SENSORLESS BRUSHLESS DC MOTOR REFERENCE DESIGN

1. Introduction

This reference design provides a hardware and software solution for Sensorless Brushless dc motors. This document includes complete schematics, printed circuit board layout, and firmware. The Sensorless Brushless dc motor reference design may be used as a starting point for motor control system designers using Silicon Laboratories MCUs, significantly reducing the design time and time to market.

Brushless dc motors consist of a permanent magnet rotor with a three-phase stator winding. Brushless dc motors evolved from conventional dc motors where the function of the brushes is replaced by electronics. Brushless dc motors offer longer life and less maintenance than conventional brushed dc motors.

Most Brushless dc motor designs historically use Hall effect sensors to measure the position of the rotor. Hall effect sensors provide absolute position information required to commutate the motor. Using Hall effect sensors provides simple, robust commutation and performance roughly comparable to brushed dc motors. One of the major barriers limiting the market penetration of Brushless dc motors has been the cost of using Hall effect sensors. The Hall effect sensors themselves are not particularly expensive. However, the Hall effect assembly adds significant expense to the cost of manufacturing the motor. Hall effect sensors also typically require 5 additional wires, adding to the installation costs.

A “Sensorless” Brushless dc motor does not have Hall effect sensors. Sensorless Brushless dc motors employ more sophisticated electronics using some alternative scheme to control the commutation of the motor. The most common scheme involves measuring the back EMF of the motor and using this information to control the commutation of the motor.

Most computer hard disc drives use a sensorless brushless dc motor. However, these small motor drives use a linear regulator to control the voltage applied to the motor. This works well for small motors, but is too inefficient to use for motors greater than a few watts. Larger motors require PWM control for efficient operation. Using PWM control makes the task of measuring the back EMF of the motor more difficult, due to noise coupled from the active windings.

Sensorless Brushless dc motors are well suited for fans and rotary pumps from a few watts up to about 1 kW. Fans and pumps’ loads are predictable and fairly well behaved. Most Sensorless Brushless dc motors do not provide the same level of dynamic speed control available from Hall effect controlled BLDC motors or dc motors. While it is theoretically possible to achieve high-performance from a sensorless BLDC motor using sophisticated vector control, most practical sensorless BLDC implementations address the much simpler fan and pump applications. This reference design is targeted for simple fan applications.

Sensorless BLDC motor drives often compete against ac induction motors for certain applications. A system designer must consider the power level, efficiency requirements, and starting requirements when choosing between BLDC motors and ac induction motors. BLDC motors are most often used for small motors ranging from 1 watt to 1 kW. AC induction motors are readily available from 250 W to 10 kW or more. BLDC motors offer potentially higher efficiency than ac induction motors. This is due to the fact that ac motors have high rotor losses while BLDC motors do not waste energy in magnetizing the rotor. AC induction motors are much easier to start than sensorless BLDC motors. The complexity of starting a fully loaded BLDC motor often makes the ac induction motor a better choice, particularly for piston pumps and compressors.

2. Theory of Operation

Brushless dc motors have a dual personality. To fully understand Sensorless Brushless dc motor control it is necessary to understand both dc motor characteristics and Stepper motor characteristics. When commutated using Hall effect sensors, the torque speed characteristics of the Brushless dc motor are virtually identical to a conventional dc motor. The ideal no load speed is a linear function of the applied voltage. The torque and motor current are at a maximum at zero speed and decrease to zero at the maximum speed. The torque-speed characteristics of a dc motor are shown in Figure 1. Under ideal conditions, the characteristics of a sensorless BLDC motor are similar.

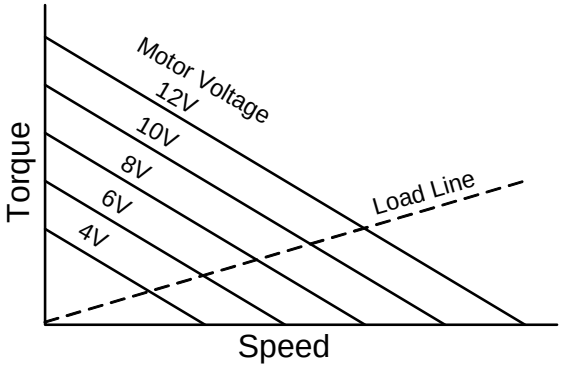


Figure 1. DC Motor Characteristics

The characteristics of a stepper motor are quite different. The motor current in a stepper motor is not proportional to the load torque. The motor winding resistance typically limits the current in a stepper motor. The motor winding resistance of a stepper motor is normally an order of magnitude higher than a BLDC motor. An ideal model for a stepper motor is shown in Figure 2.

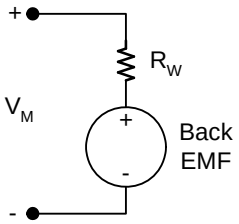


Figure 2. Stepper Motor Model

The motor is driven with a constant voltage. When the motor is at a standstill, the back EMF is zero and the current is at its maximum value. As the motor speed increases the back EMF will increase, reducing the voltage across the motor resistance, and the motor current will decrease. The available torque for a stepper motor is proportional to the current forced through the motor windings. The available torque is at its maximum value when the motor is at a standstill. The available torque decreases as the motor speed increases. If the load torque ever exceeds the available torque the motor will stall.

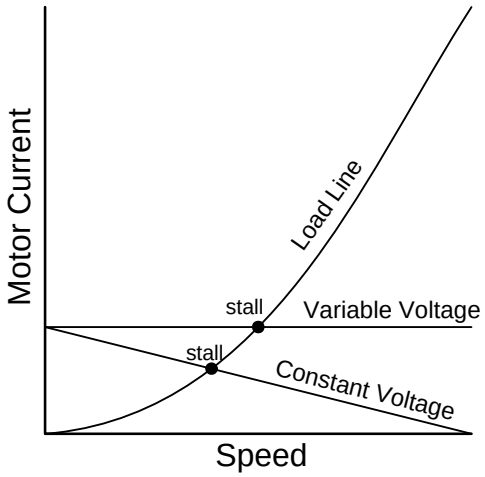


Figure 3. Stepper Motor Characteristics

The performance of the stepper motor may be improved using a variable voltage drive. By increasing the voltage applied to the stepper motor in proportion to the velocity, the available torque will remain constant and the fan can be driven to a higher velocity before stalling as shown in Figure 3.

A BLDC motor can also be considered as an unusual stepper motor. A 4-pole 12 V BLDC motor is functionally equivalent to a 12-step per revolution stepper motor with a voltage rating of about 1 V. A typical Stepper motor has a much higher step count, higher voltage, and higher resistance. A typical hybrid permanent-magnet stepper motor has a step angle of 1.8º, or 200 steps per revolution. A 12 V stepper motor might have a resistance of 30 Ω, compared to less than 1 Ω for a similar size BLDC motor. The stepper motor is optimized for precise angular positioning and constant voltage drive. The BLDC motor is optimized for Hall effect commutation and variable voltage drive.

2.1. PWM Scheme

Most Hall effect BLDC motors use a three-phase bridge to drive the motor windings. A Three-Phase bridge is shown in Figure 4.

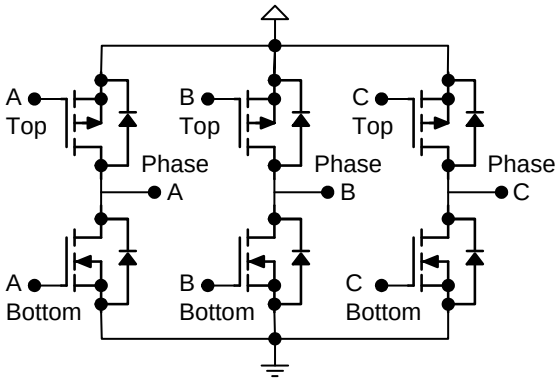


Figure 4. Three-Phase Bridge

The most common drive method is to use block commutation and apply a PWM signal to only the bottom transistors as shown in Table 1. Depending on the position of the motor and the Hall effect code, the appropriate step pattern is applied to the motor. As the motor rotates, the step pattern will increment through the table entries. Using this method, at any point in time there is only one top transistor in the continuous ON state and only one bottom transistor being driven by the PWM signal.

Table 1. Low-Side PWM Commutation

	Top			Bottom			Open
	A	**B**	**C**	**A**	**B**	**C**	**Phase**
0		ON		PWM			C
1			ON	PWM			B
2			ON		PWM		A
3	ON				PWM		C
4	ON					PWM	B
5		ON				PWM	A

Considering the motor phases, one phase is driven high, one phase is being pulse-width modulated, and one phase is open. The open phase is noted in the table. However, there is a problem using the open-phase to sense the back EMF when pulse-width modulating only the bottom transistors.

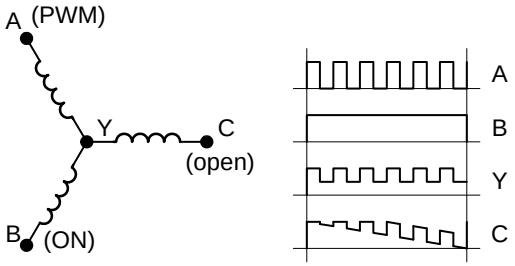


Figure 5. Low-Side PWM Back EMF

Consider the BLDC motor winding shown in Figure 5. In this case the motor is at the first state of the commutation table. A pulse-width modulated signal is applied to phase A, phase B is driven high, and phase C is open. The back EMF that we are interested in is actually the voltage from phase C to the center connection labeled point Y. The problem is that the center of the Y connection is not a constant dc voltage. Assuming that the resistance and inductance of windings A and windings B are the same, the voltage at point Y will be halfway between phase A and phase B. The voltage at point Y will be approximately half the supply voltage when phase A is driven low. When phase A is high, the center point Y will be approximately equal to the upper supply voltage. The voltage on the open phase C is now the voltage at center-point Y plus the back EMF of the motor. The voltage at phase C is also clamped to the upper rail by the MOSFET body diodes in the inverter bridge. The result is a pulse width modulated waveform where the minimum voltage level is equal to the back EMF of the motor. While it is possible to filter out the PWM signal or sample the voltage during the PWM on-time, working with this waveform is problematic.

A better approach is to use a symmetric PWM scheme. A symmetric PWM scheme is any PWM scheme where the active top and bottom transistors are turned on and off together. The simplest method is to apply identical PWM signals to the top and bottom transistors according to Table 2. This commutation table is similar to the low side PWM commutation table except that the high side transistors are pulse-width modulated instead of just being turned on.

Table 2. Symmetric PWM Commutation

	Top			Bottom			Open
	A	B	C	A	B	C	Phase
0		PWM		PWM			C
1			PWM	PWM			B
2			PWM		PWM		A
3	PWM				PWM		C
4	PWM					PWM	B
5		PWM				PWM	A

During the first state of Table 2, an identical PWM signal is applied to both the A bottom transistor and the B top transistor. The result is that Phase A will go low when Phase B goes high and visa versa. If the A and B windings are balanced, the center point Y will remain mid-rail even though phase A and phase B are being pulse-width modulated. The voltage on phase C is now roughly equal to the back EMF voltage. There may be some residual PWM noise due to the second order effects of unbalanced windings and capacitive coupling. However, the unwanted PWM signal is reduced by at least an order of magnitude, as shown in Figure 6.

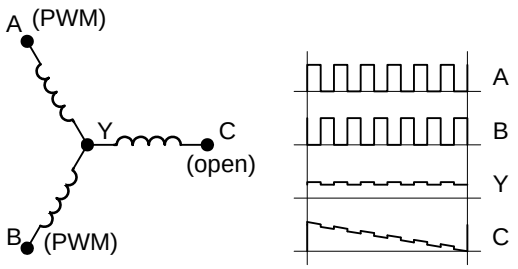


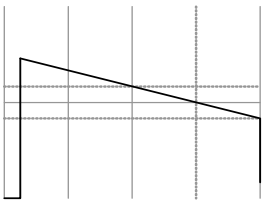
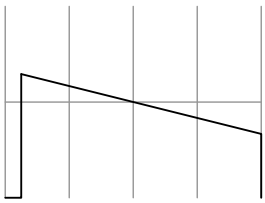
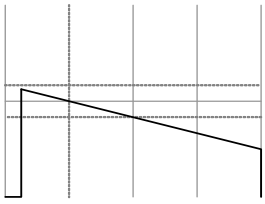
Figure 6. Symmetric PWM Back EMF

3. Back EMF Waveforms

Most sensorless BLDC control systems use the back EMF zero crossing time as a control variable for a phase locked loop. Instead of using the zero-crossing time, this reference design measures the back EMF voltage at middle of the commutation period using the ADC and uses the voltage measurement to control the commutation. This method provides higher resolution and always provides a robust feedback signal even as the motor approaches a stall condition. When using the zero-crossing time as a feedback signal, special measures must be taken when a zero crossing does not fall within the measurement window.

The back EMF voltage for three different cases is shown in Table 3. If the voltage and speed are just right the voltage ramp will be centered within the commutation period and the midpoint voltage will measure half of the supply rail. If the voltage is too low or the speed is too fast the voltage ramp will be shifted up and to the right. If the voltage is too high or the speed is too slow, the voltage ramp will be shifted down and to the left. The feedback loop should work to keep the mid-point voltage at mid-rail.

Table 3. Back EMF Control

Waveform	Speed	Voltage
	Too Fast	Too Low
	Just Right	Just Right
	Too Slow	Too High

3.1. Safe Operating Range

When driving a Sensorless BLDC motor from a MCU there are two output variables, the motor Voltage V and the motor speed ω. Figure 7 shows the output voltage verses speed characteristics for a sensorless BLDC motor. Three lines are labeled in Figure 7. The middle line is the ideal “no load” line with V equal to ω times the motor constant K_E. This line represents the optimum voltage for a perfect motor with no friction.

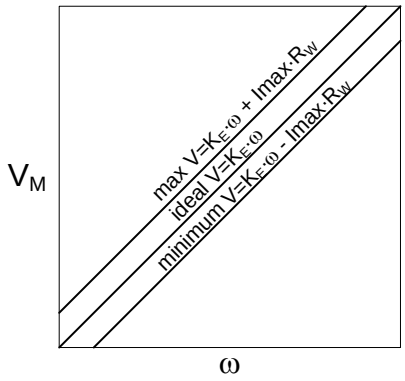


Figure 7. Output Voltage and Speed

The maximum current is normally limited by the capability of the power transistors. The maximum boost voltage is then the maximum inverter current times the winding resistance. The upper line in Figure 7 is the ideal voltage plus the maximum boost voltage. This line defines the maximum bounds for safe operation. The lower line is the ideal voltage minus the maximum boost voltage. This line defines the minimum bounds for safe operation. While not intuitive, it is quite possible to get an over-current condition by driving the motor with a voltage that is too low for a particular speed.

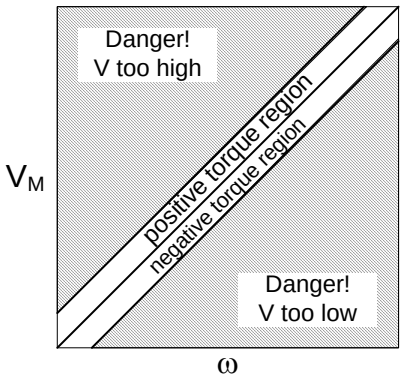


Figure 8. Safe Operating Area

The Safe Operating Area for the Sensorless BLDC motor is shown in Figure 8. Above the maximum line the voltage is dangerously too high. Below the minimum line the voltage is dangerously too low. The area between the ideal line and the max line is labeled the positive torque region. In normal steady state operation, the output variables will operate in the positive torque region.

The area below the ideal line is labeled the negative torque region. While most simple fan and pump applications do not require negative torque, operation in the negative area is permitted. If the motor speed command is decreased, the control system will output values in the negative torque region until the operating point is re-stabilized. While it is possible to prohibit operation in the negative torque region, this may adversely affect the stability of the control system.

The safe operating area is important because it is very easy to exceed the limit using conventional control techniques. If we ignore the safe operating area and use a simple PI controller to regulate the output voltage, the control loop will output excessive voltage and the power transistors may be damaged.

If we drive the motor with a voltage corresponding to the maximum line, we are forcing maximum current into the motor at all times. This will provide the maximum available torque at all times and it is unlikely that the motor will stall. However, the no load efficiency will be very low and cogging torque may be a problem. Cogging torque is the variation in torque caused by a pulsating electromotive force. Stepper motors are designed to operate on the max line with low cogging torque. Most BLDC motors are not designed for this kind of operation. Excessive cogging torque can cause vibration, noise, and stability problems.

This reference design uses a novel control technique entitled “Constant Voltage Control with Speed Dependent Limiting”. The goal of this control methodology is to provide a simple and robust control algorithm using a single control loop while keeping the output speed and voltage within the Safe Operating Area.

The control technique is illustrated in Figure 9. The input speed control potentiometer controls the target operating point indicated by the dot on the ideal line. If the error signal is zero, the output voltage and speed are set according to the speed potentiometer. As the speed control is varied, the target operating point will vary along the ideal line.

The bold line indicates variation in the output voltage and speed due to the error signal. A small positive error signal from the feedback control system will result in a decrease in speed while the voltage remains constant. The voltage will remain constant until the speed reaches the limit set by the maximum line. If the error signal is large enough to exceed the limit, both voltage and speed are reduced according to the maximum limit.

In a similar fashion, a small negative error signal will result in an increase in speed. If the negative error is increased beyond the bounds set by the minimum line, the voltage will be increased.

The resulting control loop is very stable. The motor is driven with a constant voltage, as long as the load

torque and acceleration are within the bounds set by the minimum and max limits. In this region the motor behaves like a dc motor with a constant voltage drive. A small increase in load torque will result in a small decrease in speed.

For large positive errors signals, the motor voltage is reduced to keep the current below the maximum limits. In this region the motor behaves as a constant current or constant torque drive.

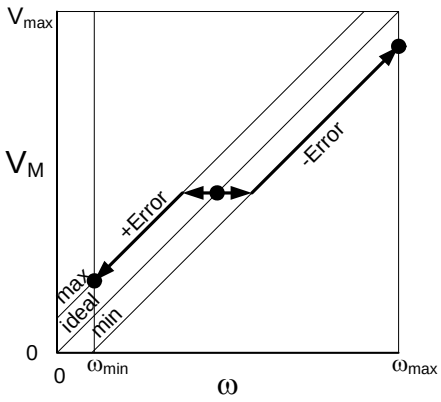


Figure 9. Constant-Voltage with Limiting

3.2. Starting

One major problem with using the back EMF to control the commutation of a sensorless BLDC motor is that the back EMF is not present or is too small to be useful until the motor is rotating at some minimum speed. The common solution is to drive the BLDC motor like a stepper motor to align the motor and accelerate the motor up to some nominal speed. This requires that the control system have multiple modes of operation. While it is generally undesirable to have a control system with multiple modes of operation, it is unavoidable in this case. A common pitfall of systems with multiple modes of operation is instability during the transition from one control mode to another. Special considerations are required in the running stage to ensure system stability during the transitions.

The motor state diagram is shown in Figure 10. The system is initially in the STOP state. Pressing the Start button will advance to the ALIGN state. The motor will then advance automatically to the START and RUN states. An over-current error will reset the motor to the STOP state.

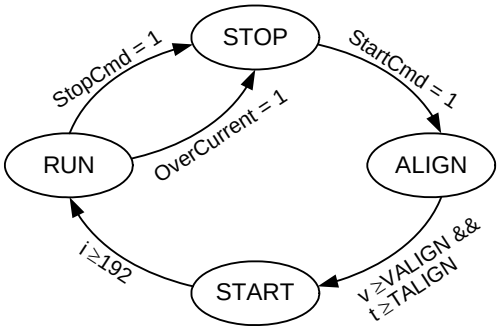


Figure 10. SBLDC State Diagram

3.3. Alignment

The alignment state consists of two stages—the Alignment Ramp and the Alignment Delay. During the Alignment Ramp, one of the motor windings is excited by PWMing the top of one phase and the bottom of another phase. The PWM duty cycle is initially at zero percent and is ramped up to 50% plus the starting boost voltage. The voltage is increased using a linear ramp with a fixed delay time between voltage increments. The current in the motor winding will depend on the motor inductance and winding resistance. For duty cycles below 50% the current will be discontinuous. The current will increase while the transistors are on and decrease to zero after the transistors turn off as shown in Figure 11.

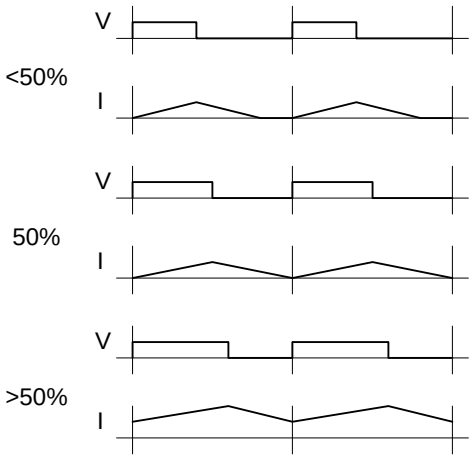


Figure 11. Motor Current vs. Duty Cycle

The peak and average dc current for duty cycles equal to or less than 50% can be calculated using Equation 1. The PWM duty cycle will always be above 50% while running.

$$I_{avg} = I_{peak} = \frac{V}{L} \times d$$

Equation 1. Peak and Average Current

For duty cycles above 50% the current will be continuous. Usually a higher current is desired for alignment. Above 50%, a small change in duty cycle will result in a much large change in average dc current. The starting boost voltage is typically a very small number.

The second part of the alignment phase consists of a simple delay. The voltage is held constant for a period of time. The alignment delay time should be long enough to allow the motor to align to the excited pole position. A one second ramp time and a two second delay time works well for the test motor with no fan. When using a large fan, a time of several seconds may be required. Finding the optimal delay time requires some experimentation.

The alignment stage ensures that the motor starts from a known position. Without the alignment stage the motor might miss the first several critical commutations in the starting acceleration ramp. If the motor does not align during this stage a motor stall is more likely in the starting phase.

During the alignment stage the system is in a relatively benign state. The dc current is predictable and well regulated. The power dissipated in the motor can be calculated from the motor winding resistance. A current measurement is only needed to protect against wiring errors.

Most BLDC motors are optimized for high-speed operation and have very limited torque in the alignment phase. This is not too much of a problem for small fans, but it does preclude the use of sensorless control in some other applications. The torque is limited by the inverter current and motor design. Over-sizing the inverter or using a lower speed motor can increase the starting torque.

3.4. Starting

During the starting phase the BLDC motor is driven like a stepper motor. The motor is commutated at first very slowly and then velocity is increased linearly using a linear velocity ramp table. The voltage is also increased in proportion to the velocity with additional boost voltage to keep the current at a constant value. The motor winding resistance normally limits the boost voltage.

The acceleration ramp table ramps the motor speed over a 30:1 range from 25 to 750 rpm. The maximum value of 750 rpm was chosen to be one tenth of the maximum speed of the motor. The closed loop sensorless control will then operate over a 10:1 range from 750 to 7500 rpm. The back EMF at 750 rpm will be one tenth of the rated voltage.

The key to getting the motor up to 750 rpm is to use a large enough acceleration ramp table with a sufficiently large ratio of minimum to maximum. A ratio of 30:1 can be achieved using a table with only 192 bytes of data. This also permits the efficient use of an 8-bit table index. Experimental results using a smaller 16-byte table with an 8:1 range proved unsatisfactory. If the table is too small, the initial velocity will be too fast or the final velocity will be too slow.

The voltage of the motor is also ramped during the acceleration phase. If the voltage were held constant at the alignment value, the current would decrease during the motor acceleration. Initially the current is only limited by the motor resistance. As the motor starts turning, the back EMF of the motor will subtract from the voltage across the motor resistance. This results in a decrease in motor current and a corresponding decrease in the amount of available torque.

Assuming the motor will accelerate in step with the commutation period, the back EMF will be proportional to the motor velocity and inversely proportional to the motor period. The motor voltage ramp may be calculated by dividing a constant by the value in the ramp table. By increasing the voltage during the acceleration phase the motor current can be held at a constant value. This ensures maximum available torque during the acceleration phase.

If the motor stalls at any point, the back EMF will be zero, and the assumption of motor velocity is no longer valid. The resulting motor voltage will be too high and the current may be much higher than expected. This is problematic for small motors and can be disastrous for large motors.

The solution is to monitor the dc motor current during acceleration and to shut down if the motor current gets too high. The back EMF is not sufficiently large to detect if the motor is moving. A current limit is not needed during the starting phase if the motor voltage is limited to the maximum inverter current times the motor resistance, though this will severely limit the starting torque on all but the smallest motors.

3.5. Running

The minimum Running speed and voltage are dictated by the starting parameters. In this example, the minimum speed is 750 rpm. The minimum voltage is 10% plus the boost voltage. At the lower operating

point, the motor is being over-driven by a voltage that is intentionally higher than necessary for stable closed loop operation. Since the motor is being driven like a stepper motor, the cogging torque and motor vibration will be higher in the starting phase.

3.6. Back EMF Measurements

The back EMF is measured at three points, as shown in Figure 12. The midpoint voltage is the primary control variable for closed loop control. The two voltage measurements at one quarter and three fourths are used for stall detection. The current measurement is used for over current detection. The time slots are discussed in further detail in the software description.

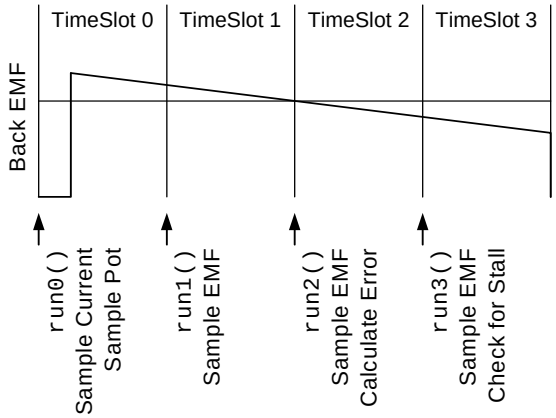


Figure 12. Back EMF Sampling

Stall detection is achieved by measuring the voltage at quarter points over two cycles. The difference between the minimum and maximum of these four points is an indication of the back EMF of the motor. If the back EMF is greater than a certain value, we can deduce with reasonable certainty that the motor is indeed running. If the back EMF is too small to give a useful measurement, we cannot know for certain if the motor is stalled or running. Thus, the stall detection operates only in the running phase. The example implementation illuminates an LED when a stall condition is detected.

4. Hardware Description

The schematic diagram for the Sensorless BLDC Motor reference design is in Appendix A. The circuit consists of the C0851F310 Microcontroller, three-phase power MOSFET bridge, three dual gate drivers, sense voltage resistive dividers, current amplifier, speed control potentiometer, two function switches, USB-UART bridge, and voltage regulator.

Port P1 is configured as push-pull outputs and is connected to the three dual gate drivers. The P1 port pins are sequenced, alternating between bottom and top starting with phase A in the least significant bit. P1.0-1.5 corresponds to A bottom, A top, B bottom, B top, C bottom, and C top respectively. This sequence facilitates commutation using a simple pattern with the Crossbar pin skip register.

Each of the three output phases are connected to a simple resistive divider. The resistive divider will divide the phase output voltage by six. This ratio permits accurate ADC measurements up to 19.8 volts. The 5 V tolerant port pins offer protection against phase voltages up to 30 Volts. The motor supply is 12 V nominal when loaded with the motor turning at full-speed. The unloaded voltage may climb as high as 18 V for the recommended wall mounted transformer.

A capacitor across the lower resistor of each divider forms simple single-pole low-pass RC filter. Each filter is tuned approximately one decade below the PWM frequency. Three test points are provided for the scaled voltages labeled VA, VB, and VC.

A forth resistive divider is used to sense the voltage of the motor supply. This resistive divider has a ratio of one to twelve. A test point is provided on the scaled motor supply voltage labeled VM. A phase voltage of 12 volts will produce voltage of 2 V on the respective ADC input, while a supply voltage of 12 V will produce a sense voltage of 1.0 V on the VM ADC input. A differential ADC voltage measurement will be used relative to the VM ADC input. The resulting ADC reading will be a signed 16-bit value with 0x0000 corresponding to mid-rail.

The sensorless BLDC motor reference design board includes a current sense resistor R25 and current amplifier U7A. The sense resistor is a 20 mΩ surface mount resistor in an 1812 package. The sense resistor was chosen for a maximum RMS current of 5 amps with a power dissipation of 500 mW. The resulting sense voltage of only 100 mV requires amplification to achieve more than 5 bits of resolution from the 10-bit ADC.

The current amplifier is a variation of the classic differential amplifier circuit. In the classic circuit configuration, the second resistors on the non-inverting input would be connected to ground. In this variation, resistor R17 is connected to a 400 mV voltage reference. The 0.4 V reference compensates for the variation in the input offset voltage of the op amp and also keeps the output within the recommended output voltage range. The gain of 26 was chosen to fully utilize the output voltage span of the amplifier allowing for variation in the input offset voltage.

The op amp chosen for this circuit is the LMV358. This is a low-voltage version of the industry standard LM358 dual CMOS op amp. The second op amp in the dual

package is used for the 400 mV reference U7B. A single zero-current calibration is used to eliminate the variation in the input offset voltage. This circuit design permits accurate differential measurements using an inexpensive CMOS op amp.

To aid development, the sensorless BLDC reference board includes a USB interface. The CP2101 USB-UART bridge IC is used in place of an RS232 transceiver chip. The CP2101 provides USB connectivity using virtual COM port drivers on the host PC. The C8051F310 UART communicates with the CP2101 using conventional asynchronous serial data. Code can be developed on the C8051F310 using the **printf()** and **scanf()** stdio library functions.

5. Software Description

The c code for the sensorless BLDC motor reference design is organized into four files:

slbdc.h

slbdc.c

T0_ISR.c

T2_ISR.c

The header file **slbdc.h** contains all of the preprocessor macros, motor parameters, and typedefs. The hardware may be used with any 12 V BLDC motor, provided that the motor parameters are changed to the appropriate values. The user-specified motor parameter macros are **MAXRPM**, **POLES**, **VMOTOR, IMOTOR, and MILLIOHMS**. These parameters are described fully in the header file. All other motor dependent constants are calculated from these values. The user need only edit the header file and recompile the code.

The **slbdc.c** file contains the **main()** function and all functions called by **main()**. The **main()** function implements initialization and the user interface. All of the time-critical tasks are handled by the interrupt service routines.

ISR Variables that are initialized or accessed by **main()** are declared in the ISR files and thus declared **extern** in **slbdc.c**.

The SBLDC user interface can be used either stand-alone or with a USB host PC running HyperTerminal.

Two push button switches labeled Start and Stop control the motor operation. Pressing the Start button will cause **main()** to call **StartMotor()**. The motor will then operate according to the state diagram. Pushing the Stop button will cause the motor to stop until restarted.

The HyperTerminal interface provides status information and allows the motor developer to modify the value of the PI controller constants on the fly. Once the start button is pressed the HyperTerminal window will display the motor state as it changes.

Aligning...

Starting...

Running...

Once the motor is running the user can display the motor status by typing an “s” on the computer keyboard. The PI controller constants can be changed using simple string commands. The proportional term KP may be changed by typing the character “p”, followed by a decimal number string. Likewise, the integral term KI may be modified by typing the character “i” followed by the new value. The ability to change the PI controller constants is very useful for development purposes. This simple user interface could easily be extended to include more parameters.

All of the motor control timing is generated using timer T0. Timer T0 is configured as a 16-bit timer module. Each time T0 overflows the interrupt service routine **T0_ISR()** will be called. The T0 interrupt is also configured as high priority so that it may interrupt other low priority tasks. The **T0_ISR()** function itself calls many other functions. All of the **T0_ISR()** related functions are located in the **T0_ISR.c** file.

The **StartMotor()** function is called from **main()** to initialize T0 and schedule the first interrupt. The **StartMotor()** function is also located in the **T0_ISR.c** file and declared extern from **main()**.

The **T0_ISR()** function itself is fairly simple. The body of the **T0_ISR()** function is a **switch** statement using the state variable **Status** that implements the state diagram. Depending on the value of **Status**, **T0_ISR()** will call the appropriate function **stop()**, **align()**, **start()**, or **run()**. The last thing the **T0_ISR()** does before returning is to update the T0 counter registers for the next period **NextT**. This is common to all states and therefore included in the **T0_ISR()** function.

The **stop()** function is very simple. It disables both PWMs, disables T0 interrupts, and stops the timer. The motor will coast down safely with all transistors disabled. Either pushing the stop button or an over-current error can set the **Status** to **STOP**.

The **align()** schedules periodic interrupts every 10 ms. It uses two static counters “v” and “d” for the “voltage ramp” and “delay” respectively. Each time the **align()** function is called it will increment the “v” counter and output the new voltage until the limit is reached. Once the “v” counter reaches its maximum value, subsequent calls to **align()** will increment the “d” counter. Once the delay counter d has reached the prescribed delay time, the **Status** state variable is incremented to **START** and the start function variables are initialized. The next time **T0_ISR()** is executed it will call the **start()** function

and not the **align()** function.

The start function uses a modulo 8 time slot manager. The motor is commutated only on time slot zero. This allows longer period values during the start period without changing the timer clock.

Only time slot zero and time slot seven are used. The **start0()** function is called on time slot zero. The **start0()** function calculates the next period **NextT** required to generate the desired linear-velocity profile.

The **start0()** function will be executed 192 times, once for each entry in the acceleration table. The table value is used to calculate the next period and output voltage. Each time the table index **AccIndex** is incremented.

The **start7()** function is called on the last time slot of the start function. The **start7()** function tests the value of the **AccIndex** to determine if the Start phase is complete. If the start phase has completed the state variable **Status** is set to **RUN**.

The **run()** function uses a modulo 4 time slot manager. The motor is commutated only on time slot zero. Each time the timeslot index is incremented and masked to 2 bits (modulo 4). Each of the four time slots has a corresponding function named **run0()**, **run1()**, **run2()**, and **run3()**. The time slot manager facilitates the scheduling of different tasks for the different time slots.

Each of the four time slot **run** functions read the value of the ADC, store the value in the appropriate location, and configure the ADC for the next time slot.

The back EMF is sampled on time slots 1, 2, & 3. The ADC is configured to initiate conversion on T0 overflow. This ensures that the first sample will occur precisely at the correct time with respect to the commutation. The ADC is reconfigured for the next sample at the end of each timeslot. Thus, the back EMF samples are scheduled at the end of time slots 0,1, & 2.

To improve resolution and reduce the effects of PWM noise, the back EMF is sampled 8 times. The initial sample is synchronized to T0 and the remaining 7 data points are sampled at the maximum sampling rate of the ADC.

The motor current and potentiometer are sampled during time slot zero, since the back EMF is not sampled during this time slot.

The calculations are distributed among the three time slots to balance the length of each function. The average midpoint and error voltage are calculated in time slot 2. The calculations related to stall detection are performed in time slot 3.

Timer T2 is configured as a 16-bit auto-reload timer generating a periodic interrupt every 1 ms. The **T2_ISR()** is used for the PI controller and ramp controller.

The T0 interrupt period varies over a 300 to 1 range. Conventional control theory is based on a discrete time system with a constant sampling rate. The PI controller should be executed at a constant rate. An acceleration controller that limits the rate of change for the speed control potentiometer also requires a constant sampling rate. The **T2_ISR()** includes code to limit the acceleration and calls the PI controller function. The PI controller is optimized the C8051. The performance of the PI controller can be further improved by fixing the coefficients as macro constants, instead of variables, once the appropriate values are determined.

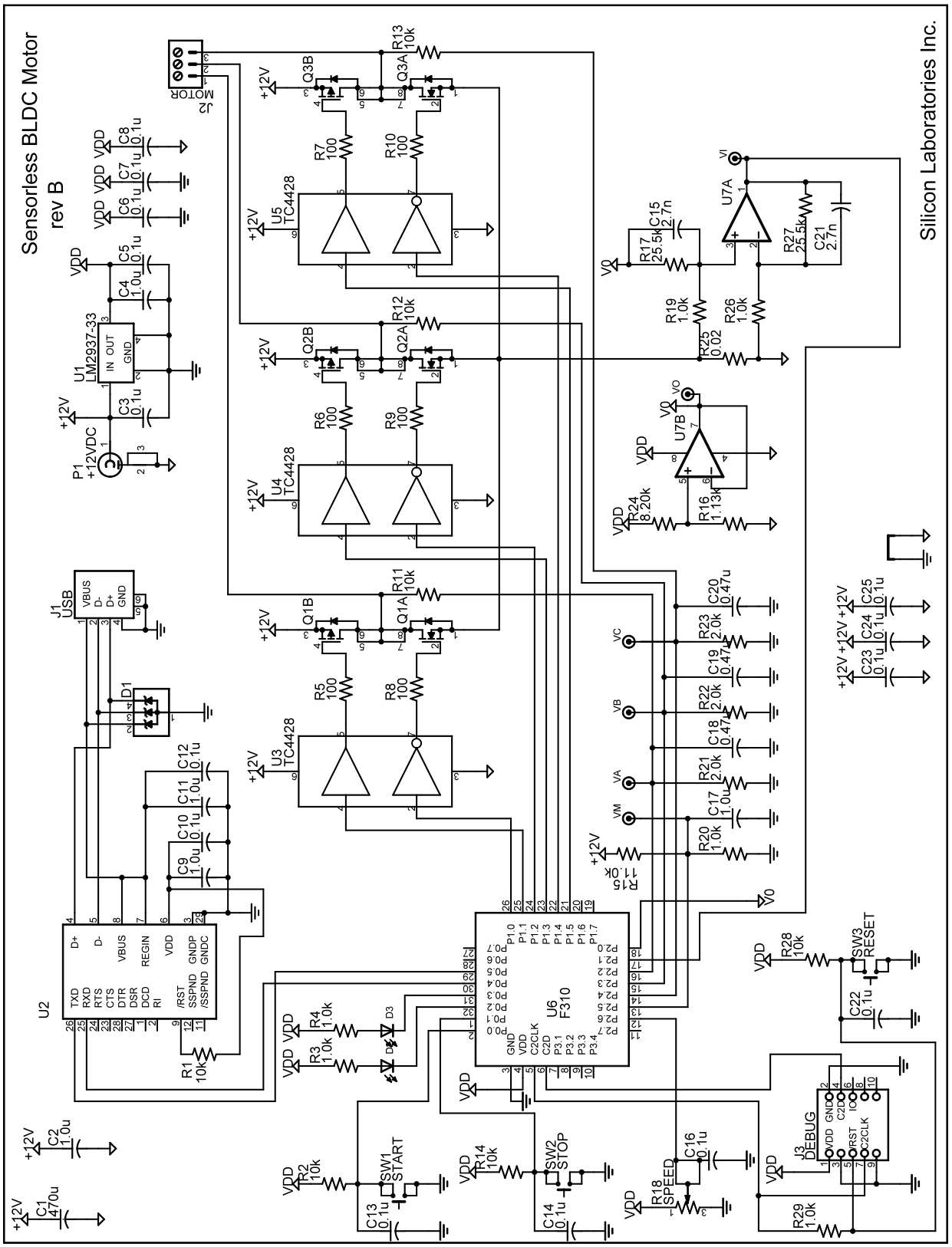
6. Summary

Brushless dc motors provide better reliability and longer life than dc motors. Sensorless control of Brushless dc motors eliminates the Hall effect sensors and reduces the system cost.

The high performance 25 MIPs C8051 core and high sample rate 10-bit ADC of the Silicon Laboratories C8051F310 MCU makes it ideally suited for Sensorless Brushless dc motor control.

The Sensorless Brushless dc motor reference design provides a complete system solution for Sensorless Brushless dc motor Fan and Pump applications.

APPENDIX A—SCHEMATIC



APPENDIX B—BILL OF MATERIALS

Table 4. Sensorless Brushless DC Motor Reference Design Bill of Materials

Qty	Designators	Description	Value	Package	PN	Mfr
1	U1	Voltage Regulator	3.3 V	SOT223	LM2937IMP-3.3	National
1	U2	USB-to-UART Bridge		MLP20	CP2101	Silicon Laboratories
3	U3, U4, U5	Gate Driver IC		SO8	TC4428COA	Maxim
1	U6	Small Form Factor MCU		QFN32	C8051F310	Silicon Laboratories
1	U7	Low-Voltage Op-Amp		SO8	LMV358M	National
3	Q1, Q2, Q3	30V Complementary MOSFET		SO8	IRF7309	International Rectifier
1	D1	Triple TVS Diode		SOT-143	SP0503BAHT	Littlefuse
2	D2, D3	LED				
1	C1	Electrolytic Capacitor	470 µF	10x16mm		Panasonic
1	C2	Chip Capacitor (25V)	1.0 µF	1206	PCC-1893CT	
14	C3, C5, C6, C7, C8, C10, C12, C13, C14, C16, C22, C23, C24, C25	Chip Capacitor	0.1 µF	805		
4	C4, C9, C11, C17	Chip Capacitor (10V)	1.0 µF	805		
3	C18, C19, C20	Chip Capacitor	.47 µF	805		
2	C15, C21	Chip Capacitor	2.7 nF	805		
5	R1, R11, R12, R13, R28	Chip Resistor	10 kΩ	805		
8	R2, R3, R4, R14, R19, R20, R26, R29	Chip Resistor	1.0 kΩ	805		
6	R5, R6, R7, R8, R9, R10	Chip Resistor	100	805		
1	R15	Chip Resistor	11.0 kΩ	805		
1	R16	Chip Resistor	1.13 kΩ	805		
2	R17, R27	Chip Resistor	25.5 kΩ	805		
1	R18	Thumb wheel Pot	10 kΩ			
3	R21, R22, R23	Chip Resistor	2.0 kΩ	805		
1	R24	Chip Resistor	8.20 kΩ	805		
1	R25	Low-Ohmage Resistor	0.02 Ω	1812	P20NBCT-ND	

Table 4. Sensorless Brushless DC Motor Reference Design Bill of Materials (Continued)

Qty	Designators	Description	Value	Package	PN	Mfr
3	SW1, SW2, SW3	6 mm push button switch				
1	J1	USB connector				
1	J2	3x5 mm terminal block				
1	J3	2x5x100 shrouded header				
1	P1	2.1 mm power connector				

APPENDIX C—PCB LAYERS

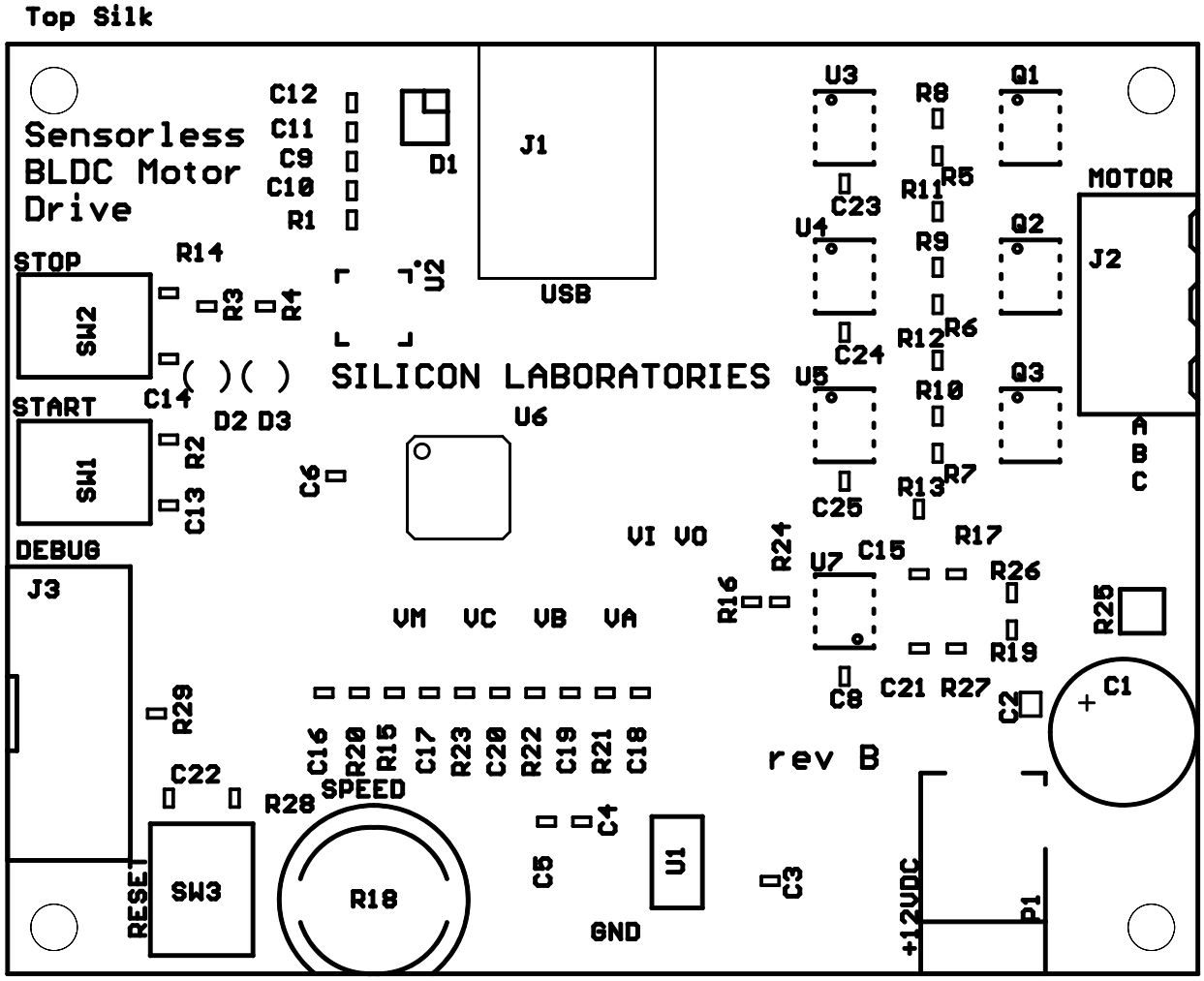


Figure 13. PCB Silkscreen Layer

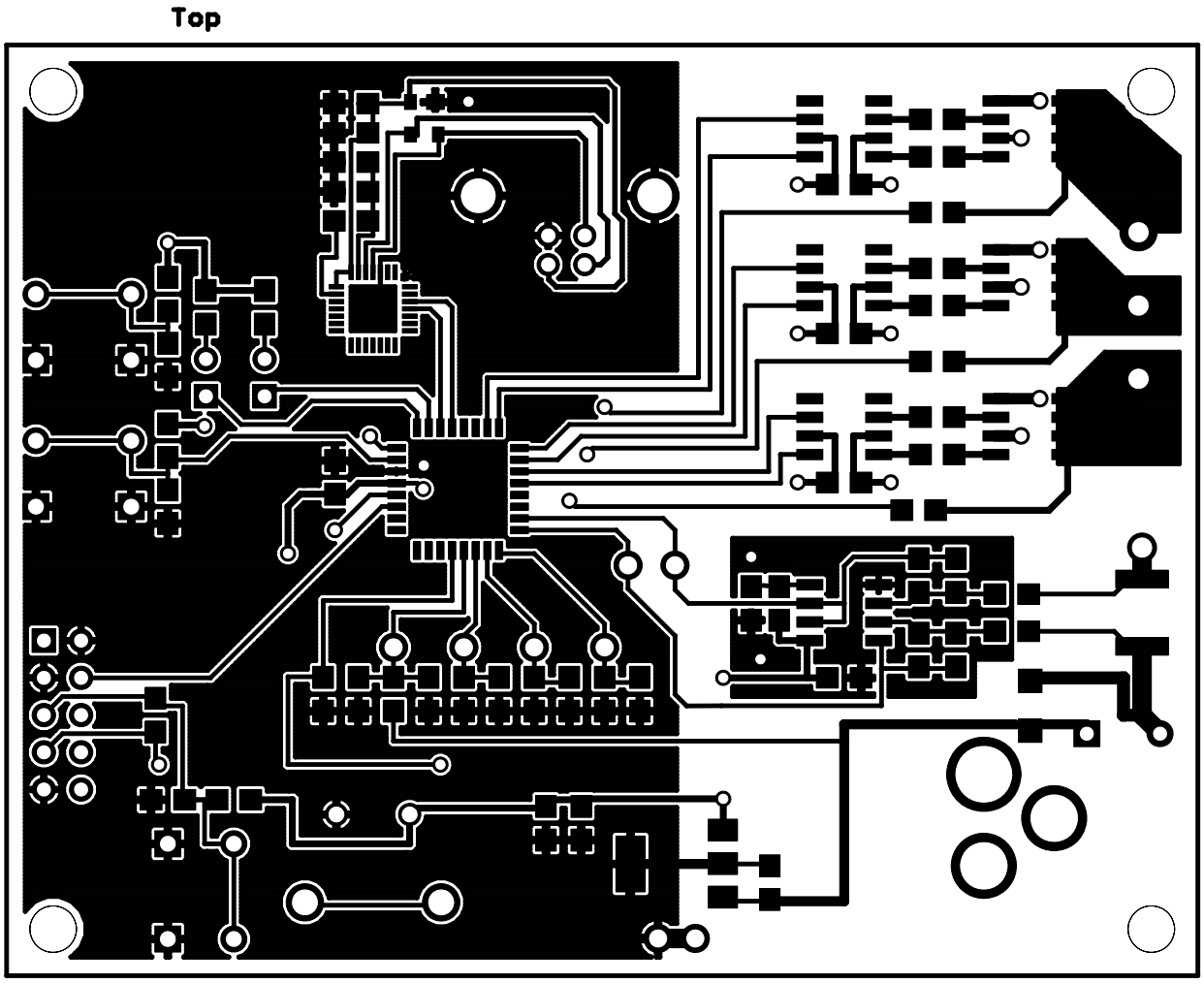


Figure 14. PCB Top Copper Layer

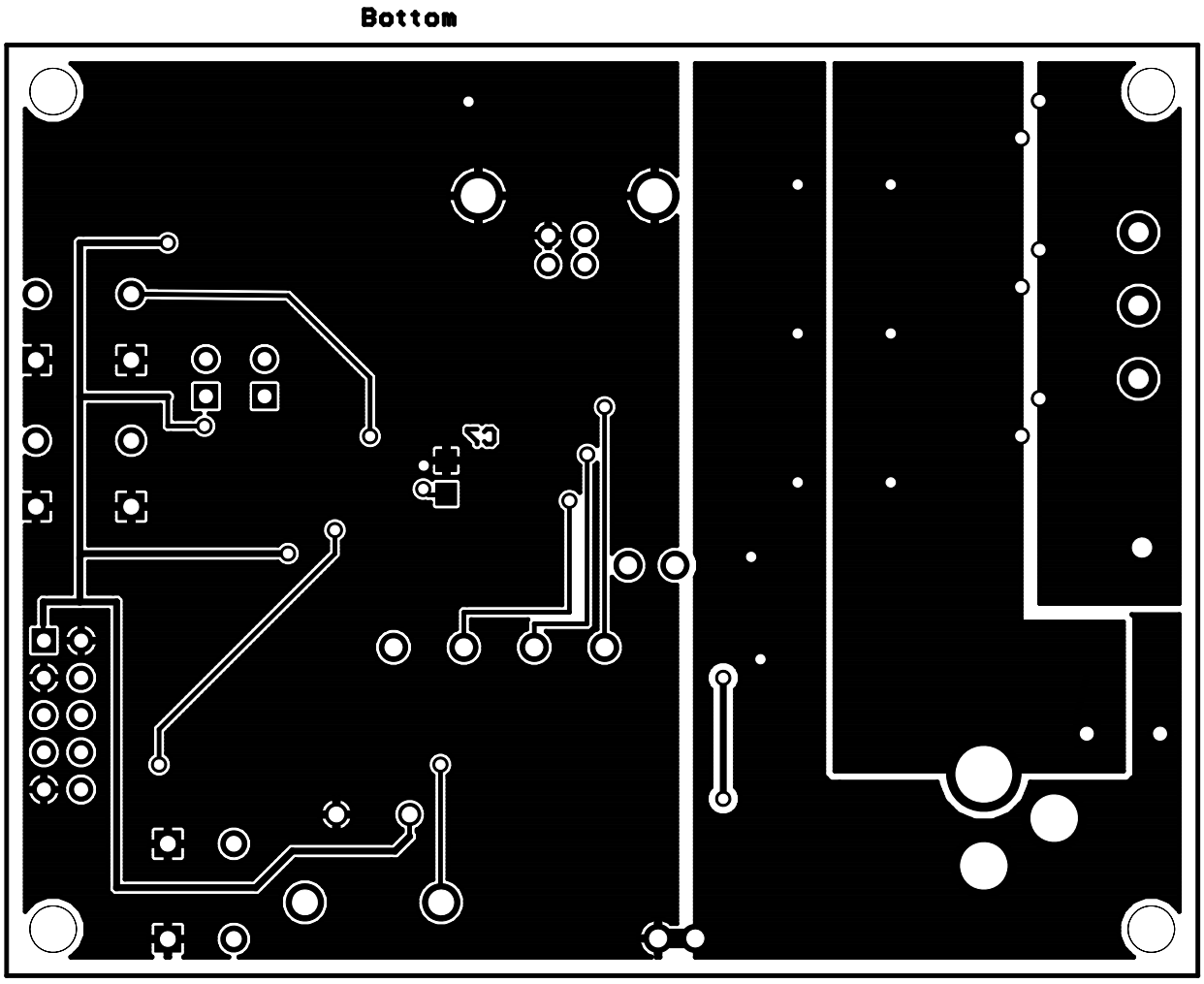


Figure 15. PCB Bottom Copper Layer

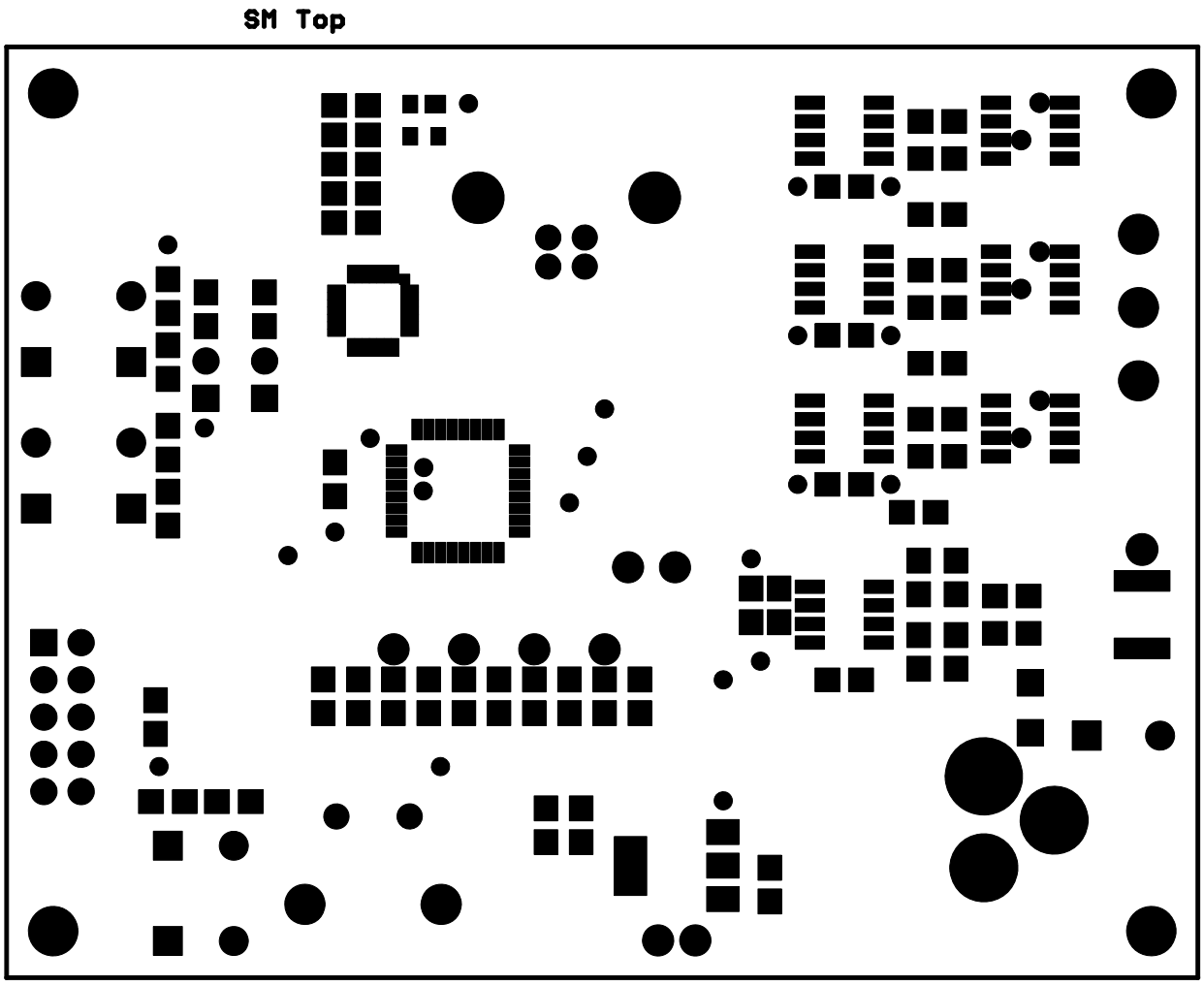


Figure 16. PCB Top Solder Mask Layer

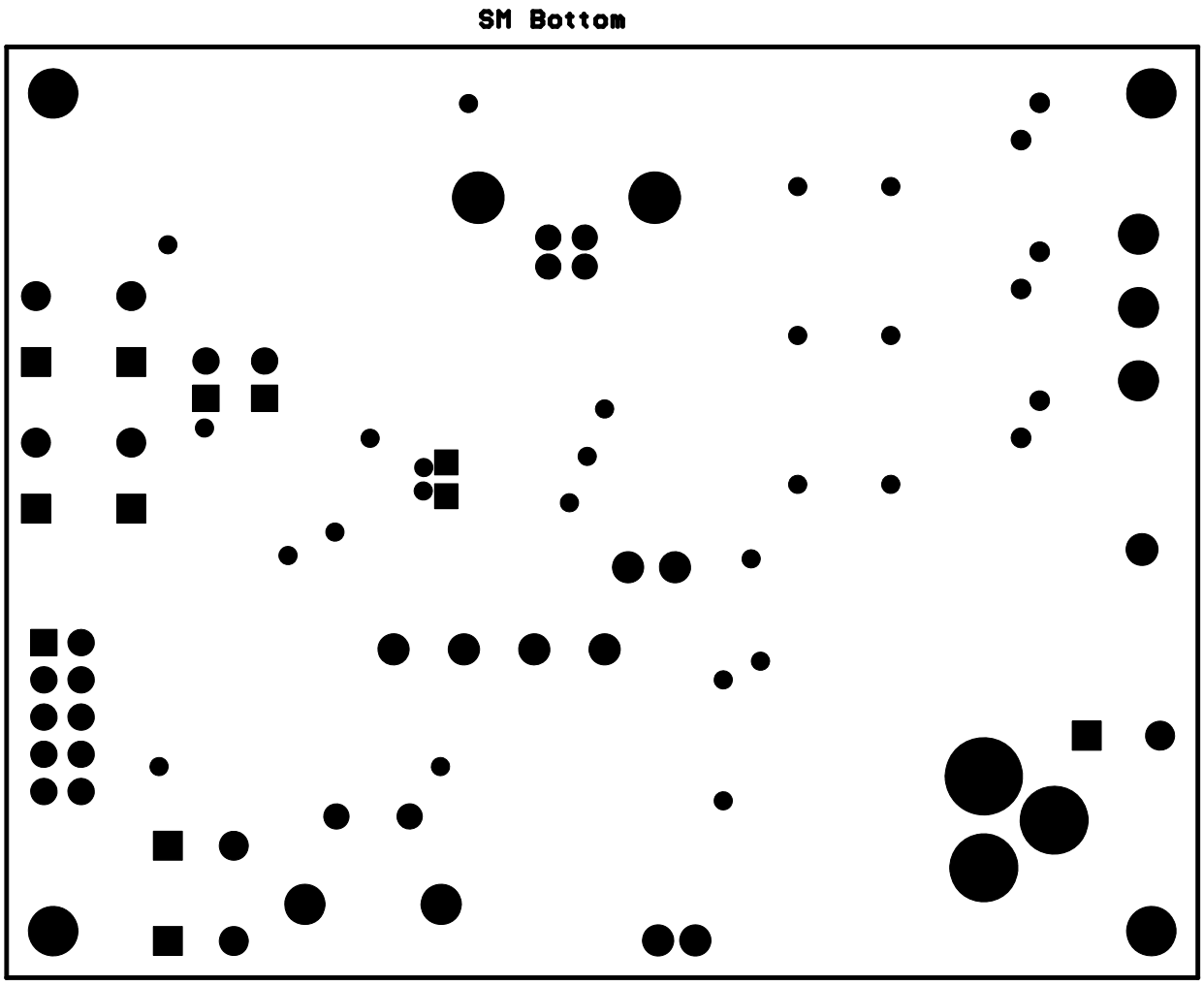


Figure 17. PCB Bottom Solder Mask

APPENDIX D—SENSORLESS BLDC MOTOR CODE

slbdc.h

```
//-----------------------------------------------------------------------------
// Sensorless BLDC Motor Reference Design
//-----------------------------------------------------------------------------
// Copyright 2006 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 14 SEP 2006
//
// This program provides Sensorless BLDC motor control using the
// 'F310. This software is written specifically for the SBLDC
// reference design hardware. Please see the full application
// note for further information.
//
// Target: C8051F30x
//
// Tool chain: KEIL 'c' full-version required
//
// Rev History:
//
// 30 AUG 2006 - Initial release (Pitmann  N2311A011 Motor)
//
// 14 SEP 2006 - Changed Motor Supplier (Anaheim Automation BLY171S-24V-4000)
//               Added switch for different motor types.
//               Modified Motor Parameters in this header file only.
//               Other source files unchanged.
//
//-----------------------------------------------------------------------------
// MACROS
//-----------------------------------------------------------------------------
// User modified Motor Parameters
//
// These parameters are related to thespecific motor used for the reference
// design. These parameters will need to be changed if using a different motor.
//
// define motor type - Pittman, Anaheim Automation, or custom motor
//
//

#define ANAHEIM_BLY171S_24V_4000
//#define PITTMAN_N2311A011

// Motor parameters for Aneheim Automation BLYS171S_24V_4000 Motor
#ifdef ANAHEIM_BLY171S_24V_4000
#define MILLIOHMS 1800                 // motor resistance in milliohms
#define VMOTOR 12                      // motor voltage in volts
#define IMOTOR 500                     // maximum motor current in millamps
#define NUM_POLES 8                    // motor poles
#define RATED_RPM 4000                 // rated no load rpm at max voltage
#define MAX_RPM 2000                   // max desired speed
#define KPINIT 20                      // Kp initial value
#endif

// Motor parameters for Pittman Motor N2311A011 Motor
#ifdef PITTMAN_N2311A011
```

```
#define MILLIOHMS 260                  // motor resistance in milliohms
#define VMOTOR 12                      // motor voltage in volts
#define IMOTOR 5                       // maximum motor current in amps
#define NUM_POLES 4                    // motor poles
#define RATED_RPM 7500                 // rated no load rpm at max voltage
#define MAX_RPM 5000                   // max desired speed
#define KPINIT 40                      // Kp initial value
#endif

//-----------------------------------------------------------------------------
// constants
#define SYSCLK 24500000L               // bus clock (long int)
#define ILIMIT 1000                    // current limit
#define VSTALL 2                       // minimum stall criteria
#define TALIGN 255                     // align time in 10ms increments
#define ON 0                           // push-button logic ON
#define OFF 1                          // push-button logic OFF
#define STOP 0                         // Motor State variable values
#define ALIGN 1
#define START 2
#define RUN 3
//-----------------------------------------------------------------------------
// preprocessor calculated values
#define TABLE_MIN 8
#define TABLE_MAX 240
#define RAMP_MOD 8
#define RUN_MOD 4
#define T0_DIV 48
#define VLIMIT  ((128L * MILLIOHMS * IMOTOR )/VMOTOR /1000000)
#define T0_CLK (SYSCLK/T0_DIV)
#define TSCALE ((2*10*60/6)*T0_CLK/RATED_RPM/NUM_POLES/RAMP_MOD/TABLE_MIN)
#define VMIN (128/10)                  // (128 /10)
#define VSTART (128 * TABLE_MIN / 10)
#define TENMS (SYSCLK / T0_DIV / 100)      // timer count for 10ms delay
#define TRUN_TIMES_127 (((127L*2*60/6)*T0_CLK)/RATED_RPM/NUM_POLES/RUN_MOD)
#define RPM_SCALE (RATED_RPM/127)
#define T2_RELOAD (65536-SYSCLK/1000)
#define T2_RELOAD_L (T2_RELOAD & 0x00FF)
#define T2_RELOAD_H (T2_RELOAD/256)
#define OMEGA_LIMIT (127*MAX_RPM/RATED_RPM)

//-----------------------------------------------------------------------------
// preprocessor error checking
// These macros check for range errors on user modifiable parameters
#if ((TSCALE)>(255))
   #error “TSCALE requires 16 bit math”
#endif
#if ((TSCALE*TABLE_MAX)>65535)
   #error “TSCALE too large”
#endif

//-----------------------------------------------------------------------------
// typedefs
//-----------------------------------------------------------------------------
typedef union                          // unsigned union for accessing SFRs
{
  struct
  {
      unsigned char hi;
```

```
      unsigned char lo;
  } b;
  unsigned int w;
}udblbyte;

typedef union                          // signed union used for signed ADC
{
  struct
  {
      unsigned char hi;
      unsigned char lo;
  } b;
  signed int w;
}sdblbyte;
```

slbdc.c

```
//-----------------------------------------------------------------------------
// Sensorless BLDC Motor Reference Design
//-----------------------------------------------------------------------------
// Copyright 2006 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 30 AUG 2006
//
// This program provides Sensorless BLDC motor control using the
// 'F310. This software is written specifically for the SBLDC
// reference design hardware. Please see the full application
// note for further information.
//
// Target: C8051F30x
//
// Tool chain: KEIL 'c' full-version required
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f310.h>                 // SFR declarations
#include <stdio.h>                     // printf()
#include <slbdc.h>                     // macro constants
//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void SYSCLK_Init (void);
void PORT_Init (void);
void PCA0_Init (void);
void ADC_Init(void);
void UART0_Init (void);
extern void StartMotor(void);          // located in T0_ISR.c
extern void T0_Init(void);             // located in T0_ISR.c
extern void T2_Init(void);             // located in T2_ISR.c
//-----------------------------------------------------------------------------
// External Public Variables
//-----------------------------------------------------------------------------
// T0_ISR Variables accessed by GUI
extern unsigned char Status;           // Motor State Variable
extern unsigned int Imotor;            // Motor current
extern unsigned int Vpot;              // Speed Pot Voltage
extern signed int Vemf;                // back EMF magnitude
extern signed int Verror;              // error voltage from midpoint
extern bit Stall;                      // Stall detection flag
extern bit OverCurrent;                // Over current flag
// T2_ISR Variables accessed by GUI
extern signed int Kp;                  // Proportional Constant
extern signed int Ki;                  // Integral Constant
extern signed int Vpi;                 // output from PI controller
extern unsigned int Vout;              // Output Voltage
extern unsigned int SpeedRPM;          // Motor speed in RPM

//-----------------------------------------------------------------------------
// Global Local Variables
//-----------------------------------------------------------------------------

sbit Start = P0^0;
sbit Stop = P0^1;
```

```
//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
void main (void)
{
   char theChar;

   PCA0MD &= ~0x40;                    // Disable Watchdog Timer

   SYSCLK_Init ();
   PORT_Init();                        // initialize system clock
   PCA0_Init();
   T0_Init ();
   T2_Init ();
   UART0_Init ();                      // initialize UART
   ADC_Init();
   EA = 1;                             // enable global interrupts
   printf("\r\nreset...\r\n");
   while(1)
   {
      printf("push start button...\r\n");
      while(Start==OFF);               // wait for start
      while(Start==ON);                // wait for release
      StartMotor();                    // start motor
      printf("aligning...\r\n");
      while(Status!=START);            //wait for run
      printf("starting...\r\n");
      while(Status!=RUN);              //wait for run
      printf("running...\r\n");
      while(Status==RUN)
      {
         if (RI0)                      // check for input (non-blocking)
         {
            theChar = getkey();
            while (theChar<'a' || theChar>'z')
               theChar = getkey();     // ignore non-alpha
            switch(theChar)            // parse theChar
            {
               case 'p':               // set Kp
                  printf("Kp?\r\n");
                  scanf("%d",&Kp);
                  printf("\r\nKp=%d\r\n",Kp);
                  break;
               case 'i':               // set Ki
                  printf("Ki?\r\n");
                  scanf("%d",&Ki);
                  printf("\r\nKi=%d\r\n",Ki);
                  break;
               case 's':               // display Status
                  printf("Speed=%u\r\n",SpeedRPM);
                  printf("Vout=%u\r\n",Vout);
                  printf("Verror=%d\r\n",Verror);
                  printf("Imotor=%u\r\n",Imotor);
                  printf("Vemf=%d\r\n",Vemf);
                  printf("Vpi=%d\r\n",Vpi);
                  printf("Vpot=%u\r\n",Vpot);
                  printf("Kp=%d\r\n",Kp);
                  printf("Ki=%d\r\n",Ki);
                  printf("\r\n");
```

```
                  break;
               default:
                  printf("Error\r\n");
            } //end switch
         } //end if
         if(Stall)                     // check stall flag
         {
            printf("Stall!\r\n");
         }
         if(Stop==ON)                  // check Stop button
         {
            Status = STOP;
            printf("Stopping Motor!\r\n");
            while(Stop==ON);           // wait for release
         }
      }
      if(OverCurrent)               // check over current flag
      {
         printf("OverCurrent!\r\n");
      }

   }
}

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------

void SYSCLK_Init (void)
{
   OSCICN = 0x83;                      // configure for 24.5 MHz
}
//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------

void PORT_Init (void)
{
   // P0.0 = Run,       Skipped, Open-Drain Output/Input
   // P0.1 = Reverse    Skipped, Open-Drain Output/Input
   // P0.2 = LED1       Skipped, Push-Pull Output
   // P0.3 = LED1       Skipped, Push-Pull Output
   // P0.4 = Txd        UART, Push-Pull Output
   // P0.5 = Rxd        UART, Open-Drain Output/Input
   // P0.6 = NC         Skipped, Open-Drain Output/Input
   // P0.7 = NC         Skipped, Open-Drain Output/Input
   // P1.0 = Abottom    PCA, Push-Pull Output
   // P1.1 = Atop       PCA, Push-Pull Output
   // P1.2 = Bbottom    PCA, Push-Pull Output
   // P1.3 = Btop       PCA, Push-Pull Output
   // P1.4 = Cbotton    PCA, Push-Pull Output
   // P1.5 = Ctop       PCA, Push-Pull Output
   // P1.6 = NC         Skipped, Open-Drain Output/Input
   // P1.7 = NC         Skipped, Open-Drain Output/Input
   // Port 2
   // P2.0 = VI         Skipped, Analog Input
   // P2.1 = VO         Skipped, Analog Input
   // P2.2 = VA         Skipped, Analog Input
   // P2.3 = VB         Skipped, Analog Input
   // P2.4 = VC         Skipped, Analog Input
```

```
   // P2.5 = VM         Skipped, Analog Input
   // P2.6 = Pot        Skipped, Analog Input
   // P2.7 = NC         Skipped, Analog Input


   XBR0 = 0x01;                        // Enable UART on Crossbar
   XBR1 = 0x02;                        // Enable CEX0,CEX1 on Crossbar
   P0MDOUT = 0x1C;                     // P0.2, P0.3, & P0.4 are outputs
   P1MDOUT = 077;                      // enable motor outputs (octal)
   P2MDIN = 0x00;                      // all P2 pins are Analog inputs
   P0SKIP = ~0x30;                     // Skip all, except UART
   P1SKIP = 071;                       // initial PSKIP pattern (octal)
   P2SKIP = 0x0F;                      // Skip all P2 Pins
   XBR1 |= 0x40;                       // enable crossbar
   P1    = 0xff;                       // P1 all high
}
//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------
void PCA0_Init (void)
{

   PCA0MD = 0x02;                      // PCA uses sysclk/4, no CF int
   PCA0CPL0 = 0x00;                    // clear mode, pin high
   PCA0CPL1 = 0x00;                    // clear mode, pin high
   PCA0L = 0x00;                       // reset the timer
   PCA0H = 0x00;                       // reset the timer
   PCA0CPH0 = 0x00;                    // init to 0%
   PCA0CPH1 = 0x00;                    // init to 0%
   CR = 1;                             // START PCA0 timer
}
//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------
void ADC_Init()
{
   AMX0P = 0x09;                       // config for motor current
   AMX0N = 0xFF;                       // single ended
   ADC0CF = 0x40;                      // SARCLK 272222,
   ADC0CN = 0x80;                      // initiate on AD0BUSY
   REF0CN    = 0x08;                   // use vdd for reference
}
//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------

void UART0_Init (void)
{
   SCON0 = 0x10;                       // enable receiver
   TMOD  &= ~0x30;                     // clear T1 mode
   TMOD |= 0x20;                       // T1 mode 2
   CKCON |= 0x08;                      // T1 uses SYSCLK
   TH1 = 0x96;                         // fixed 115200 baud
   TL1 = TH1;                          // init Timer1
   TR1 = 1;                            // START Timer1
   TI0 = 1;                            // Indicate TX0 ready
}
```

```
T0_ISR.c
//-----------------------------------------------------------------------------
// Sensorless BLDC Motor Reference Design
//-----------------------------------------------------------------------------
// Copyright 2006 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 30 AUG 2006
//
// This program provides Sensorless BLDC motor control using the
// 'F310. This software is written specifically for the SBLDC
// reference design hardware. Please see the full application
// note for further information.
//
// Target: C8051F30x
//
// Tool chain: KEIL 'c' full-version required
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f310.h>                 // SFR declarations
#include <slbdc.h>                     // macro constants

//-----------------------------------------------------------------------------
// commutation patterns
//-----------------------------------------------------------------------------

const unsigned char code skipPattern[6]=
    {071,055,047,066,036,033};         // code in octal

const unsigned char code openPhase[6]=
    {0x0C,0x0B,0x0A,0x0C,0x0B,0x0A};   // open phase mux value


const unsigned char code TRamp[192]=   // linear acceleration table
{
   0xF0, 0x63, 0x4C, 0x40, 0x38, 0x33, 0x2F, 0x2B,
   0x29, 0x26, 0x25, 0x23, 0x21, 0x20, 0x1F, 0x1E,
   0x1D, 0x1C, 0x1B, 0x1B, 0x1A, 0x19, 0x19, 0x18,
   0x18, 0x17, 0x17, 0x16, 0x16, 0x16, 0x15, 0x15,
   0x15, 0x14, 0x14, 0x14, 0x13, 0x13, 0x13, 0x13,
   0x12, 0x12, 0x12, 0x12, 0x11, 0x11, 0x11, 0x11,
   0x11, 0x11, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
   0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F,
   0x0E, 0x0E, 0x0E, 0x0E, 0x0E, 0x0E, 0x0E, 0x0E,
   0x0E, 0x0D, 0x0D, 0x0D, 0x0D, 0x0D, 0x0D, 0x0D,
   0x0D, 0x0D, 0x0D, 0x0D, 0x0D, 0x0C, 0x0C, 0x0C,
   0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C,
   0x0C, 0x0C, 0x0C, 0x0C, 0x0B, 0x0B, 0x0B, 0x0B,
   0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B,
   0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0A,
   0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A,
   0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A,
   0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A,
   0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
   0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
   0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
```

```
   0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
   0x09, 0x09, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08
};


//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void StartMotor(void);
void T0_Init(void);
void T0_ISR (void);
void stop(void);
void align(void);
void start(void);
void start0(void);
void start7(void);
void run(void);
void run0(void);
void run1(void);
void run2(void);
void run3(void);
unsigned int avgCurrent(void);
signed int avgVoltage(void);
signed int avgVmid(void);
signed int avgVdelta(void);
signed int flip(signed int);
void updateT(void);
void commutate();
void error(void);


//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------

// Public accessed by GUI
unsigned char Status;                  // motor state variable
unsigned int Imotor;                   // motor current measurement
unsigned int Vpot;                     // speed pot voltage measurement
signed int Vemf;
signed int Verror;                     // error voltage from min-point voltage
bit Stall;                             // stall flag bit
bit OverCurrent;                       // over-current flag bit

// Local global T0_ISR variables
unsigned char Tslot;
unsigned int NextT;
unsigned char AccIndex;
unsigned char MtrIndex;
signed int idata Vmid[6];              // used for avg Verror calc
signed int idata Vdelta[6];            // used for average Vdelta calc
signed int Vmin;
signed int Vmax;
bit Flip;
sbit AlignLED = P0^2;
sbit BlinkLED = P0^3;

// external T2_ISR variables
```

```
extern unsigned char Speed;

//-----------------------------------------------------------------------------
// StartMotor function - called from main
//-----------------------------------------------------------------------------

void StartMotor(void)
{
   TL0 = 0xF0;                          // T0 overflow in 16 ticks
   TH0 = 0xFF;                          // msb set to FF
   ET0 = 1;                             // enable T0 interrupt
   TR0 = 1;                             // start Timer0
   MtrIndex = 5;                        // MtrIndex will flip to 0
   Status = ALIGN;                      // set Status to ALIGN state
   AlignLED = ON;                       // turn on LED
   Tslot = 0;                           // commutate on next ISR
   Speed = 0;                           // reset speed ramp controller
}

//-----------------------------------------------------------------------------
// T0_Init - called from main
//-----------------------------------------------------------------------------

void T0_Init(void)
{
   TMOD    &= ~0x03;                    // clear T0 mode
   TMOD    |= 0x01;                     // T0 mode 1
   CKCON   &= ~0x07;                    // clear SCAx and T0M
   CKCON   |= 0x02;                     // T0 uses SYSCLK/48
   IP      |= 0x02;                     // T0 high priority
   CKCON   |= 0x10;                     // T2 uses SYSCLK
   TMR2RLL   = T2_RELOAD_L;             // Timer 2 Low Byte
   TMR2RLH   = T2_RELOAD_H;             // Timer 2 High Byte
}
//-----------------------------------------------------------------------------
// T0_ISR
//-----------------------------------------------------------------------------

void T0_ISR (void) interrupt 1

{
   if (Tslot==0) commutate();           // commutate first if time slot zero

   switch(Status)                       // implement state diagram
   {
      case STOP:
         stop();                        // stop motor
         break;

      case ALIGN:
         align();                       // align motor
         break;

     case START:
         start();                       // start motor
         break;

      case RUN:
         run();                         // run
```

```
         break;
   }
   updateT();                          // update T before returning
}

//-----------------------------------------------------------------------------
// stop() called from T0_ISR()
//-----------------------------------------------------------------------------

void stop (void)
{
      PCA0CPH0 = 0;                    // reset duty cycle
      PCA0CPH1 = 0;                    // reset duty cycle
      PCA0CPM0 = 0x00;                 // disable PWM
      PCA0CPM1 = 0x00;                 // disable PWM
      ET0 = 0;                         // disable T0 interrupt
      ET2 = 0;                         // disable T2 interrupt
      TR0 = 0;                         // stop Timer 0
       TMR2CN &= ~0x04;                 // stop Timer 2

}

//-----------------------------------------------------------------------------
// align() called from T0_ISR()
//-----------------------------------------------------------------------------

void align(void)
{
   static unsigned char v=0;
   static unsigned char d=0;
   Tslot = 1;                          // don't commutate
   NextT = TENMS;                      // execute align every 10ms
   if(v < (128 + VLIMIT))              // ramp voltage to 50% + VLIMIT
   {
      v++;                             // increment v
      PCA0CPH0 = v;                    // update PWM
      PCA0CPH1 = v;                    // update PWM
   }
   else if (d < TALIGN)                // align delay
   {
      d++;                             // increment d
   }
   else
   {
      Status = START;                  // go to next state
      AccIndex = 0;                    // reset table index
      Tslot = 0;                       // commutate on next interrupt
      AlignLED = OFF;                  // turn off LED
      v = 0;                           // clear v for next restart
      d = 0;                           // clear d for next restart
   }

}
//-----------------------------------------------------------------------------
// start() called from T0_ISR()
//-----------------------------------------------------------------------------
void  start(void)
{
      switch(Tslot)                    // implement time slot manager
```

```
      {
         case 0:
            start0();                  // do start0() one of eight times
            break;
         case 7:
            start7();                  // check if done
            break;
      }
      Tslot++;                         // increment time slot
      Tslot &= 0x07;                   // mask to 3 bits (modulus 8)
}
//-----------------------------------------------------------------------------
// start0()
//-----------------------------------------------------------------------------
void start0(void)
{
   unsigned int t,v;
   unsigned char d;

   t = TRamp[AccIndex];              // look-up table value
   v = VSTART / t;                   // divide to get v
   t *= TSCALE;                      // scale to get t
   v += 128;                         // add 50%
   v += VLIMIT;                      // add VLIMIT
   if(v > 255)                       // limit to unsigned char
   {
      v = 255;
   }
   d = v;                            // copy to unsigned char d
   PCA0CPH0 = d;                     // update  PWM
   PCA0CPH1 = d;                     // update PWM
   AccIndex++;                       // increment index
   NextT = t;                        // update next T
}

//-----------------------------------------------------------------------------
// start7()
//-----------------------------------------------------------------------------
void start7(void)
{
   if (AccIndex > 191)               // check if done accelerating
   {
      Status = RUN;                  // go to next state
      NextT = 16 * TSCALE;           // update next T
      AMX0P = 0x09;                  // config for motor current
      AMX0N = 0xFF;                  // single ended
      ADC0CN = 0x81;                 // initiate on T0 overflow
      TMR2CN|= 0x04;                 // enabe Timer 2
      ET2 = 1;                       // Enable T2 interrupt
   }
}
//-----------------------------------------------------------------------------
// run()
//-----------------------------------------------------------------------------

void run(void)
{
   Tslot &= 0x03;                    // mask Tslot to 2 bits (modulo 4)
   switch(Tslot)                     // implement time slot manager
```

```
   {
      case 0:
         run0();                        // run time slot 0
         break;
      case 1:
         run1();                        // run time slot 1
         break;
      case 2:
         run2();                        // run time slot 2
         break;
      case 3:
         run3();                        // run time slot 3
         break;
   }
   Tslot++;                             // increment time slot
   Tslot &= 0x03;                       // mask Tslot to 2 bits (modulo 4)
}
//-----------------------------------------------------------------------------
// run0()
//-----------------------------------------------------------------------------
void run0(void)
{
   Imotor = avgCurrent();               // avg motor current using ADC
   if (Imotor > ILIMIT)                 //
   {
      OverCurrent = 1;                  //
      AlignLED = ON;                    //
      Status = STOP;
   }
  AMX0P = 0x0E;                         // config for pot
  AMX0N = 0xFF;                         // single ended
  ADC0CN = 0xC0;                        // initiate on AD0BUSY, LPT
  ADC0CF |= 0x04;                       // left justify
  AD0INT = 0;                           // clear ADC0 end-of-conversion
  AD0BUSY = 1;                          // initiate conversion
  while (!AD0INT);                      // wait for conversion to complete
  Vpot = ADC0H;                         // read speed pot
  AMX0P = openPhase[MtrIndex];          // config for open phase
  AMX0N = 0x0D;                         // differential measurement
  ADC0CF &= ~0x04;                      // right justify
  ADC0CN = 0x81;                        // initiate on T0 overflow
}
//-----------------------------------------------------------------------------
// run1()
//-----------------------------------------------------------------------------
void run1(void)
{
   signed int v;
   v = avgVoltage();                    // read back EMF
   v = flip(v);                         // flip every other cycle
   Vmin = v;                            // save min voltage
   ADC0CN = 0x81;                       // initiate on T0 overflow
}
//-----------------------------------------------------------------------------
// run2()
//-----------------------------------------------------------------------------
void run2(void)
{
   signed int v;
```

```
   Vmid[MtrIndex] = avgVoltage();      // read avg voltage and store in array
   v = avgVmid() - Vmid[MtrIndex];     // subtract from avg midpoint
   v = flip(v);                        // flip every other cycle
   Verror = v;                         // store in global Verror
   ADC0CN = 0x81;                      // initiate on T0 overflow
}

//-----------------------------------------------------------------------------
// run3()
//-----------------------------------------------------------------------------
void run3(void)
{
   static unsigned char stable = 12;
   signed int v;
   v = avgVoltage();                   // read avg voltage
   v = flip(v);                        // flip every other cycle
   Vmax = v;                           // store max voltage
   Vdelta[MtrIndex]= Vmax - Vmin;      // calculate delta
   if(stable==0)                       // stabilization delay
   {
      Vemf = avgVdelta();              // calculate back EMF magnitude
      if(Vemf<VSTALL)                  // check for stall
      {
         Stall = 1;                    // set stall flag
         AlignLED = ON;                // blink LED
      }
      else
      {
         Stall = 0;                    // clear stall flag
         AlignLED = OFF;               // turn off LED
      }

   }
   else
   {
      stable--;                        // decrement stable delay
   }
   AMX0P = 0x09;                       // config for motor current
   AMX0N = 0xFF;                       // single ended
   ADC0CN = 0x81;                      // initiate on T0 overflow
}
//-----------------------------------------------------------------------------
// run4()
//-----------------------------------------------------------------------------
unsigned int avgCurrent(void)
{
   unsigned int sum, result;
   udblbyte v;

   unsigned char i ;

   sum = 0;
   while (!AD0INT);                    // wait for conversion to complete
   v.b.lo = ADC0L;                     // read ADC
   v.b.hi = ADC0H;
   sum += v.w;
   ADC0CN = 0x80;                      // initiate on AD0BUSY
   for (i = 7; i != 0; i--)            // repeat 7 more times
   {
```

```
      AD0INT = 0;                       // clear ADC0 end-of-conversion
      AD0BUSY = 1;                      // initiate conversion
      while (!AD0INT);                  // wait for conversion to complete
      v.b.lo = ADC0L;                   // read ADC
      v.b.hi = ADC0H;
      sum += v.w;                       // add to sum
   }
   result = sum>>2;
   return result;                       // return average reading
}

//-----------------------------------------------------------------------------
// flip()
//-----------------------------------------------------------------------------
signed int flip (signed int v)
{
   if ((MtrIndex & 0x01) == 0x00)       // flip on 0, 2, and 4
   {
      v = -v;
   }
   return v;
}
//-----------------------------------------------------------------------------
// avgVoltage()
//-----------------------------------------------------------------------------

signed int avgVoltage(void)
{
   signed int sum, result;
   sdblbyte v;

   unsigned char i ;

   sum = 0;
   while (!AD0INT);                     // wait for conversion to complete
   v.b.lo = ADC0L;                      // read ADC
   v.b.hi = ADC0H;
   sum += v.w;                          // add to sum
   ADC0CN = 0x80;                       // initiate on AD0BUSY
   for (i = 7; i != 0; i--)             // repeat 7 more times
   {
      AD0INT = 0;                       // clear ADC0 end-of-conversion
      AD0BUSY = 1;                      // initiate conversion
      while (!AD0INT);                  // wait for conversion to complete
      v.b.lo = ADC0L;                   // read ADC
      v.b.hi = ADC0H;
      sum += v.w;
   }
   result = sum>>2;                     // divide by 4,  11-bit effective
   return result;                       // return average reading
}
//-----------------------------------------------------------------------------
// avgVmid()
//-----------------------------------------------------------------------------
signed int avgVmid(void)
{
   signed int sum;
   unsigned char i;
   sum = 0;
```

```
   for (i = 0; i < 6; i++)            // repeat 6  times
   {
         sum += Vmid[i];              // calculate sum of midpoints
   }
   sum /=6;                           // divide by 6
   return sum;                        // return average
}
//-----------------------------------------------------------------------------
// avgVdelta()
//-----------------------------------------------------------------------------

 signed int avgVdelta(void)
 {
    signed int sum;
    unsigned char i;
    sum = 0;
    for (i = 0; i < 6; i++)           // repeat 6  times
    {
       sum += Vdelta[i];              // calculate sum of Vdelta
    }
    sum /=6;                          // divide by 6
    return sum;
}

//-----------------------------------------------------------------------------
// updateT()
//-----------------------------------------------------------------------------
void updateT (void)
{
   udblbyte t0;
   t0.b.lo = TL0;                     // get current value
   t0.b.hi = TH0;
   t0.w -= NextT;                     // subtract period
   TL0 = t0.b.lo;                     // save new overflow value
   TH0 = t0.b.hi;
}
//-----------------------------------------------------------------------------
// commutate()
//-----------------------------------------------------------------------------
void commutate (void)
{
   MtrIndex++;                        // increment MtrIndex
   if(MtrIndex>5)                     // fix if greater than 5
   {
      MtrIndex = 0;
   }
   P1       = 0xff;                   // P1 all high
   PCA0CPM0 = 0x00;                   // disable PWM
   PCA0CPM1 = 0x00;                   // disable PWM
   XBR1 &= ~0x40;                     // disable crossbar
   P1SKIP = skipPattern[MtrIndex];
   XBR1 |= 0x40;                      // enable crossbar
   PCA0CPM0 = 0x42;                   // enable 8-bit PWM mode
   PCA0CPM1 = 0x42;                   // enable 8-bit PWM mode


   if(MtrIndex==0)                    // toggle LED on zero
   {
      BlinkLED = ON;
```

```
   }
   else
   {
      BlinkLED = OFF;
   }

}
```

T2_ISR.c

```
//-----------------------------------------------------------------------------
// Sensorless BLDC Motor Reference Design
//-----------------------------------------------------------------------------
// Copyright 2006 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 30 AUG 2006
//
// This program provides Sensorless BLDC motor control using the
// 'F310. This software is written specifically for the SBLDC
// reference design hardware. Please see the full application
// note for further information.
//
// Target: C8051F30x
//
// Tool chain: KEIL 'c' full-version required
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f310.h>                  // SFR declarations
#include <slbdc.h>                      // macro constants
//-----------------------------------------------------------------------------
// External Public Variables
//-----------------------------------------------------------------------------
extern unsigned int Vpot;               // pot reading from T0_ISR.c
extern signed int Verror;               // error signal from T0_ISR.c
extern unsigned int NextT;              // next period to T0_ISR.c
//-----------------------------------------------------------------------------
// Public Variables
//-----------------------------------------------------------------------------
signed int Ki=0;                        // integral constant (initially zero)
signed int Kp=KPINIT;                   // proportional constant
unsigned int SpeedRPM;                  // speed in RPM for GUI
signed int Vpi;                         // output from PI controller
signed int Vout;                        // output voltage
unsigned char Speed;                    // speed for ramp controller

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void T2_Init(void);
void T2_ISR (void);
void PI (void) ;
saturate (signed int,signed int);

//-----------------------------------------------------------------------------
// T2_Init - called from main
//-----------------------------------------------------------------------------

void T2_Init(void)
{
   CKCON   |= 0x10;                    // T2 uses SYSCLK
   TMR2RLL   = T2_RELOAD_L;             // Timer 2 Low Byte
```

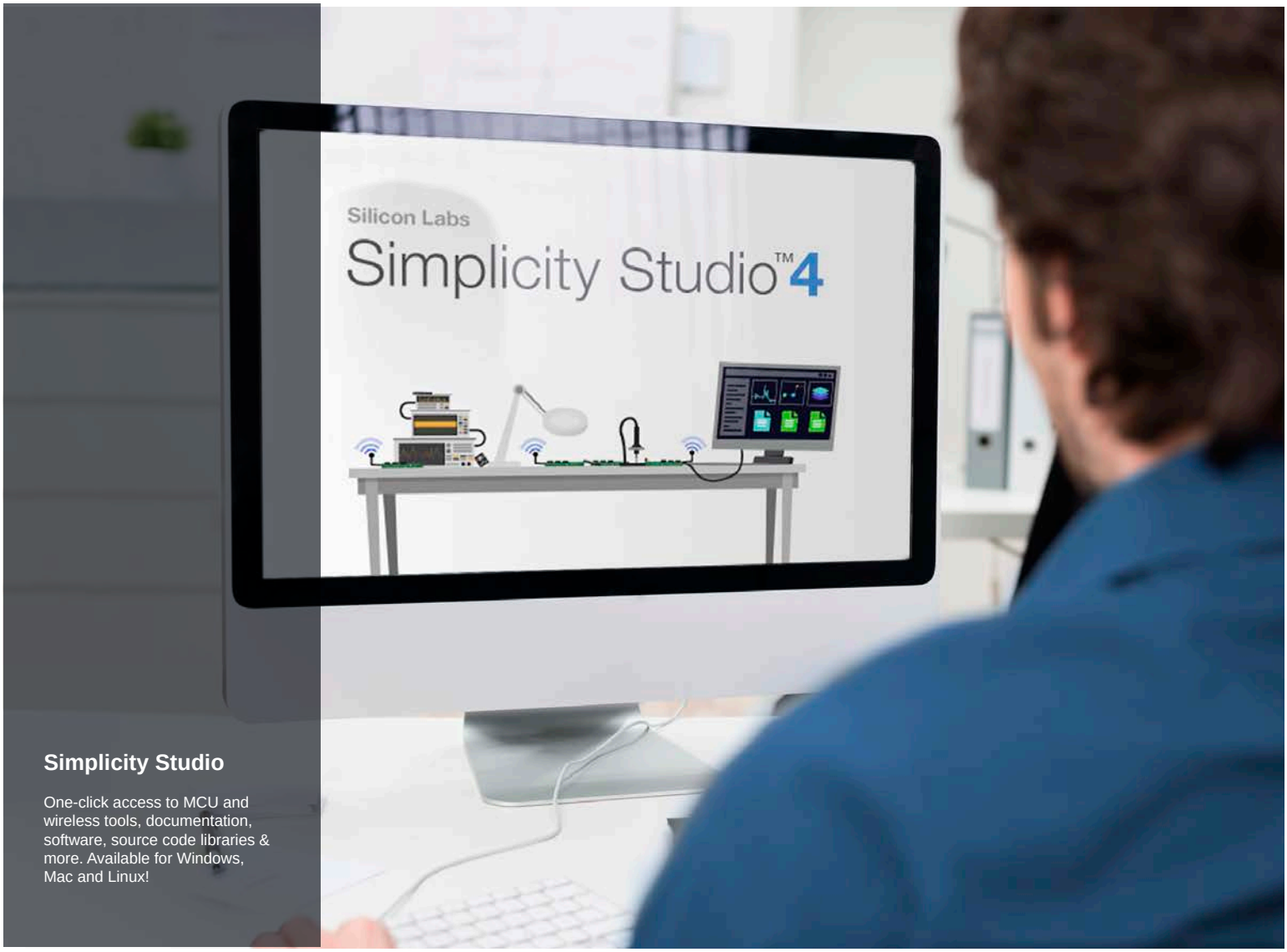
```
   TMR2RLH   = T2_RELOAD_H;            // Timer 2 High Byte
}

//-----------------------------------------------------------------------------
// T2_ISR()
//-----------------------------------------------------------------------------

void T2_ISR (void) interrupt 5
{
   static unsigned int ramp = 1024;    // sets speed ramp
   signed int omega;                   // used for omega calculation
   unsigned char v;                    // used for voltage calculation

   if(ramp==0)                         // do 1 of 1024 times
   {
      if(Speed < (Vpot>>1))
      {
         Speed++;                      // bump up
      }
      else if (Speed > (Vpot>>1))
      {
         Speed--;                      // bump down
      }
      ramp = 1024;                     // reload ramp value
   }
   else
   {
      ramp--;                          // decrement ramp counter
   }

   omega = (Speed>>1) + VMIN;          // convert 0-255 to omega

   Vout = omega;                       // copy to Vout

   PI();                               // do PI

   omega -= (Vpi>>5);                  // subtract error


   if(omega<VLIMIT)                    // fix if under limit
   {
      omega=VLIMIT;
   }
   else if(omega > (128-VLIMIT))       // fix if over limit
   {
      omega = (128-VLIMIT);
   }


   if(Vpi > 128)                       // fix if over limit
   {
      Vout = omega + VLIMIT;
   }
   else if (Vpi <-128)                 // ix if under limit
   {
      Vout = omega - VLIMIT;
   }

   if(Vout< (VMIN+VLIMIT))             // fix if under limit
   {
```

```
      Vout=(VMIN+VLIMIT);
   }
   else if(Vout > 127)                // fix if over limit
   {
      Vout = 127;
   }

   SpeedRPM = omega * RPM_SCALE;             // calc RPM for GUI

   NextT = TRUN_TIMES_127/(unsigned char)(omega);               // calc next period


   v = Vout + 128;                    //update output voltage
   PCA0CPH0 = v;
   PCA0CPH1 = v;

}

//-----------------------------------------------------------------------------
// PI()
//-----------------------------------------------------------------------------
void PI(void)
{
   static signed int i = 0;
   signed int p,e;
   {
      e = 0;                          // clear error
      if(Kp!=0)                       // check for zero
      {
         // pre-saturate p term to 14 bits
         p = saturate (Verror, (8192/Kp));
         e += p * Kp;                 // multiply p term
      }
      if(Ki!=0)                       //check for zero
      {
         i += Verror;                 // integrate error
         i = saturate (i, (8192/Ki)); // pre-saturate i term to 14 bits
         e += i * Ki;                 // multiply i term and add (15 bits)
      }


      Vpi = e>>2;                     // shift output
   }
}

//-----------------------------------------------------------------------------
// saturate()
//-----------------------------------------------------------------------------
signed int saturate (signed int i, signed int l)
{
     if(i > l)                        // if greater than upper limit
     {
        i = +l;                       // set to upper limit
     }
     else if (i < -l)                 // if less than lower limit
     {
        i = -l;                       // set to lower limit
     }
     return i;                        // return saturated value
}
```

DOCUMENT CHANGE LIST

Revision 0.1 to Revision 0.2

- Updated code for slbdc.h.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com