback   */
                              /*     background task        */
  LEDvalue = 1;               /* current LED value          */
  Update_Velocity = 0;        /* flag for velocity update   */
                              /*     background task        */
  Msmt_Update = 0;            /* flag for mode of position  */
                              /*     estimate update        */

}

/*************************************************************************/
void enable_interrupts() {

  *IFR_REG = 0xffff;          /* Clear pending interrupts   */
  *IFRA = 0xffff;
  *IFRB = 0xffff;
  *IFRC = 0xffff;
  *IMR_REG = 0x000c;          /* Enable CPU Interrupts:     */
                              /*     INT4 & INT3            */
  *IMRA = 0x0000;             /* Disable all event manager  */
                              /*     Group A interrupts     */
  *IMRB = 0x0010;             /* Enable timer 3 period      */
                              /*     interrupt              */
  *IMRC = 0x0007;             /* Enable CAP1-CAP3 interrupts*/
  asm(" CLRC   INTM");        /* Global interrupt enable    */

}

/*************************************************************************/
void check_for_stall()
{
  int delta_count;

  /*-------------------------------------------------------------*/
  /*  The SRM is assumed to have stalled if the number of timer    */
  /*    ISR's which are executed exceeds 1000.  At F = 5 kHz        */
  /*    this corresponds to roughly 6 rpm.  If this condition       */
  /*    is detected, the opto-coupler levels are read and the       */
  /*    rotor position is re-initialized                            */
  /*-------------------------------------------------------------*/
  delta_count = count - old_count;
  if (delta_count < 0) delta_count = delta_count + ONE_HALF_SECOND;
  if (delta_count > 1000) {
     SRM.wEst_10xrpm = 0;
     SRM.position_state = *PBDATDIR & 0x7;
     SRM.position = SRM.position_initial_guess[SRM.position_state];
  }

}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*     File:                  SRM.C                                   */
/*Target Processor:   TMS320F240                                      */
/*Compiler Version:   6.6                                             */
```

```
/*Assembler Version:   6.6                                         */
/*Created:             10/31/97                                    */
/*-----------------------------------------------------------------*/
/*  This file contains the algorithms for control of anSRM using   */
/*a position sensor.  The position sensor consists of a slotted    */
/*disk and opto-couplers.                                          */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

/*-----------------------------------------------------------------*/
/*INCLUDE FILES                                                    */
/*---------------------------------------------------------------- */
#include "srm.h"
#include "c240.h"


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*TIME UPDATE OF THE ROTOR POSITION ESTIMATE                       */
/*---------------------------------------------------------------- */
/*  Between the capture events, which provide a shaft position     */
/*measurement, position is estimated according to the equation     */
/*                                                                 */
/*      theta(k) = theta(k-1) + w * delta_t;                       */
/*                                                                 */
/*where        theta = the position measurement (electrical angle) */
/*      w = the current shaft velocity estimate                    */
/*      delta_t = the execution frequency of the algorithm         */
/*                                                                 */
/*The arithmetic is performed using double precision.              */
/*                                                                 */
/*input:       old position (where 2^16 = 2*pi radians)            */
/*      w (units of rpm * 10)                                      */
/*      K (constant incorporate delta_t and units)                 */
/*                                                                 */
/*output:      new position (where 2^16 = 2*pi radians)            */
/*                                                                 */
/*pseudo-code: dp = w * K;                                         */
/*             position = position + (dp * NR)                     */
/*                                                                 */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Time_Update_Position(anSRM_struct *anSRM)
{


   long dp;    /* delta-position in mechanical angle */
   int speed;
   int temp;

   if (anSRM->wEst_10xrpm > 0) {
        dp = anSRM->wEst_10xrpm * K_POSITION_EST + anSRM->dp_remainder;
        anSRM->dp_remainder = dp & 0xffff;
  temp = (int) (dp >> 16);
        anSRM->position = anSRM->position + (temp * NR);
   }
   else {
  speed = -anSRM->wEst_10xrpm;
  dp = speed * K_POSITION_EST + anSRM->dp_remainder;
        anSRM->dp_remainder = dp & 0xffff;
  temp = (int) (dp >> 16);
        anSRM->position = anSRM->position - (temp * NR);
   }


} /* end Time_Update_Position */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*MEASUREMENT UPDATE OF THE ROTOR POSITION ESTIMATE                */
/*-----------------------------------------------------------------*/
/* At a capture interrupt, the rotor is at 1 of 6 positions.       */
/* In between interrupts, the pickoff will be at 1 of six states,  */
/* defined by the opto-couplers.  The states are defined by [zyx]  */
/* where:      z = output of opto-coupler #3                       */
/*      y = output of opto-coupler #2                              */
/*      x = output of opto-coupler #1                              */
/*                                                                 */
/*State 2: 010                                                     */
/*State 3: 011                                                     */
/*State 1: 001                                                     */
/*State 5: 101                                                     */
/*State 4: 100                                                     */
```

```
/*State 6: 110                                                      */
/*                                                                  */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  */
void Msmt_Update_Position(anSRM_struct *anSRM)
{

   int old_state, new_state;
   int cap;

   /*-------------------------------------------------------------- */
   /* Based on capture and current state, get new state from the    */
   /*   state-machine look-up table                                 */
   /*-------------------------------------------------------------- */
   cap = anSRM->last_capture;
   old_state = anSRM->position_state;
   new_state = anSRM->trans_lut[old_state][cap].state;

   /*---------------------------------------------------*/
   /* If transition is valid, update position and state  */
   /*---------------------------------------------------*/
   if (new_state != 0) { /* valid transition, update data */

  anSRM->position = anSRM->trans_lut[old_state][cap].position;
  anSRM->shaft_direction = anSRM->trans_lut[old_state][cap].direction;
  anSRM->position_state = new_state;
   }

   else { /* else, not a valid transition, use opto-coupler */
    /*  level & re-initialize position estimate             */

  anSRM->position_state = *PBDATDIR & 0x7;

   }

}
```

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  */
/*VELOCITY ESTIMATION ALGORITHM                                    */
/*---------------------------------------------------------------  */
/* This algorithm estimates the SRM shaft velocity.  It is executed */
/*after each capture interrupt is received.  If the shaft is       */
/*moving fast enough, this routine is called with mode = 1 and     */
/*the capture data is used.  Otherwise, the # of timer ISRs        */
/*which are executed between capture events is used in the         */
/*velocity calculation.                                            */
/*                                                                 */
/*Velocity is calculated according to the equation:               */
/*                                                                 */
/*     w = delta_theta / delta_t                                   */
/*                                                                 */
/*where delta_theta is known:                                      */
/*                      (7.5 mech deg between each capture)        */
/*                     (22.5 mech deg between the same capture)    */
/*and delta_t is the measured number of clock cycles.              */
/*                                                                 */
/*The algorithm is implemented in double precision and is of       */
/*the form:                                                        */
/*             w = Kx_VELOCITY_EST/count                           */
/*                                                                 */
/*where the constant Kx_VELOCITY_ESTIMATE (x=1,2) incorporates     */
/*delta_theta and other units so that                              */
/*w has units of (rpm * 10).                                       */
/*                                                                 */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
void Msmt_Update_Velocity(anSRM_struct *anSRM, int mode)
{
   DWORD a1,a2,a3,a4,a5,a6;
   DWORD sum_cnt;
   int  inst_velocity;
   long filt_velocity;

   /*-------------------------------------------- */
   /*  Obtain instantaneous velocity estimate       */
   /*-------------------------------------------- */
   if (mode == 1) {    /* use timer #2 as time base  */

  /*------------------------------------------------------------*/
  /*  FIR filter for removing once per electrical cycle effects */
  /*------------------------------------------------------------*/
       a1 = (DWORD) anSRM->capture_delta[0][0];
       a2 = (DWORD) anSRM->capture_delta[0][1];
       a3 = (DWORD) anSRM->capture_delta[1][0];
       a4 = (DWORD) anSRM->capture_delta[1][1];
       a5 = (DWORD) anSRM->capture_delta[2][0];
       a6 = (DWORD) anSRM->capture_delta[2][1];
  sum_cnt = a1+a2+a3+a4+a5+a6;

  /*---------------------------------------------------*/
  /* apply velocity = delta_theta/delta_time algorithm */
  /*---------------------------------------------------*/
  sum_cnt = K1_VELOCITY_EST/sum_cnt;
  inst_velocity = ((int) sum_cnt) * anSRM->shaft_direction;
   }

   else {      /* else, use timer ISR count as time base */

  /*---------------------------------------------------*/
  /* apply velocity = delta_theta/delta_time algorithm */
  /*---------------------------------------------------*/
  sum_cnt = K2_VELOCITY_EST/anSRM->delta_count;
       inst_velocity = ((int) sum_cnt) * anSRM->shaft_direction;
   }

   /*-------------------------------------------- */
   /*  IIR filter for smoothing velocity estimate    */
   /*-------------------------------------------- */
   filt_velocity = (ALPHA * anSRM->wEst_10xrpm)
               + (ONE_MINUS_ALPHA * inst_velocity);
   anSRM->wEst_10xrpm = (int) (filt_velocity >> 3);

} /* end, velocity estimation */


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
```

```
/*COMMUTATION ALGORITHM                                                   */
/*---------------------------------------------------------------------- */
/* A four quadrant commutation algorithm, using a fixed-dwell angle        */
/*     of 120 electrical degrees and a variable turn on angle.  With       */
/*a fixed dwell of 120 electrical degrees, only a single phase            */
/*is active at any one time.  The advance angle is calculated as          */
/*a function of speed and desired current.                                */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
void Commutation_Algorithm(anSRM_struct *anSRM)
{
   int phase;
   WORD electricalAngle;
   WORD angle;
   int channel;
   long advance;
   int whats_active;
   int desiredCurrent;
   int temp;

   /*----------------------------*/
   /* Advance angle calculation */
   /*----------------------------*/
   advance = (anSRM->wEst_10xrpm * anSRM->desiredTorque);
   advance = advance >> 9;

   /*------------------------------------------------------*/
   /* Offset for advance angle negative torque, if required */
   /*------------------------------------------------------*/
   if (anSRM->desiredTorque > 0) {
  electricalAngle = anSRM->position + (int) advance;
  desiredCurrent = anSRM->desiredTorque;
   }
   else {
  electricalAngle = anSRM->position + PI_16 - (int) advance;
  desiredCurrent = -anSRM->desiredTorque;
   }

   /*-------------------------------- */
   /* for each phase do ...           */
   /*-------------------------------- */
   whats_active = 0x0;
   for (phase=0; phase< NUMBER_OF_PHASES; phase++) {

  /*------------------------------*/
  /* 120 degree offsets for phase */
  /*------------------------------*/
  angle = electricalAngle - phase * TWOPIBYTHREE_16;

  /*---------------------------------------------------------*/
  /* turn phase on, if between desired angles and switch      */
  /*    the mux on the A/D to measure the desired             */
  /*    phase current                                         */
  /*---------------------------------------------------------*/
  if ( (angle >= (PIBYSIX_16)) && (angle < (FIVEPIBYSIX_16)) ) {
     anSRM->active[phase] = 1;
     temp = 0x1 << phase;
     channel = anSRM->a2d_chan[phase];
     switch_mux(channel,channel+8);
     anSRM->iDes[phase] = desiredCurrent;
     if (anSRM->iDes[phase] > ILIMIT) anSRM->iDes[phase] = ILIMIT;
  }
  else {
     anSRM->active[phase] = 0;
     temp = 0;
     anSRM->iDes[phase] = 0;
  }
  whats_active = whats_active | temp;

   }

   /*------------------------------------*/
   /* switch low-side FETs, as required  */
   /*------------------------------------*/
   switch_lowside(whats_active);

}
```

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*VELOCITY CONTROL LOOP ALGORITHM                                          */
/*------------------------------------------------------------------------ */
/*  The algorithm implements a PI compensator for the velocity             */
/*control of the SRM.  The PI filter limits the integrator         */
/*to prevent windup                                                */
/*                                                                         */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
void velocityController(anSRM_struct *anSRM)
{


  int speed_error;
  int integral_error;

  /*------------------------------*/
  /* calculate error signal       */
  /*------------------------------*/
  speed_error = anSRM->wDes_10xrpm - anSRM->wEst_10xrpm;

  /*------------------------------*/
  /* integrate error              */
  /*------------------------------*/
  anSRM->integral_speed_error = anSRM->integral_speed_error + (long)speed_error;

  /*------------------------------*/
  /* apply integrator limit       */
  /*------------------------------*/
  if (anSRM->integral_speed_error > INTEGRAL_LIMIT) {
       anSRM->integral_speed_error = INTEGRAL_LIMIT;
  }
  if (anSRM->integral_speed_error < -INTEGRAL_LIMIT) {
       anSRM->integral_speed_error = -INTEGRAL_LIMIT;
  }

  /*------------------------------*/
  /* PI filter                    */
  /*------------------------------*/
  integral_error = (int) ((KI*anSRM->integral_speed_error) >> 13);
  anSRM->desiredTorque = ((KP*speed_error) >> 1) + integral_error;


}   /* end  velocityController */
```

```
/******************************************************** */
/*CURRENT CONTROL LOOP ALGORITHM                          */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
void currentController(anSRM_struct *anSRM) {

   int phase;
   int ierr;

   for (phase=0; phase < NUMBER_OF_PHASES; phase++) {


  /*----------------------------------------------*/
  /* for each active phase do ...                 */
  /*----------------------------------------------*/
  if (anSRM->active[phase] > 0) {

     /*--------------------*/
     /* read A/D converter */
     /*--------------------*/
     anSRM->iFB[phase] = read_a2d(1);

     /*---------------------------*/
     /* calculate error signal       */
     /*---------------------------*/
     ierr = anSRM->iDes[phase] - anSRM->iFB[phase];

     /*---------------------------*/
     /* current loop compensation */
     /*---------------------------*/
          anSRM->dutyRatio[phase] = ILOOP_GAIN * ierr;
     anSRM->dutyRatio[phase] = (anSRM->dutyRatio[phase] >> 3);

     /*------------------*/
     /* limit duty ratio */
     /*------------------*/
     if (anSRM->dutyRatio[phase] < 0) {
               anSRM->dutyRatio[phase] = 0;
     }
     if (anSRM->dutyRatio[phase] > MAXIMUM_DUTYRATIO) {
               anSRM->dutyRatio[phase] = MAXIMUM_DUTYRATIO;
     }

  }

  /*----------------------------------------------*/
  /* else, phase is not active                    */
  /*----------------------------------------------*/
  else {
     anSRM->iFB[phase] = 0;
     anSRM->dutyRatio[phase] = 0;
  }

   } /* end for loop */

   /*----------------------------------------*/
   /* output PWM signals to high-side FET's */
   /*----------------------------------------*/
   *CMPR1 = anSRM->dutyRatio[0];
   *CMPR2 = anSRM->dutyRatio[1];
   *CMPR3 = anSRM->dutyRatio[2];

} /* end currentController */


/****************************************************************** */
/*SRM ALGORITHM INITIALIZATION                                      */
/*----------------------------------------------------------------- */
void initializeSRM(anSRM_struct *anSRM)
{

   int i,j;

   /*--------------------------------------------------------*/
   /* define mux positions for current feedback of each phase */
   /*--------------------------------------------------------*/
   anSRM->a2d_chan[0] = 1;    /* phase A current on pin ADCIN1 */
   anSRM->a2d_chan[1] = 2;    /* phase B current on pin ADCIN2 */
   anSRM->a2d_chan[2] = 3;    /* phase C current on pin ADCIN3 */
```

```
 /*------------------------------------------------------------- */
 /* Define position estimation state machine.                    */
 /*                                                              */
 /*   Given current state, i, and capture event, j, with         */
 /*   every transition (capture event), 3 parameters are defined: */
 /*          1. trans_lut[i][j].state = the new state            */
 /*          2. trans_lut[i][j].position = the shaft position    */
 /*          3. trans_lut[i][j].direction = the shaft direction  */
 /*------------------------------------------------------------- */

/*-----------------------------------------------------*/
/* fill table with zeros.  zeros will define illegal */
/*   transitions                                     */
/*-----------------------------------------------------*/
     for (i=0; i<7; i++) {
     for (j=0; j<4; j++) {
             anSRM->trans_lut[i][j].state = 0;
             anSRM->trans_lut[i][j].position = 0;
             anSRM->trans_lut[i][j].direction = 0;
     }
     }

/*------------------------------*/
/* 'new-state' definitions  */
/*------------------------------*/
     anSRM->trans_lut[1][2].state = 3;
     anSRM->trans_lut[1][3].state = 5;
     anSRM->trans_lut[2][1].state = 3;
     anSRM->trans_lut[2][3].state = 6;
     anSRM->trans_lut[3][1].state = 2;
     anSRM->trans_lut[3][2].state = 1;
     anSRM->trans_lut[4][1].state = 5;
     anSRM->trans_lut[4][2].state = 6;
     anSRM->trans_lut[5][1].state = 4;
     anSRM->trans_lut[5][3].state = 1;
     anSRM->trans_lut[6][2].state = 4;
     anSRM->trans_lut[6][3].state = 2;

/*--------------------------------------*/
/* 'shaft direction' definitions    */
/*--------------------------------------*/
     anSRM->trans_lut[1][2].direction = -1;
     anSRM->trans_lut[1][3].direction = 1;
     anSRM->trans_lut[2][1].direction = 1;
     anSRM->trans_lut[2][3].direction = -1;
     anSRM->trans_lut[3][1].direction = -1;
     anSRM->trans_lut[3][2].direction = 1;
     anSRM->trans_lut[4][1].direction = -1;
     anSRM->trans_lut[4][2].direction = 1;
     anSRM->trans_lut[5][1].direction = 1;
     anSRM->trans_lut[5][3].direction = -1;
     anSRM->trans_lut[6][2].direction = -1;
     anSRM->trans_lut[6][3].direction = 1;
```

```
  /*--------------------------------------*/
  /* 'shaft position' definitions         */
  /*--------------------------------------*/
       anSRM->trans_lut[1][2].position = TWOPIBYTHREE_16;
  anSRM->trans_lut[1][3].position = PI_16;
       anSRM->trans_lut[2][1].position = PIBYTHREE_16;
       anSRM->trans_lut[2][3].position = 0;
       anSRM->trans_lut[3][1].position = PIBYTHREE_16;
       anSRM->trans_lut[3][2].position = TWOPIBYTHREE_16;
       anSRM->trans_lut[4][1].position = FOURPIBYTHREE_16;
       anSRM->trans_lut[4][2].position = FIVEPIBYTHREE_16;
       anSRM->trans_lut[5][1].position = FOURPIBYTHREE_16;
       anSRM->trans_lut[5][3].position = PI_16;
       anSRM->trans_lut[6][2].position = FIVEPIBYTHREE_16;
       anSRM->trans_lut[6][3].position = 0;


   /*------------------------------------------------------------------- */
   /*  define initial guesses for each state.  The initial position        */
   /*  is assumed at the midpoint of each state                            */
   /*------------------------------------------------------------------- */
   anSRM->position_initial_guess[1] = TWOPIBYTHREE_16 + PIBYSIX_16;
   anSRM->position_initial_guess[2] = PIBYSIX_16;
   anSRM->position_initial_guess[3] = PIBYTHREE_16 + PIBYSIX_16;
   anSRM->position_initial_guess[4] = FOURPIBYTHREE_16 + PIBYSIX_16;
   anSRM->position_initial_guess[5] = PI_16 + PIBYSIX_16;
   anSRM->position_initial_guess[6] = FIVEPIBYTHREE_16 + PIBYSIX_16;


   /*-----------------------------------------------------*/
   /* read opto-couplers and get initial position estimate */
   /*-----------------------------------------------------*/
   anSRM->position_state = *PBDATDIR & 0x7;
   anSRM->position = anSRM->position_initial_guess[anSRM->position_state];


   /*-------------------------*/
   /* set initial conditions  */
   /*-------------------------*/
   for(i = 0; i < NUMBER_OF_PHASES; i++) {
  anSRM->iDes[i] = 0;
       anSRM->active[i] = 0;
       anSRM->iFB[i] = 0;
  anSRM->capture_delta[i][0] = 65535;
  anSRM->capture_delta[i][1] = 65535;
   }

   anSRM->wEst_10xrpm = 0;
   anSRM->shaft_direction = 0;
   anSRM->dp_remainder = 0;
   anSRM->integral_speed_error = 0;
   anSRM->wDes_10xrpm = 0;


}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
/*File:               EVMGR.C
/*Target Processor:   TMS320F240
/*Compiler Version:   6.6
/*Assembler Version:  6.6
/*Created:            10/31/97
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/* This file contains the routines for initializing and using the event
/* manager peripherials.
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

/*------------------------------------------------------------------ */
/*INCLUDE FILES                                                      */
/*------------------------------------------------------------------ */
#include "c240.h"
#include "constant.h"
```

```
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*EVENT MANAGER INITIALIZATION                                      */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Through appropriate programming of the event manager control
/*registers, this routine sets up the event manager so that:
/*
/*all timers run in the continuous up count mode
/*timer 1 provides the desired PWM frequency timebase
/*timer 2 counts at 1/16 of the CPUCLK and is used as the time
/*      base for capture events.  Prescaling prevents overflow
/*      except at only low shaft speeds.
/*timer 3 provides the CPU interrupt
/*A/D conversions are synchronized with timer 3 period occurences
/*compare units are configured to the PWM mode
/*PWMs 1,3, and 5 (used for high-side switching) are active low
/*PWMs 2,4, and 6 (used for low-side switching) are forced hi/low
/*sets up shared pins as capture inputs and digital inputs for
/*      interface with the opto-couplers
/*initiates continuous A/D conversions.
/*
/*--------------------------------------------------------------------*/
/*   GPTCON Initialization parameters
/*                                           GPTCON = 0x1055
/*
/*                                           xxx1 0000 0101 0101
/*                                           ___---__---_-- __--
/*                                            |  | |  | | |  | |
/* (15-13)     Read-only status bits ---------|  | |  | | |  | |
/* (12-11)     Start A/D on timer 3 period ------| |  | | |  | |
/* (10-9)      No timer 2 event starts A/D --------|  | | |  | |
/* (8-7)       No timer 1 event starts A/D -----------| | |  | |
/* (6)         Enable timer compares -------------------| |  | |
/* (5-4)       Timer 3 active low ------------------------|  | |
/* (3-2)       Timer 2 active low ---------------------------| |
/* (1-0)       Timer 1 active low -----------------------------|
/*


/*--------------------------------------------------------------------*/
/*   T3CON Initialization parameters
/*                                           T3CON = 0x9040
/*
/*                                           1001 0000 0100 0000
/*                                           __----___ -_-- __-_
/*                                           |  |   |  || | | ||
/* (15-14)     Stop on suspend --------------|  |   |  || | | ||
/* (13-11)     Continuous up-count mode --------|   |  || | | ||
/* (10-8)      Clock prescaler = 1 -----------------|  || | | ||
/* (7)         Use own TENABLE bit --------------------|| | | ||
/* (6)         Enable timer  ---------------------------| | | ||
/* (5-4)       Use internal clock source -----------------| | ||
/* (3-2)       Reload at zero ------------------------------| ||
/* (1)         disable timer compare -------------------------||
/* (0)         Use own period register ------------------------|
/*
/*--------------------------------------------------------------------*/
/*   T2CON Initialization parameters
/*                                           T2CON = 0x9440
/*
/*                                           1001 0100 0100 0000
/*                                           __----___ -_-- __-_
/*                                           |  |   |  || | | ||
/* (15-14)     Stop on suspend --------------|  |   |  || | | ||
/* (13-11)     Continuous up-count mode --------|   |  || | | ||
/* (10-8)      Clock prescaler = 1/16 --------------|  || | | ||
/* (7)         Use own TENABLE bit --------------------|| | | ||
/* (6)         Enable timer  ---------------------------| | | ||
/* (5-4)       Use internal clock source -----------------| | ||
/* (3-2)       Reload at zero ------------------------------| ||
/* (1)         disable timer compare -------------------------||
/* (0)         Use own period register ------------------------|
/*
/*--------------------------------------------------------------------*/
/*   T1CON Initialization parameters
/*                                           T1CON = 0x9040
/*
/*                                           1001 0000 x100 0000
/*                                           __----___ -_-- __-_
/*                                           |  |   |  || | | ||
/* (15-14)     Stop on suspend --------------|  |   |  || | | ||
/* (13-11)     Continuous up-count mode --------|   |  || | | ||
/* (10-8)      Clock prescaler = 1 -----------------|  || | | ||
/* (7)         Reserved on timer 1 --------------------|| | | ||
/* (6)         Enable timer  ---------------------------| | | ||
/* (5-4)       Use internal clock source -----------------| | ||
/* (3-2)       Reload at zero ------------------------------| ||
/* (1)         Disable timer compare -------------------------||
/* (0)         Use own period register ------------------------|
/*
/*--------------------------------------------------------------------*/
/*   COMCON Initialization parameters
/*
/*                                           COMCON = 0x8207
/*
/*                                           1000 0010 xxxx x111
/*                                           _--_ --_- -__---_-_
/*                                           | || | || | | | |||
/* (15)        Enable compares --------------| || | || | | | |||
/* (14-13)     Reload compare at 0 ------------|| | || | | | |||
/* (12)        Disable Space Vector PWM --------| | || | | | |||
/* (11-10)     Reload ACTR at 0 ------------------| || | | | |||
/* (9)         Enable full compare output pins -----|| | | | |||
/* (8)         Hi-Z simple compare output pins ------| | | | |||
/* (7)         Simple compare time base ---------------| | | |||
/* (6-5)       Simple compare reload --------------------| | |||
/* (4-3)       Simple compare SACTR reload ----------------| |||
/* (2)         Compare #3 to PWM mode -----------------------|||
/* (1)         Compare #2 to PWM mode ------------------------||
/* (0)         Compare #1 to PWM mode -------------------------|
/*
/*--------------------------------------------------------------------*/
/*   ACTR Initialization parameters
/*                                           ACTR = 0x0111
/*
/*                                           xxxx 0001 0001 0001
/*                                           ____ --__ --__ --__
/*                                            |   |  | |  | |  |
/* (15-12)     Space vector PWM related ------|   |  | |  | |  |
/* (11-10)     PWM6 = Force Low ------------------|  | |  | |  |
/* (9-8)       PWM5 = Active Low --------------------| |  | |  |
/* (7-6)       PWM4 = Force Low -----------------------|  | |  |
/* (5-4)       PWM3 = Active Low  ------------------------| |  |
/* (3-2)       PWM2 = Force Low ----------------------------|  |
/* (1-0)       PWM1 = Active Low ------------------------------|
/*
/*--------------------------------------------------------------------*/
/*   ADCTRL1 Initialization parameters
/*                                           ADCTRL1 = 0x2c00
/*
/*                                           0010 110x 0000 0000
/*                                           _-_- _-_- _--- ___-
/*                                           |||| |||| | |   | |
/* (15)        Suspend - Soft ---------------|||| |||| | |   | |
/* (14)        Suspend - Free ----------------||| |||| | |   | |
/* (13)        Start A/D Conversions ----------|| |||| | |   | |
/* (12)        Disable Channel 1 ---------------| |||| | |   | |
/* (11)        Enable Channel 2 ------------------|||| | |   | |
/* (10)        Continuous conversion  -------------||| | |   | |
/* (9)         Disable interrupt  ------------------|| | |   | |
/* (8)         ADC Interrupt flag -------------------| | |   | |
/* (7)         Conversion status ----------------------| |   | |
/* (6-4)       ADC1 mux select --------------------------|   | |
/* (3-1)       ADC2 mux select ------------------------------| |
/* (0)         Start conversion bit ---------------------------|
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void eventmgr_init() {

  WORD iperiod;

  /*--------------------------------------------------------*/
  /* Initialize GP timer 3 to provide desired CPU interrupt */
  /*--------------------------------------------------------*/
  iperiod = (SYSCLK_FREQ/CPU_INT_FREQ) - 1;

  *GPTCON = 0x1055;   /* Setup general-purpose control reg         */
  *T3PER = iperiod;    /* Load timer #2 period register             */
  *T3CON = 0x9040;     /* Initialize timer #3 control register      */

  /*-------------------------------------------------------------*/
  /* Initialize GP timer 1 to provide a 20 kHz time base for */
  /*        fixed frequency PWM generation                   */
  /*-------------------------------------------------------------*/
  iperiod = (SYSCLK_FREQ/PWM_FREQ) - 1;

  *T1PER = iperiod;    /* Load timer #1 period                      */
  *T1CON = 0x9040;     /* Initialize timer #1 control register      */

  /*---------------------------------------------------------------*/
  /* Initialize GP timer 2 to provide time base for clocking   */
  /*   capture events                                          */
  /*---------------------------------------------------------------*/
  *T2PER = 0xffff;     /* Load timer #2 period                      */
  *T2CON = 0x9440;     /* Initialize timer #2 control register      */

  /*------------------------------------*/
  /* Setup Compare units for PWM outputs */
  /*------------------------------------*/
  *ACTR = 0x0111;      /* Initialize action on output pins */
  *DBTCON = 0x0;       /* Disable deadband */
  *CMPR1 = 0x0;        /* Clear period registers */
  *CMPR2 = 0x0;
  *CMPR3 = 0x0;
  *COMCON = 0x0207;    /* Setup COMCON w/o enable */
  *COMCON = 0x8207;    /* Setup COMCON and enable */

  /*-------------------*/
  /* Setup shared pins */
  /* ------------------*/
  *OCRA = 0x0;         /* pins IOPB0-IOPB7 & IOPA0-IOPA3 to I/O pins */
  *OCRB = 0xf1;        /* pins are: ADSOC, XF, /BIO, CAP1-CAP4 */
  *PBDATDIR = 0xf0f0;  /* inputs IOPB0-IOPB3 */
                       /* outputs IOPB4-IOPB7, set high */

  /*---------------------*/
  /* Setup capture units */
  /*---------------------*/
  *CAPCON = 0x0;       /* reset capture control register */
  *CAPFIFO = 0xff;     /* Clear FIFO's */
  *CAPCON = 0xb0fc;    /* enable #1-3, use Timer2, both edges */


  /*---------------------*/
  /* Setup A/D converter */
  /*---------------------*/
  *ADCTRL1 = 0x2c00;   /* Initialize A/D control register */
  *ADCTRL2 = 0x0403;   /* Clear FIFO's, Pre-scaler = 4 */

}


/***************************************************************/
/*SWITCH A/D INPUT CHANNEL                                    */
/*-------------------------------------------------------------*/
/* Each A/D converter unit has an 8:1 input multiplexer which
/* must be selected to the desired channel, prior to sampling.
/* The channel is selected by manipulating bits
/* of the ADCTRL1 control register
/*
/* inputs:     adc1 = desired input channel for A/D #1
/*                    range:  0-7
/*             adc2 = desired input channel for A/D #2
/*                    range:  8-15
/*outputs:     none
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
void switch_mux(int adc1, int adc2)
{

  WORD ctrl_word;

  ctrl_word = 0x2c00;                      /* mask channel select bits */
  ctrl_word = ctrl_word | (adc1 << 4);     /* set ADC1 channel bits */
  ctrl_word = ctrl_word | ((adc2-8) << 1);   /* set ADC2 channel bits */
  *ADCTRL1 = ctrl_word;
  *ADCTRL2 = 0x0403;
}


/********************************************************* */
/*READ A/D FIFO REGISTER                                   */
/*-------------------------------------------------------- */
/* This routine is used to read the sampled A/D data from the
/*appropriate FIFO.  The 10-bit A/D data is stored in the
/*FIFO in bits 15-6.  A right shift of 6, limits the data
/*to the range 0-1023.
/*
/*inputs:             a2d_chan = which FIFO to read
/*                    range: 1-2
/*outputs:     inval = A/D data
/*                    range: 0-1023
/*                           0 VDC = 0 bits
/*                           5 VDC = 1023 bits
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
WORD read_a2d(int a2d_chan)
{

  WORD inval;

  if (a2d_chan == 1) {
     inval = (*ADCFIFO1 >> 6) & 0x03ff;
  }
  else if (a2d_chan == 2) {
     inval = (*ADCFIFO2 >> 6) & 0x03ff;
  }

  return inval;
}


/**********************************************************/
/*SWITCH LOW-SIDE MOSFETS                                */
/*------------------------------------------------------ */
/* The state of the low-side power MOSFETS is controlled by the
/*level on the PWM2, PWM4, and PWM6 output pins, for phases
/*A, B, and C, respectively.  Active high logic is used,
/*but since the low-side switches are used for commutation
/*instead of PWM control, we just use the force-low or
/*force-high action options.
/*
/*inputs:             phaseactive = bits 0,1, and 2 control
/*                    the state of the PWM2, PWM4, and
/*                    PWM6 output pins, respectively.
/*
/*            (ex. phaseactive = 0x5 will force PWM2 &
/*                    PWM6 high, PWM4 low )
/*outputs:     none
/*
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
void switch_lowside(int phaseactive)
{

  WORD action;

  /*---------------------------------------------------*/
  /* load action register and mask PWM2, PWM4, and PWM6 */
  /*   to force low                                    */
  /*---------------------------------------------------*/
  action = *ACTR;
  action = action & 0xf333;

  /*---------------------------------------*/
  /* Force hi PWM2 if phase0 (A) is active */
  /*---------------------------------------*/
  if (phaseactive & 0x1) {
     action = action | 0x000c;
  }

  /*---------------------------------------*/
  /* Force hi PWM4 if phase1 (B) is active */
  /*---------------------------------------*/
  if (phaseactive & 0x2) {
     action = action | 0x00c0;
  }

  /*----------------------------------------*/
  /* Force hi PWM6 if phase2 (C) is active */
  /*----------------------------------------*/
  if (phaseactive & 0x4) {
     action = action | 0x0c00;
  }

  /*------------------------------------*/
  /* Write new word to action register */
  /*------------------------------------*/
  *ACTR = action;

}
```

```
/********************************************************** */
/*READ CAPTURE FIFO REGISTERS                               */
/*--------------------------------------------------------- */
/* This routine is used to read the data from the capture FIFO
/*registers.
/*
/*inputs:              capture = which FIFO to read?
/*                     range = 1-3
/*outputs              fifo_data =
/*                     range = 0-65535
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
WORD read_fifo(int capture)
{
  WORD fifo_data;
  int fifo_status;

  if (capture == 1) {
     do {
        fifo_data = *FIFO1;            /* read value */
        fifo_status = *CAPFIFO & 0x0300;     /* read status register, mask bits */
     } while (fifo_status != 0);
        }
  else if (capture == 2) {
     do {
        fifo_data = *FIFO2;            /* read value */
        fifo_status = *CAPFIFO & 0x0c00;     /* read status register, mask bits */
     } while (fifo_status != 0);
  }
  else if (capture == 3) {
     do {
        fifo_data = *FIFO3;            /* read value */
        fifo_status = *CAPFIFO & 0x3000;     /* read status register, mask bits */
     } while (fifo_status !=0);
  }
  else {
     fifo_data = 0xffff;                     /* error, not a valid capture */
  }

  return fifo_data;
}


*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*
*                                                          *
* File:              VECTORS.ASM                           *
* Target Processor:  TMS320F240                            *
* Assembler Version: 6.6                                   *
* Created:           10/31/97                              *
*                                                          *
*----------------------------------------------------------*
*   This file contains the interrupt vectors               *
************************************************************

    .length 58
    .option T
    .option X

********************************************************** *
*   ILLEGAL INTERRUPT ROUTINE                                *
********************************************************** *;
    .text
    .def         _int_0

_int_0:   B      _int_0           ; ILLEGAL INTERRUPT SPIN
```

```
*********************************************************** *
*   INTERRUPT VECTORS                                       *
*********************************************************** *
    .sect      "VECTOR"
        .ref   _c_int0
    .ref       _c_int3
  .ref  _c_int4

        B      _c_int0        ; RESET
    B   _int_0         ; INT1
    B   _int_0         ; INT2
    B   _c_int3        ; INT3
    B   _c_int4        ; INT4
    B   _int_0         ; INT5
    B   _int_0         ; INT6
    B   _int_0         ; Reserved
    B   _int_0         ;
    B   _int_0         ;
    B   _int_0         ;
    B   _int_0         ;
    B   _int_0         ;
    B   _int_0         ;
    B   _int_0         ;
    B   _int_0         ;
    B   _int_0         ;
    B   _int_0         ; TRAP
    B   _int_0         ; NMI
    B   _int_0         ;
    B   _int_0         ;
    B   _int_0         ;

    .end


/* Linker command file for TMS320F240 EVM */

vectors.obj
evmgr.obj
srm.obj

-c             /* use ROM autoinitialization model */
-m  main.map
-o  main.out
-l  c:\2xxtools\rts2xx.lib

MEMORY
{
    PAGE 0:    VECTORS:  origin = 0x0000 length = 0x0040    /* EMIF    */
    PAGE 0:    CODE:     origin = 0x0040 length = 0x1FC0    /* EMIF    */
    PAGE 1:    MMRS:     origin = 0x0000 length = 0x0060    /* MMRS    */
               B2:       origin = 0x0060 length = 0x0020    /* DARAM   */
       B0:       origin = 0x0100 length = 0x0100    /* DARAM   */
       B1:       origin = 0x0300 length = 0x0100    /* DARAM   */
       DATA:     origin = 0xa000 length = 0x2000    /* EMIF    */

}

SECTIONS
{
    .VECTOR  > VECTORS PAGE 0
    .text    > CODE    PAGE 0
    .cinit   > CODE    PAGE 0
    .switch  > CODE    PAGE 0
    .mmrs    > MMRS    PAGE 1    /* Memory Mapped Registers */
    .data    > DATA    PAGE 1
    .bss     > DATA    PAGE 1
    .const   > DATA    PAGE 1
    .stack   > DATA    PAGE 1
    .sysmem  > DATA    PAGE 1
```

Appendix B. Software Listings for a TMS320F240-Based SRM Drive Without Position Sensor

This appendix contains the software to implement an SRM drive without a position sensor using the TMS320F240 DSP.

The files below are used for both the position sensorless drive and the drive using opto-couplers for position feedback. Their listings can be found in Appendix A.

File	Major Modules	Description
TYPEDEFS.H		header file – data type definitions
C240.H		header file – C240 register definitions
EVMGR.C		modules for event manager initialization and operating the event manager peripherials
VECTORS.ASM		interrupt vectors

Listings for these files are found in this Appendix.

File	Major Modules	Description
SL_CONST.H		header file – SRM constant defintions
SL_SRM.H		header file – SRM variable declarations
SL_MAIN.C	main( )	supervisory program
	c_int3( )	timer ISR
SL_SRM.C	Sensorless_Commutation ( )	positing sensorless, single-quadrant SRM commutation algorithm.
		velocity estimation algorithm
	Msmt_Update_Velocity( )	shaft velocity loop compensation algorithm
	velocityController( )	phase current loop compensation algorithm
	currentController( )	
FLUX_EST.C	estimateFluxLinkage( )	training and calibration algorithm for estimating the flux-linked vs. current characterstics of the SRM at the aligned position.
	leastSquaresFit( )	linear least squares fit algorithm.
	get_alignedFlux( )	using the estimated characteristics, given measured current, this function returns the flux-linked at the aligned position.
	update_flux_estimate( )	a discretized version of Faraday’s law used to estimated the phase flux.
SL_LINK.CMD		linker command file

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*                                                                        */
/*File:               SL_CONST.H                                          */
/*Target Processor:   TMS320F240                                          */
/*Compiler Version:                                                       */
/*Assembler Version:                                                      */
/*Created:                   10/1/97                                      */
/*                                                                        */
/*------------------------------------------------------------------------*/
/*  Constants for the SRM control algorithms                              */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

/*---------------------------------------------*/
/* clock frequencies and time related constants */
/*---------------------------------------------*/
#define PWM_FREQ              20000          /* PWM frequency (Hz)         */
#define SYSCLK_FREQ           20000000       /* DSP clock frequency (Hz)   */
#define CPU_INT_FREQ          6000           /* timer ISR frequency (Hz)   */
#define ONE_HALF_SECOND       (CPU_INT_FREQ/2)
#define TWENTY_MSEC           (CPU_INT_FREQ/50)
#define NUM_20MS              3

/*------------------------------------------*/
/* calibration/training algorithm constants  */
/*------------------------------------------*/
#define ALIGN_CURRENT         300            /* current to align phase: 1023=4.27A */
#define MAX_TEST_CURRENT      320            /* max test current: 1023=4.273A      */
#define NUM_POINTS            80             /* total number of test points/phase  */
#define VBUS                  174            /* Vdc (Volts) x 1.024                */
#define R_PHASE               8              /* phase resistance (Ohms)            */

/*------------------------------------*/
/* current loop algorithm constants   */
/*------------------------------------*/
#define ILOOP_GAIN            22             /* Q3: gain = 2.75                    */
#define MAXIMUM_DUTYRATIO     999            /* limit on the PWM duty cycle:       */
                                             /*    100% = (SYSCLK_FREQ/PWM_FREQ-1) */
#define ILIMIT                1023           /* current limit: (1023 = 4.273 A)    */

/*------------------------------------*/
/* commutation algorithm constants    */
/*------------------------------------*/
#define ALPHA                 5              /* Q3: alpha = 0.625                  */
#define MIN_DECISION_CURRENT  120            /* min decision current (1023=4.273A) */

/*--------------------------------------*/
/* velocity loop algorithm constants    */
/*--------------------------------------*/
#define KI                    1              /* (Q14 x 1200): Ki = 0.073           */
#define KP                    1              /* Q2: Kp = 0.25                      */
#define SPEED_THRESHOLD       4000           /* reduce velocity bw below 400 rpm   */
#define INTEGRAL_LIMIT        16384000
#define MIN_TORQUE_COMMAND    150

/*------------------------------------------*/
/* velocity estimation algorithm constants   */
/*------------------------------------------*/
#define K_VELOCITY_EST        150000
#define BETA                  15             /* Q4: beta = 0.9375   */
#define ONE_MINUS_BETA        (16-BETA)      /* Q4: 1-beta = 0.0625 */
```

```
/*------------------------------------*/
/* motor geometry related             */
/*------------------------------------*/
#define NR                            8      /* number of rotor poles */
#define NUMBER_OF_PHASES              3
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*                                                                       */
/*File:               SL_SRM.H                                           */
/*Target Processor:   TMS320F240                                         */
/*Compiler Version:                                                      */
/*Assembler Version:                                                     */
/*Created:                   10/1/97                                     */
/*                                                                       */
/*-----------------------------------------------------------------------*/
/*  Variable declarations for the SRM control algorithms                 */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

#include "constant.h"
#include "typedefs.h"

/*-----------------------------------------------------------------------*/
/*SRM variables data structure:                                          */
/*-----------------------------------------------------------------------*/
/*fluxEstimate[i]      -> estimated flux linked for the ith phase
/*b0[i]        -> flux-linked vs. current estimated at the aligned
/*                     position in the form y = b1*x + b0;  this
/*                     is the estimated b0 for the ith phase
/*b1[i]        -> estimated b1 coefficient for the ith phase.
/*                     corresponds to aligned inductance
/*bias[i]              -> flux estimation bias correction for the ith phase
/*df1_remainder        -> 16-bit remainder used in the flux-estimation alg
/*df2_remainder        -> 16-bit remainder used in the flux-estimation alg
/*waitFlag     -> flag indicating a software wait, used for waits
/*                     during the training/calibration algorithm
/*waitCount    -> counter used for software waits during the training
/*                     & calibration algorithm
/*delta_current        -> change in desired current between test points of
/*                     the training/calibration algorithm
/*alignedCurrent[j]    -> the measured phase current for the jth test
/*                             point during training/calibration
/*alignedFlux[j]               -> the estimated phase flux for the jth test
/*                             point during trainig/calibration
/*minFlux              -> minimum limit for the flux estimate
/*latch_current        -> measured current at the end of an estimation period,
/*                     used during training/calibration
/*latch_flux   -> flux estimate at the end of an estimation period,
/*                     used during training/calibration
/*a2d_chan[i]  -> sets which A/D pin is used for the ith phase current
/*desiredTorque        -> torque command (output of velocity loop)
/*integral_speed_error -> velocity loop integrator for PI compensator
/*     iDes[i]         -> current command for the ith phase
/*delta_count  -> change in the software counter of the timer ISR,
/*                     between commutations
/*last_count   -> count of the software counter of the timer ISR at
/*                     the most recent commutation
/*wEst_10xrpm  -> shaft velocity estimate (units of rpm*10)
/*wDes_10xrpm  -> desired shaft velocity (units of rpm*10)
/*Active               -> indicates which phase is current active 0,1, or 2
/*iFB[i]               -> current feedback measurement for the ith phase
/*dutyRatio[i] -> commanded % duty ratio for the high-side FET of the
/*                     ith phase
/*Update_Velocity      -> flag to initiate the executation of the velocity estimation
/*                     algorithm in background
/*-----------------------------------------------------------------------*/
typedef struct  {
  long  fluxEstimate[NUMBER_OF_PHASES];
  int   b0[NUMBER_OF_PHASES];
  int   b1[NUMBER_OF_PHASES];
  int   bias[NUMBER_OF_PHASES];
  long  df1_remainder;
  long  df2_remainder;
  int   waitFlag;
  int   waitCount;
  int   delta_current;
  WORD  alignedCurrent[NUM_POINTS];
  long  alignedFlux[NUM_POINTS];
  long  minFlux;
  WORD  latch_current;
  long  latch_flux;
  int   a2d_chan[NUMBER_OF_PHASES];
  int   desiredTorque;
  long  integral_speed_error;
  WORD  iDes[NUMBER_OF_PHASES];
  int   delta_count;
  int   last_count;
  int          wEst_10xrpm;
  int   wDes_10xrpm;
  int   Active;
  WORD  iFB[NUMBER_OF_PHASES];
  WORD  dutyRatio[NUMBER_OF_PHASES];
  int   Update_Velocity;
} anSRM_struct;


/*-----------------------------------------------------------------*/
/*PROTOTYPE DEFINITIONS                                            */
/*-----------------------------------------------------------------*/
void eventmgr_init();
void initializeSRM(anSRM_struct *anSRM);
void velocityController(anSRM_struct *anSRM);
void currentController(anSRM_struct *anSRM);
void Sensorless_Commutation(anSRM_struct *anSRM, int count);
void switch_lowside(int phaseactive);
void switch_mux(int adc1, int adc2);
void estimateFluxLinkage(anSRM_struct *anSRM);
void disable_interrupts();
void dsp_setup();
void initialize_counters_and_flags();
void enable_interrupts();
void start_background();
long get_alignedFlux(anSRM_struct *anSRM, int current);
void update_flux_estimate(anSRM_struct *anSRM);


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
/*File:                SL_MAIN.C
/*Target Processor:    TMS320F240
/*Compiler Version:    6.6
/*Assembler Version:   6.6
/*Created:                    10/31/97
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/* This file is the main program for the control of an SRM drive without */
/*using a position sensor                                               */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/


/*----------------------------------------------------------*/
/*INCLUDE FILES                                             */
/*----------------------------------------------------------*/
#include "c240.h"
#include "srm.h"
```

```
/*---------------------------------------------------------*/
/*GLOBAL VARIABLE DECLARATIONS                             */
/*---------------------------------------------------------*/
int  count;
int  slice;
int  Learn_Flux_Curves;
int  Toggle_LED;
anSRM_struct SRM;
int  LEDvalue;


/*----------------------------------------------------------*/
/*MAIN PROGRAM                                              */
/*----------------------------------------------------------*/
void main() {

  disable_interrupts();
  dsp_setup();
  initializeSRM(&SRM);
  eventmgr_init();
  initialize_counters_and_flags();
  enable_interrupts();

  start_background();

}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*BACKGROUND TASKS                                                 */
/*-----------------------------------------------------------------*/
/*Upon completion of the required initialization, the main
/*program starts the background task.  The background is
/*simply an infinite loop.  Time critical motor control
/*processing is done via interrupt service routines and lower
/*priority processing is done in the background, when they
/*are needed.  Three background operations are defined:
/*
/*1) Learn_Flux_Curves - this is the initial calibration task.
/*      The flux-linkage vs. current data for each phase
/*      is estimated during this task.  This task runs one
/*      time and is initiated by setting the LEARN_FLUX_CURVES
/*      flag.
/*2) Update_Velocity - when a capture interrupt occurs,
/*      the ISR stores the capture data and then intiates
/*      this task.  The velocity update is done in
/*      background, because it is doing a floating point
/*      division.
/*3) Toggle_LED - this task toggles an LED on the EVM to
/*      provide visual feedback to the user that the code
/*      is running.  This task is initiated at a fixed
/*      rate set by the ONE_HALF_SECOND value.
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void start_background()
{


  while (1)
  {

     /*------------------------------------*/
     /*  Flux-vs-current estimation task  */
     /*------------------------------------*/
     if (Learn_Flux_Curves) {
               estimateFluxLinkage(&SRM);
               Learn_Flux_Curves = 0;
     }

     /*------------------------------------*/
     /*  Velocity update task             */
     /*------------------------------------*/
     if (SRM.Update_Velocity) {
               Msmt_Update_Velocity(&SRM);
               SRM.Update_Velocity = 0;
     }

     /*------------------------------------*/
     /*  Visual feedback task             */
     /*----------------------------------*/
     if (Toggle_LED) {
               LEDvalue = -LEDvalue;
               if (LEDvalue == 1) {
                      asm("        OUT     1, 000ch");
                             }
               else {
                      asm("        OUT     0, 000ch");
               }
               Toggle_LED = 0;
        SRM.wDes_10xrpm = 6000;    /*   motor speed cmd units = (rpm x 10)  */
                                  /*    just hard-coded here, but setup    */
                                  /*    another background task to allow   */
                                  /*    command from an external input     */
     }
  }

}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*TIMER ISR                                                                  */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*
/*This interrupt service routine is executed at each occurence of the
/*maskable CPU interrupt INT3.  This CPU interrupt corresponds to the
/*event manager group B interrupts, of which we enable only the timer #3
/*period interrup, TPINT3.  The frequency, F, at which this routine is
/*executed is specified using the CPU_INT_FREQ parameter.
/*
/*While the calibration background task is active, the SRM control
/*algorithms which are implemented during the timer ISR are:
/*
/*      1. Current control (frequency = F)
/*
/*During normal operation, the SRM control algorithms which are implemented
/*during the timer ISR are:
/*
/*      1. Current control (frequency = F)
/*      2. Commutation (frequency = F)
/*      3. Velocity control (frequency = F/5)
/*
/*Additionally, time can be measured by counting the number of executions
/*of this ISR, which runs at a known, fixed rate.  This measure of time
/*is used for several reasons including:
/*
/*      - velocity estimation
/*      - initiate the visual feedback background task
/*      - software timing loops during the calibration task
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void c_int3()
{

  *IFR_REG = 0x0004;                        /* clear interrupt flags             */
  *IFRB = 0xff;

  currentController(&SRM);                  /* current loop algorithm            */

  count = count + 1;                        /* increment count                   */
  slice = slice + 1;                        /* increment slice                   */

  if (!Learn_Flux_Curves) {                 /* Normal operation, perform         */
     Sensorless_Commutation(&SRM, count);   /*    commutation algorithm          */
     if (slice == 1) {
               velocityController(&SRM);    /*      do velocity loop algorithm   */
     }                                      /*      in the 1st slice             */
  }
  else {                                    /* Else, calibration active          */
     SRM.waitCount = SRM.waitCount + 1;     /*      increment wait count         */
     if (count % TWENTY_MSEC == 0 ) {       /*      toggle software wait         */
        SRM.waitFlag = 1;                   /*      flag as needed               */
               SRM.latch_current = SRM.iFB[SRM.Active];
               SRM.latch_flux = SRM.fluxEstimate[SRM.Active];
     }
  }
  if (slice == 5) {                         /* reset slicer                     */
     slice = 0;
  }

  if (count == ONE_HALF_SECOND) {           /* set flag for toggling the        */
     Toggle_LED = 1;                        /*      EVM's LED                   */
     count = 0;
  }

}

/*****************************************************************/
void c_int4()
{
  ;
}



/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*UTILITY SUBROUTINES                                             */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

/*****************************************************************/
void disable_interrupts()
{
  asm(" SETC   INTM");
}

/*****************************************************************/
void dsp_setup() {

  int temp;

  /*------------------------*/
  /* Disable watchdog timer */
  /*------------------------*/
  temp = *WDCR;
  temp = temp | 0x68;
  *WDCR = temp;

  /*--------------------------------------*/
  /* initialize PLL module (10 MHz XTAL1) */
  /*--------------------------------------*/
  *CKCR1 = 0xb1;              /* 20MHz CPUCLK = 10MHz crystal      */
                              /*      and 2x PLL mult ratio        */
  *CKCR0 = 0xc3;              /* low-power mode 0,                 */
                              /*      ACLK enabled,                */
                              /*      PLL enabled,                 */
                              /*      SYSCLK=CPUCLK/2              */
  *SYSCR = 0x40c0;

}

/***********************************************************************/
void initialize_counters_and_flags() {

  count = 0;                  /* current timer ISR count           */
  slice = 0;                  /* ISR slice count                   */
  Toggle_LED = 0;             /* flag for visual feedback          */
                              /*      background task              */
  LEDvalue = 1;               /* current LED value                 */
  SRM.Update_Velocity = 0;    /* flag for velocity update          */
                              /*      background task              */
  Learn_Flux_Curves = 1;      /* flag for initial calibration      */
                              /*      background task              */

}

/***********************************************************************/
void enable_interrupts() {

  *IFR_REG = 0xffff;          /* Clear pending interrupts   */
  *IFRA = 0xffff;
  *IFRB = 0xffff;
  *IFRC = 0xffff;
  *IMR_REG = 0x0004;          /* Enable CPU Interrupts:     */
```

```
                              /*      INT3                    */
  *IMRA = 0x0000;             /* Disable all event manager  */
                              /*      Group A interrupts     */
  *IMRB = 0x0010;             /* Enable timer 3 period      */
                              /*      interrupt              */
  *IMRC = 0x0000;             /* Disable Group C interrupts */
  asm(" CLRC    INTM");       /* Global interrupt enable    */

}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
/*File:                 SL_SRM.C
/*Target Processor:     TMS320F240
/*Compiler Version:     6.6
/*Assembler Version:    6.6
/*Created:              10/31/97
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*  This file contains the algorithms for control of an SRM without
/*use of a position sensor.
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

/*-------------------------------------------------------------------*/
/*INCLUDE FILES                                                      */
/*-------------------------------------------------------------------*/
#include "c240.h"
#include "srm.h"
```

```
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*COMMUTATION ALGORITHM                                                    */
/*-------------------------------------------------------------------------*/
/*  This is a single-quadrant, one-phase at a time type of commutation
/*algorithm.  The algorithm compares the estimated phase flux to a
/*switching flux.  When the estimated phase flux exceeds the switching
/*flux, then a commutation to the next phase occurs.  The switching flux
/*represents the expected flux at a "good" switching angle, for the
/*measured value of the phase current.  The location of the switching
/*angle is determined by the constant ALPHA, 0 < ALPHA < 1.
/*
/*Also, to avoid the problem of this type of position sensorless
/*commutation at low current levels, the commutation condition is checked
/*only if the measured phase current exceeds a minimum threshold.
/*
/*If a commutation occurs, then the number of timer ISR's which were
/*executed since the last commutation is saved for use in the velocity
/*estimation algorithm, and the flag for running the velocity estimate
/*background task is set.
/*
/*----------------------------------------------------------------------------*/
void Sensorless_Commutation(anSRM_struct *anSRM, int count)
{
   int active, next;
   long alignedFlux;
   long switchingFlux;
   int channel;
   int current;

   active = anSRM->Active;                          /* the active phase           */
   current = anSRM->iFB[active];                    /* the current measurement    */

   if (current > MIN_DECISION_CURRENT) {     /* if above current threshold check   */
                                             /*      commutation condition         */

  alignedFlux = get_alignedFlux(anSRM, current);    /* get aligned flux for given */
                                             /*      current msmt from model       */

  switchingFlux = (ALPHA*alignedFlux) >> 3;  /* calc switching flux as a   */
                                                    /*      percentage of the     */
                                                    /*      aligned flux          */


  if (anSRM->fluxEstimate[active]>switchingFlux)    /* check commutation condtion */
  {
     next = active + 1;                             /* switch to next phase       */
     if (next > NUMBER_OF_PHASES-1) next = 0;       /* check range of phase value */
     anSRM->Active = next;                          /* store change               */
     anSRM->fluxEstimate[active] = 0;               /* reset flux integrator      */
     anSRM->df1_remainder = 0;                      /*      and remainders        */
          anSRM->df2_remainder = 0;
     anSRM->iDes[active] = 0;                       /* current this phase to 0    */
     anSRM->iDes[next] = anSRM->desiredTorque;      /* torque to current          */
     if (anSRM->iDes[next] > ILIMIT) {              /* current limit              */
              anSRM->iDes[next] = ILIMIT;
     }
     channel = anSRM->a2d_chan[next];               /* switch A/D input mux       */
     switch_mux(channel,channel+8);
     switch_lowside(0x1 << next);                   /* switch low-side FET's      */



     anSRM->delta_count =                           /* store count of timer ISR's */
              count - anSRM->last_count;            /*    since last commutation  */
     if (anSRM->delta_count < 0) {                  /*    for velocity estimation */
              anSRM->delta_count = anSRM->delta_count
              + ONE_HALF_SECOND;
     }
     anSRM->last_count = count;
     anSRM->Update_Velocity = 1;                    /* set flag to run velocity   */
                                                    /*   estimation in background */
  }

   }

   else {                                    /* else, not above current minimum   */
                                             /*      threshold                    */
  anSRM->iDes[active] = anSRM->desiredTorque;       /* torque to current          */
  if (anSRM->iDes[active] > ILIMIT) {               /* current limit              */
     anSRM->iDes[active] = ILIMIT;
  }
   }


}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*VELOCITY ESTIMATION ALGORITHM                                                    */
/*---------------------------------------------------------------------------------*/
/*  This algorithm estimates the SRM shaft velocity.  It is executed after        */
/*each commutation occurs.                                                         */
/*                                                                                 */
/*Velocity is calculated according to the equation:                                */
/*                                                                                 */
/*      w = delta_theta / delta_t                                                  */
/*                                                                                 */
/*where delta_theta is known (15 mech deg between commutations)                    */
/*and delta_t is the time between commutations as is measured by                   */
/*counting timer ISR executions.                                                   */
/*                                                                                 */
/*The algorithm is implemented in double precision and is of                       */
/*the form:                                                                        */
/*             w = K_VELOCITY_EST/count                                            */
/*                                                                                 */
/*where the constant K_VELOCITY_ESTIMATE incorporates                              */
/*delta_theta and other units so that                                              */
/*w has units of (rpm * 10).                                                       */
/*                                                                                 */
/*---------------------------------------------------------------------------------*/

void Msmt_Update_Velocity(anSRM_struct *anSRM)
{

   DWORD sum_cnt;
   int  inst_velocity;
   long filt_velocity;

   if (anSRM->delta_count > 7) {      /* protect from divide by 0 and      */
                                      /*     estimate out of range         */

  /*---------------------------------------------------*/
  /* apply velocity = delta_theta/delta_time algorithm  */
  /*---------------------------------------------------*/
  sum_cnt = K_VELOCITY_EST/anSRM->delta_count;
       inst_velocity = ((int) sum_cnt);

  /*---------------------------------------------------*/
  /* IIR filter for smoothing velocity estimate         */
  /*---------------------------------------------------*/
  filt_velocity = (BETA * anSRM->wEst_10xrpm)
             + (ONE_MINUS_BETA * inst_velocity);
     anSRM->wEst_10xrpm = (int) (filt_velocity >> 4);
   }

}
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*VELOCITY CONTROL LOOP ALGORITHM                                                  */
/*---------------------------------------------------------------------------------*/
/*  The algorithm implements a PI compensator for the velocity                     */
/*control of the SRM.  The PI filter limits the integrator                         */
/*to prevent windup.  While the shaft speed is low, the loop bandwidth             */
/*is reduced, so that the loop remains stable when the velocity feedback           */
/*is relatively infrequent (because of the commutation scheme, velocity            */
/*"feedback" occurs only when a commutation occurs, thus at low speed,             */
/*the velocity feedback frequency is low).  Also, to help with the                 */
/*inherent algorithm difficulty that low current results in increased              */
/*commutation errors with this type of sensorless approach, a minimum              */
/*torque command is imposed at the output of this algorithm                        */
/*---------------------------------------------------------------------------------*/
void velocityController(anSRM_struct *anSRM)
{

   int speed_error;
   int integral_error;
   /*-------------------------*/
   /* calculate error signal  */
   /*-------------------------*/
   speed_error = anSRM->wDes_10xrpm - anSRM->wEst_10xrpm;

   /*-----------------------------------------------*/
   /* reduce loop bandwidth at low shaft speed      */
   /*-----------------------------------------------*/
   if (anSRM->wEst_10xrpm < SPEED_THRESHOLD) {
  speed_error = speed_error >> 2;
   }


   /*-------------------------*/
   /* integrate error         */
   /*-------------------------*/
   anSRM->integral_speed_error = anSRM->integral_speed_error + (long)speed_error;

   /*-------------------------*/
   /* apply integrator limit  */
   /*-------------------------*/
   if (anSRM->integral_speed_error > INTEGRAL_LIMIT) {
  anSRM->integral_speed_error = INTEGRAL_LIMIT;
   }
   if (anSRM->integral_speed_error < -INTEGRAL_LIMIT) {
  anSRM->integral_speed_error = -INTEGRAL_LIMIT;
   }

   /*-------------------------*/
   /* PI filter               */
   /*-------------------------*/
   integral_error = (int) ((KI*anSRM->integral_speed_error) >> 14);
   anSRM->desiredTorque = ((KP*speed_error) >> 2) + integral_error;

   /*--------------------------------*/
   /* insure a minimum phase current */
   /*--------------------------------*/
   if (anSRM->desiredTorque < MIN_TORQUE_COMMAND) {
  anSRM->desiredTorque = MIN_TORQUE_COMMAND;
   }


}   /* end  velocityController */
```

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*CURRENT CONTROL LOOP ALGORITHM                                          */
/*------------------------------------------------------------------------*/
void currentController(anSRM_struct *anSRM) {

       int phase;
       int ierr;

       phase = anSRM->Active;

  /*-------------------------------------------*/
  /* read A/D converter for current feedback    */
  /*-------------------------------------------*/
       anSRM->iFB[phase] = read_a2d(1);

  /*------------------------------------------*/
  /* update flux linkage estimate     ---       */
  /*    do here because it is a function of     */
  /*    current measurement, at time k &        */
  /*    voltage cmd (dutyRatio) at time k-1     */
  /*------------------------------------------*/
  update_flux_estimate(anSRM);

  /*-------------------------------------------*/
  /* calculate error signal                    */
  /*-------------------------------------------*/
  ierr = anSRM->iDes[phase] - anSRM->iFB[phase];
  if (ierr < 0) ierr = 0;

  /*-------------------------------------------*/
  /* current loop compensation                 */
  /*-------------------------------------------*/
  anSRM->dutyRatio[0] = 0;
  anSRM->dutyRatio[1] = 0;
       anSRM->dutyRatio[2] = 0;
  anSRM->dutyRatio[phase] = ILOOP_GAIN * ierr;
  anSRM->dutyRatio[phase] = (anSRM->dutyRatio[phase] >> 3);


  /*----------------------------------------------- */
  /* limit duty ratio                               */
  /*----------------------------------------------- */
  if (anSRM->dutyRatio[phase] > MAXIMUM_DUTYRATIO) {
       anSRM->dutyRatio[phase] = MAXIMUM_DUTYRATIO;
  }

  /*------------------------------------------------*/
  /* output PWM signals to high-side FET's          */
       /*-----------------------------------------*/
       *CMPR1 = anSRM->dutyRatio[0];
       *CMPR2 = anSRM->dutyRatio[1];
       *CMPR3 = anSRM->dutyRatio[2];

} /* end  currentController */



/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*SRM ALGORITHM INITIALIZATION                                            */
/*----------------------------------------------------------------------- */
void initializeSRM(anSRM_struct *anSRM)
{

   int i;

   /*------------------------------------------------------------ */
   /* define mux positions for current feedback of each phase      */
   /*-----------------------------------------------------------  */
   anSRM->a2d_chan[0] = 1;    /* phase A on pin ADCIN1 */
   anSRM->a2d_chan[1] = 2;    /* phase B on pin ADCIN2 */
   anSRM->a2d_chan[2] = 3;    /* phase C on pin ADCIN3 */


   /*------------------------------  */
   /* specify initial conditions     */
   /*------------------------------- */
   for (i=0; i < NUMBER_OF_PHASES; i++) {
  anSRM->iDes[i] = 0;
```

```
        anSRM->iFB[i] = 0;
  anSRM->fluxEstimate[i] = 0;
   }

   anSRM->wEst_10xrpm = 0;
   anSRM->integral_speed_error = 0;
   anSRM->desiredTorque = 0;
   anSRM->last_count = 0;
   anSRM->minFlux = -999999;
   anSRM->df1_remainder = 0;
   anSRM->df2_remainder = 0;

   /*---------------------------------------- */
   /* initialization for calibration routine */
   /*---------------------------------------- */
   for (i=0; i < NUM_POINTS; i++) {
  anSRM->alignedCurrent[i] = 0;
  anSRM->alignedFlux[i] = 0;
   }

   for (i=0; i < NUMBER_OF_PHASES; i++) {
  anSRM->b1[i] = 0;
  anSRM->b0[i] = 0;
  anSRM->bias[i] = 0;
   }

   anSRM->delta_current = MAX_TEST_CURRENT/NUM_POINTS;

}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
/*File:               FLUX_EST.C
/*Target Processor:   TMS320F240
/*Compiler Version:   6.6
/*Assembler Version:  6.6
/*Created:            10/31/97
/*
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*  This file contains the algorithms used to estimate the SRM's
/*flux-linkage vs. current characteristics at the aligned
/*position.
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

/*------------------------------------------------------------------ */
/*INCLUDE FILES                                                      */
/*------------------------------------------------------------------ */
#include "c240.h"
#include "srm.h"


/*----------------------------------------------------*/
/*MACRO DEFINITION                                    */
/*----------------------------------------------------*/
#define WAIT(time)                   \
   for (j=0; j < time; j++){         \
  anSRM->waitFlag = 0;        \
  while(!anSRM->waitFlag); }

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*CALIBRATION ROUTINE                                    */
/*-------------------------------------------------------*/
void estimateFluxLinkage(anSRM_struct *anSRM)
{
  int i,j;
  int test_current;
  int channel;
  int phase;

  /*------------------------------------*/
  /* for each phase do ...              */
  /*------------------------------------*/
  for (phase=0; phase < NUMBER_OF_PHASES; phase++) {

          channel = anSRM->a2d_chan[phase];  /* switch mux A/D for current */
          switch_mux(channel,channel+8);     /*     measurement            */
     anSRM->Active = phase;
     /*---------------------------*/
          /* rotor to aligned position */
          /* -------------------------*/
     switch_lowside((0x1 << phase));           /* turn low-side FET on             */
          anSRM->iDes[phase] = ALIGN_CURRENT; /* current command                   */
     WAIT(300);                              /* wait for some time for phase      */
                                             /*     to settle                     */

     /*---------------------------*/
          /* cycle through test points */
          /*---------------------------*/
          for (i=0; i < NUM_POINTS; i++) {

               switch_lowside(0);            /* turn low-side FET off      */
               anSRM->iDes[phase] = 0;       /* command current to zero    */
               WAIT(1);                      /* wait for current decay     */

               switch_lowside(0x1 << phase); /* turn low-side FET on              */
               test_current = (i+1)*anSRM->delta_current;   /* new test point     */
               anSRM->iDes[phase] = test_current;   /* cmd current to test value  */

               anSRM->fluxEstimate[phase] = 0;      /* reset integrator and       */
               anSRM->df1_remainder = 0;            /*      remainders            */
               anSRM->df2_remainder = 0;            /*                            */

               WAIT(NUM_20MS);                      /* wait for current rise      */

               anSRM->alignedFlux[i] = anSRM->latch_flux;   /* store flux/current data */
               anSRM->alignedCurrent[i] = anSRM->latch_current;

          }


     /*----------------------------------*/
     /* make sure rotor is still aligned */
     /*----------------------------------*/
          anSRM->iDes[phase] = ALIGN_CURRENT; /* current command                   */
     WAIT(100);                              /* wait for some time for phase      */
                                             /*     to settle                     */

     /*-------------------*/
          /* turn off phases  */
          /*-------------------*/
          *CMPR1 = 0; *CMPR2 = 0; *CMPR3 = 0;       /* all high-side FET's off    */
          switch_lowside(0);                        /* low-side FET's off         */
          anSRM->iDes[phase] = 0;                   /* current command to zero    */
     WAIT(3);                                       /* wait for current decay     */

     /*-------------------*/
          /* curve fit data   */
          /*-------------------*/
          leastSquaresFit(anSRM, phase);

  } /* end for loop */

  /*------------------------------------------------------------------     */
  /* set rotor initial conditions for start of normal operation          */
  /*--------------------------------------------------------------------*/
  anSRM->minFlux = 0;                        /* flux estimate lower limit  */
  anSRM->Active = 0;                         /* from aligned C, turn on A  */
                                             /*      for positive rotation */
  channel = anSRM->a2d_chan[anSRM->Active];  /* set A/D mux                */
  switch_mux(channel,channel+8);
  switch_lowside((0x1 << anSRM->Active));    /* turn low-side FET on       */


}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*LEAST SQUARES ESTIMATION ALGORITHM                       */
/*---------------------------------------------------------*/
/*
/*Produces a least squares estimate of the form:
/*
/*            y = b1*x + b0
/*
```

```
/*Given data vectors x and y of length N, this algorithm
/*estimates coefficients b0 and b1 such that the sum of
/*square errors (SSE) is minimum.
/*
/*               N
/*      SSE =   sum [y(i) - b0 - b1*x(i)]^2
/*              i=1
/*
/*      For this application, x is the measured current vector
/*and y is the estimated flux linked vector.
/*------------------------------------------------------------*/
void leastSquaresFit(anSRM_struct *anSRM, int phase)
{
  int i;
       float current;
       float flux;
       float sumX, sumY, sumXY, sumX2;
       float xbar, ybar;
       float b1, b0;
       float bias;

  /*----------------------*/
  /* clear summers        */
  /*----------------------*/
       sumX = 0;
       sumY = 0;
       sumXY = 0;
       sumX2 = 0;

  /*-----------------------------*/
  /* perform required summations */
  /*-----------------------------*/
       for (i=0; i < NUM_POINTS; i++) {
     current = (float) anSRM->alignedCurrent[i];
     flux = (float) anSRM->alignedFlux[i];
     sumX = sumX + current;
     sumY = sumY + flux;
     sumXY = sumXY + current*flux;
     sumX2 = sumX2 + current*current;
       }

  /*-----------------------------*/
  /* apply linear fit equations  */
  /*-----------------------------*/
  b1 = (NUM_POINTS*sumXY - sumX*sumY)/(NUM_POINTS*sumX2 - sumX*sumX);
       xbar = sumX/NUM_POINTS;
       ybar = sumY/NUM_POINTS;
       b0 = ybar - b1*xbar;

  /*-----------------------------*/
  /* determine bias correction   */
  /*-----------------------------*/
  bias = b0/(TWENTY_MSEC*NUM_20MS);

  /*-----------------------------*/
  /* store estimation results    */
  /*-----------------------------*/
       anSRM->b1[phase] = (int) b1;
       anSRM->b0[phase] = (int) b0;
       anSRM->bias[phase] = (int) bias;
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*CALCULATE ALIGNED FLUX                                     */
/*-----------------------------------------------------------*/
long get_alignedFlux(anSRM_struct *anSRM, int current)
{

  long alignedFlux;

  alignedFlux = current * anSRM->b1[anSRM->Active]; /* calc aligned flux   */

  return alignedFlux;

}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*UPDATE_FLUX_ESTIMATE                                        */
```

```
/*--------------------------------------------------------- */
void update_flux_estimate(anSRM_struct *anSRM)
{
  int phase;
       long df1, df2;
       int temp1, temp2;
       long dflux;

  phase = anSRM->Active;

  /*------------------------------------------*/
  /* update flux linkage estimate              */
  /*------------------------------------------*/
  df1 = VBUS * anSRM->dutyRatio[phase] + anSRM->df1_remainder;
  anSRM->df1_remainder = df1 & 0x3ff;
  temp1 = (int) (df1 >> 10);
  df2 = R_PHASE * anSRM->iFB[phase] * 273 + anSRM->df2_remainder;
  anSRM->df2_remainder = df2 & 0xffff;
  temp2 = (int) (df2 >> 16);
  dflux = 100*(temp1-temp2) - anSRM->bias[phase];

  anSRM->fluxEstimate[phase] = anSRM->fluxEstimate[phase] + dflux;

  if (anSRM->fluxEstimate[phase] < anSRM->minFlux ) {
       anSRM->fluxEstimate[phase] = anSRM->minFlux;
  }

}

/* FILE:       SL_LINK.CMD                                                */
/* Linker command file for F240 EVM implementation of sensorless SRM drive        */
vectors.obj
evmgr.obj
sl_srm.obj
flux_est.obj

-c            /* use ROM autoinitialization model */
-m  main.map
-o  main.out
-l  c:\2xxtools\rts2xx.lib

MEMORY
{
    PAGE 0:  VECTORS:  origin = 0x0000 length = 0x0040      /* EMIF    */
    PAGE 0:  CODE:     origin = 0x0040 length = 0x1FC0      /* EMIF    */
    PAGE 1:  MMRS:     origin = 0x0000 length = 0x0060      /* MMRS    */
             B2:       origin = 0x0060 length = 0x0020      /* DARAM   */
         B0:       origin = 0x0100 length = 0x0100          /* DARAM   */
         B1:       origin = 0x0300 length = 0x0100          /* DARAM   */
         DATA:     origin = 0xa000 length = 0x2000          /* EMIF    */

}

SECTIONS
{
    .VECTOR  > VECTORS PAGE 0
    .text    > CODE    PAGE 0
    .cinit   > CODE    PAGE 0
    .switch  > CODE    PAGE 0
    .mmrs    > MMRS    PAGE 1    /* Memory Mapped Registers    */
    .data    > DATA    PAGE 1
    .bss     > DATA    PAGE 1
    .const   > DATA    PAGE 1
    .stack   > DATA    PAGE 1
    .sysmem  > DATA    PAGE 1

}
```