Motor Control Software Examples

1. Introduction

Small motors, less than 300 W, are found in a wide variety of applications. These include automobiles, printers, copiers, paper handlers, factory automation, test equipment, robotics, space & military, and many others. The most popular small motors types are DC, Brushless DC, and Stepper motors. The quantity of motors produced is roughly inversely proportional to the power level. Small motors are produced in much higher quantities than larger motors.

Motor-control-specific DSPs are designed primarily to address the requirements of large off-line motors. Off-line motors are typically AC induction or Brushless DC motors operating from 110 to 480 VAC and ranging from 1/4 to 100 HP. Motor-control-specific DSPs are often too costly for small motors control systems.

The C8051F3xx series of small form-factor microcontrollers is well suited for the control of small motors. These microcontrollers have several features that are very useful in motor control systems. In addition to the standard 8051 timers, the C8051F3xx series also features a programmable counter array (PCA). The PCA has several operating modes. The 8-bit PWM mode is ideally suited for most small motor control applications. The high-speed output mode can be used to generate multiple center-aligned PWM signals with dead-time. The digital crossbar on the C8051F3xx family can be used to select which port pin receives the PWM signal. The crossbar can also be used to eliminate the need for an external multiplexer.

The ADC of the C8051F3xx MCUs can be used to measure the motor current, supply voltage, back-emf, and temperature of the motor. The analog multiplexer and differential measurement capability are very useful in measuring bidirectional motor currents and differential phase voltages. The differential mode and the programmable window detector are used in "Example 3—DC Motor with Soft Reversing," on page 6. The ADC is used to measure the position of a speed control potentiometer in most of the software examples.

The standard 8051 timers T0 and T1 provide a useful second time-base in addition to the PCA. The T0 and T1 timers can be used to control commutation or update rate while the PCA is used for PWM.

The analog comparators can be used to measure zero crossing, over-current, over-voltage, or over-temperature. These types of measurements and protection features are prevalent in motor drive systems.

The C8051F3xx family of microcontrollers feature a two-wire C2 interface that permits Flash programming and debugging. The On-Chip debug circuitry facilitates full speed, non-intrusive in-system debugging. When the MCU hits a breakpoint, the pins are effectively frozen in time. While this behavior is generally desirable, it can leave the PWM outputs in the active state. The suggested debug procedure is to always disconnect the motor leads when single stepping code or using breakpoints.

The purpose of this application note is to provide software examples using the ‘F3xx MCUs to control various types of motors. All examples are relatively simple, but demonstrate effective solutions for the various motor types. A typical motor control system requires additional features and higher complexity. These software examples may be used as a starting point for the development of more complex motor drive systems.

The code listings for the software examples are found in the appendices starting on page 17. The source code may also be downloaded from the Silicon Laboratories web site www.silabs.com.

The code accompanying this application note was originally written for C8051F30x devices. The code can also be ported to other devices in the Silicon Labs microcontroller range.

2. Example 1—DC Motor

DC motors are the most common and least expensive of all the small motors. There are several varieties of DC motors. In this application note the term “DC Motor” refers more specifically to a brush-commutated permanent-magnet DC motor. DC motors are used in a wide range of applications in the automotive, consumer and industrial market segments. Brushless DC (BLDC) motors promise improved reliability, reduced noise, and potentially lower cost. However, BLDC motors have only supplanted conventional DC motors in a few specialized high volume applications—disk drives and computer fans.

The characteristics of a DC motor make it the easiest motor to use in a variable-speed system. The torque speed characteristics of a DC motor are shown in Figure 1. The no-load speed of a DC motor is proportional to the voltage applied across the motor. The voltage-speed characteristics of a DC motor driving a constant-torque load, linear-load or exponential-load are also continuous, positive-slope, and predictable. Thus, in most cases it is feasible to use open-loop control. By simply varying the voltage across the motor, one can control the speed of the motor. Pulse width modulation (PWM) can be used to vary the voltage applied to the motor. The average voltage applied to the motor is proportional to the PWM duty cycle (ignoring the second order effects of the motor inductance and discontinuous operation).

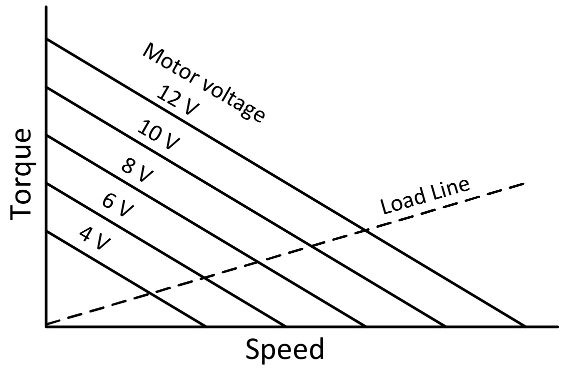


Figure 1. DC Motor Characteristics

Example 1 provides simple speed control of a DC motor using the ‘F300. This example reads the position of a potentiometer using the ADC on P0.6 and outputs a corresponding PWM signal using the PCA 8-bit PWM mode on P0.0. The hardware configuration for software Example 1 is illustrated in Figure 2. A single N-channel Power MOSFET Q1 is used to drive the DC motor. The Power MOSFET should be chosen for the particular motor voltage and current requirements. A free-wheeling diode D1 is connected across the DC motor. When the MOSFET is turned off, the current through the motor inductance will continue to flow. The MOSFET drain voltage will rise to one diode-drop above the motor supply voltage. The current will then flow through the free-wheeling diode. The current will normally continue to flow until the transistor turns on again. Most low-voltage motor drive circuits employ Schottky power rectifiers for the free-wheel diode. Schottky rectifiers have a low forward voltage and a very fast reverse recovery time. Both are important factors in a motor drive application.

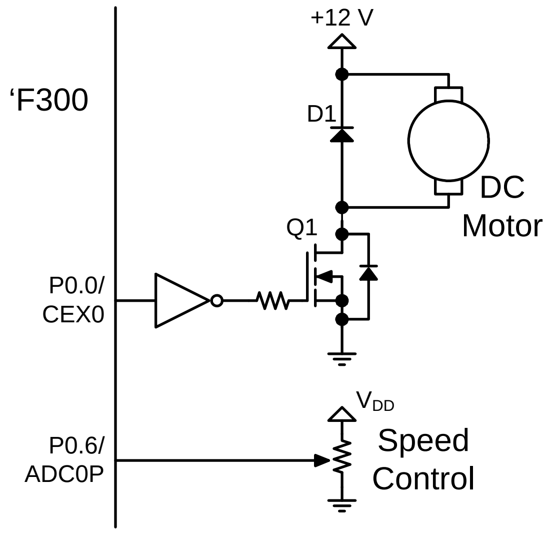


Figure 2. DC Motor Drive Circuit

The power MOSFET is driven by an inverting gate driver connected to P0.0. The port pins of ‘F300 are configured by default as inputs with a weak 100 kΩ pullup enabled. The port pins will remain high until the port is configured and the crossbar and peripherals are enabled. The port pins will also be configured as inputs with the weak pullup enabled while the reset pin is held low. By using an inverting driver, the power transistor will be off in the default state. If a non-inverting driver is used, a 10 kΩ pulldown resistor should be connected between the port pin and ground.

The gate driver should have a 3 V compatible input level threshold for use with a 3 V microcontroller. If the motor voltage is between 5 V and 15 V, the gate drive can be powered directly off the motor supply voltage. If the motor voltage is higher that 15 V, a separate gate drive supply voltage is needed, typically 5 V or 12 V. The Maxim Integrated Products MAX626 or TC426 has a 3 V compatible input, 5–20 V supply range, and 1.5 A peak drive current capability. A logic-level power MOSFET should be used when working with a gate drive supply voltage below 10 V.

The software for Example 1 is very simple. The **`main()`** function initializes the clock, ports, and peripherals and enters the **`while(1)`** loop. The **`while(1)`** loop reads the value of the potentiometer voltage using the **`avgADC()`** function and outputs the value to the PCA0 8-bit PWM by writing to the PCA0CPH0 special function register (SFR).

The **`PORT_Init()`** function configures the port I/O, peripherals, and enables the digital crossbar. The PCA0 CEX0 output is enabled by setting the appropriate bit in the XBR1 SFR. The CEX0 output is used for the 8-bit PWM. The pin skip SFR XBR0 is cleared so that no pins will be skipped. This configures the PCA0 output on P0.0. Clearing XBR0 is not strictly required, as this is the default state. P0.0 is configured for push-pull output by setting bit 0 in the P0MDOUT SFR. P0.6 is configured as analog input by clearing bit 6 in the P0MDIN SFR. Lastly, the crossbar is enabled by setting bit 6 in the XBR2 SFR.

The system clock, SYSCLK, is configured to operate at the maximum speed of 24.5 MHz. Programmable counter array, PCA0, is configured to use SYSCLK divided by four as a time-base for the 8-bit PWM. This yields a counter clock period of 160 ns and an 8-bit PWM frequency of 24 kHz (24.5 MHz/4/256 = 24 kHz).

PCA0 module 0 is configured for 8-bit PWM mode by writing 0x42 to the PCA0CPM0 SFR. The PCA0 interrupt is not used in this example.

The ADC in Example 1 is used in the polled mode. The **`ADC0_Init()`** function configures the ADC for polled mode by clearing the ADC0CN SFR. P0.6 is selected as the input for a single-ended measurement by writing 0xf6 to the AMX0SL SFR. The ADC gain is set to 1 and a conservative frequency of 1 MHz is chosen for the ADC clock. It is important to remember to also initialize the voltage reference and configure the ADC to use V_{DD} for full-scale by writing 0x0a to the REF0CN SFR.

The function **`readADC()`** reads the voltage on P0.6 one time using polled mode and returns the ADC value. The function **`avgADC()`** calls the **`readADC()`** function and will return the average value of 64 samples. Averaging the ADC reading minimizes the effects of noise and reduces jitter in the PWM output.

When using the PCA 8-bit PWM mode, a value of 0x00 corresponds to a duty cycle of 100% and a value of 0xFF corresponds to a duty cycle of 0.39% at the CEX0 output. A duty cycle of 0% may be achieved by clearing the ECOM0 bit in PCA0CPM0 SFR.

When using an inverting driver, the relationship is reversed. A value of 0x00 corresponds to a 0% duty cycle and a value of 0xFF corresponds to a duty cycle of 99.6% on the MOSFET gate. A duty cycle of 100% may be achieved by clearing the ECOM0 bit in PCA0CPM0 SFR. All software examples in this application note using 8-bit PWM are limited to 99.6% PWM for simplicity.

There are some cases where a 100% duty cycle is desirable. A 100% duty-cycle will effectively eliminate switching losses. Since the MOSFET never turns off, there are no switching losses in the MOSFET and no losses in the diode. The only power losses are conduction losses in the power MOSFET. If the motor is expected to run at full-speed most of the time, a maximum duty cycle of 100% is desirable.

In some motor control systems a maximum duty cycle of somewhat less that 100% is actually desirable. If a cycle-by-cycle current limit is used, a short low-time is necessary to reset the over-current latch. When using a high-side driver with a bootstrap driver, a short low pulse is needed to recharge the bootstrap capacitor. When using a transformer isolated gate driver, a DC signal is not permitted.

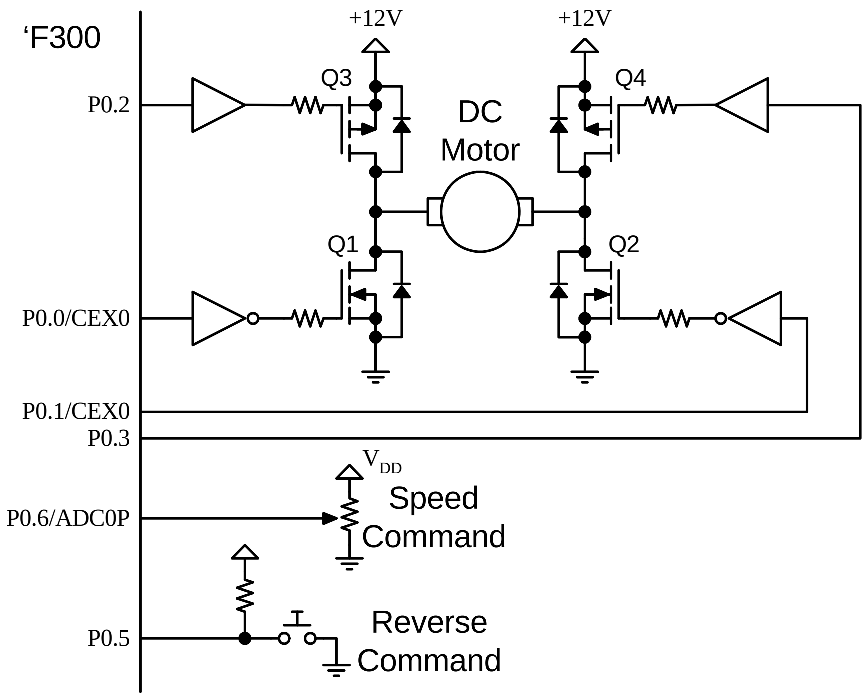


Figure 3. DC Motor Full-Bridge Circuit

3. Example 2—DC Motor with Reversing

Permanent-Magnet DC Motors are often used in applications that require the ability to reverse the direction of the motor. To reverse the direction of rotation, it is necessary to reverse the polarity of the voltage on the motor. This requires the use of an H-Bridge. An H-Bridge has four transistors as shown in Figure 3. When driving the motor in the forward direction, Q4 is turned on and a PWM signal is applied to transistor Q1. To drive the motor in the reverse direction, Q3 is turned on and a PWM signal is applied to Q2. In this example, the lower transistors are used for PWM speed control and the upper transistors are used for steering. Using this topology, it is possible to provide variable speed control in both directions.

In Figure 3, N-channel power MOSFETs are used for the low-side transistors and P-channel power MOSFETs are used for the high-side transistors. Using complementary power MOSFETs is very cost effective solution for DC motor drives below 20 V. As shown in Figure 3, the low-side gate drivers are inverting and the high-side gate drivers are non-inverting. The gate driver polarities are chosen to ensure that the power transistors are off while the port pins are in the reset configuration with the weak pullups enabled. A suitable gate driver IC is the TC428. The TC428 has one inverting and one non-inverting gate driver, 3 V compatible inputs, and a 5–20 V supply range.

In most applications, the crossbar is configured once and then never modified. However it is possible to modify the contents of the crossbar registers after initialization. In this example the digital crossbar is used as a PWM multiplexer by writing to the pin skip SFR XBR0 “on the fly”. This technique is also used in software examples 3 and 4. Extreme care should be used when writing to the pin skip register in this fashion. Changing the crossbar configuration may affect the states of other I/O pins.

The software for this example builds on the code of Example 1. The main loop now includes an *if* statement that checks the state of the reverse switch SW1. When the reverse button is pressed, the PWM is disabled and all of the P0 outputs are disabled by writing 0xFF to P0. When the button is released the motor will reverse directions.

The initialization functions are similar to Example 1, except that P0.0-3 are configured as push-pull outputs and P0.3 is initially high.

The **`reverse()`** function is called to reverse the direction of the motor. A flag bit **`Fwd`** is used to save the state of the motor. The **Fwd** bit is toggled and then used to determine which outputs to activate.

The general procedure to safely modify the crossbar is to first disable the crossbar by clearing the XBARE bit to zero. Then the crossbar is re-configured by writing to the crossbar registers, pin skip registers, and port configuration registers as necessary. Once re-configured, the crossbar is enabled by setting the XBARE bit to 1.

In this example, the PWM is first disabled by writing 0x00 to PCA0CPM0 and then all outputs are forced high by writing 0xFF to P0. This ensures that all outputs are forced high before reconfiguring the crossbar. The crossbar is then re-configured using the procedure described above. First the crossbar is disabled by clearing the XBARE bit to zero. The new pin skip value for the ‘F300 is written to the XBR0 SFR. The new P0 state is written to P0. The values written to XBR0 and P0 depend on the state of the **Fwd** bit. Once XBR0 and P0 are re-configured, the XBARE bit is set to 1 to enable the crossbar. The 8-bit PWM mode is then enabled by writing 0x42 to PCA0CPM0.

There is a potential problem reversing the motor in this manner. While the reverse switch SW1 is held down the motor may continue to spin for some time due to the inertia of the motor. While the motor is turning, it will generate a back-emf proportion to the speed of the motor. If the reverse button is released before the motor stops spinning, the motor back-emf will be shorted out by the upper transistors as described below.

Referring to Figure 4, suppose Q4 is initially on and the motor is turning in the forward direction. Assume the motor is turning and the back-EMF is about 6 V. Now the switch is pressed and all four transistors are turned off. The right side of the motor will be 6 V higher than the left side of the motor. Then the switch is released and Q3 is turned on. The left side of the motor is pulled up to the supply voltage and the back-emf of the motor is shorted by the internal diode of Q4.

The end result is that the motor stops and all energy stored in the mechanical inertia of the motor is dumped into Q4. This could easily damage the upper transistors during reversal. In some applications with a large frictional load, a fixed delay may be adequate to ensure the motor has time to stop. In other applications, the motor may take several seconds to come to a complete stop. A universal solution to this problem is demonstrated in "Example 3—DC Motor with Soft Reversing," on page 6.

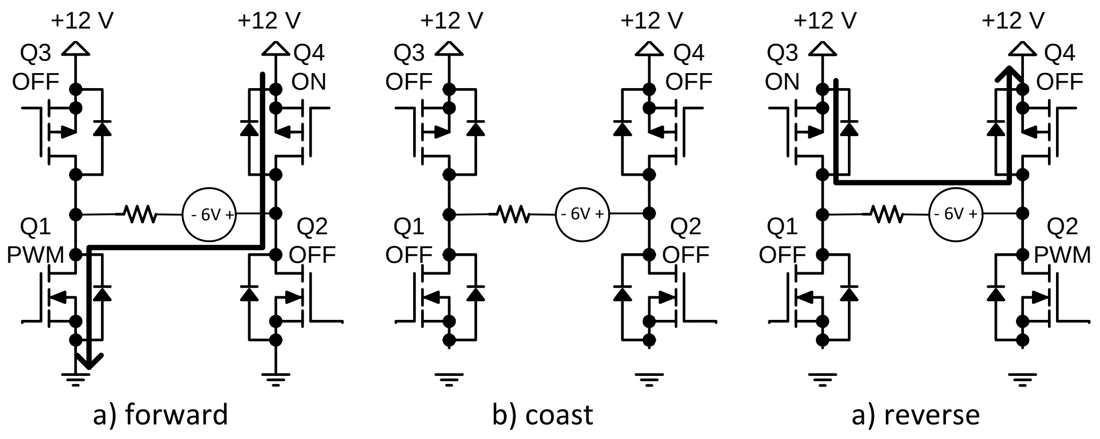


Figure 4. DC Motor Reversing Hazard

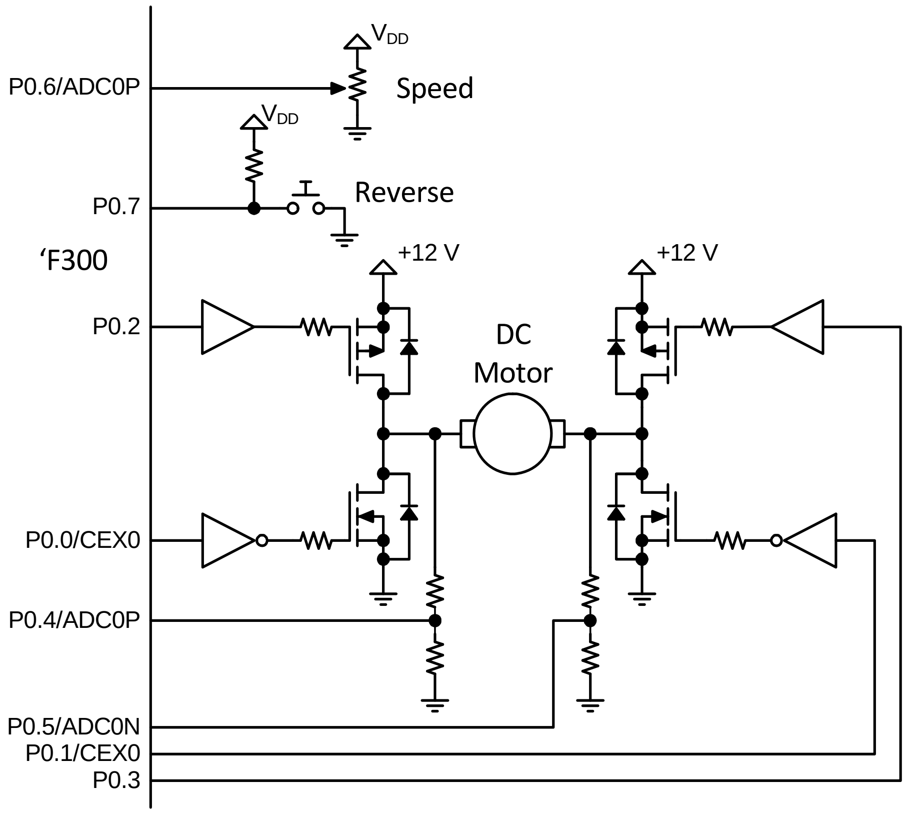


Figure 5. DC Motor Drive with Voltage Sensing

4. Example 3—DC Motor with Soft Reversing

This software example for a DC motor builds on Example 2 and provides soft reversing. To safely reverse a DC motor it is necessary to determine if the motor is still in motion.

A simple and effective method to determine if the motor is still spinning is to measure the differential voltage across the motor terminals. The ADC on the 'F3xx family can be configured to measure the differential voltage between any two inputs of the analog multiplexer. The programmable window detector may also be used to determine if the differential voltage has fallen within preset limits. In this software example, the motor will reverse after the differential motor voltage remains below 3% of full scale for 100 ms.

The hardware for this example is similar to Example 2 except for the addition of two resistor dividers connected to the motor terminals, as shown in Figure 5. The sense voltage outputs from the resistive dividers are connected to P0.4 and P0.5. The reverse switch is now on P0.7.

The software builds on Example 2. The main loop has been modified to detect motor stop. The **`detectStop()`** function first configures the ADC to measure the differential voltage between P0.5 and P0.4 by writing 0x54 to the AMX0SL SFR. The ADC and window detector are both used in polled mode. If the ADC value is within the preset window a counter is incremented. A 10 ms delay using timer T0 sets the sample time. Any sample outside the window will reset the counter. It will take 10 consecutive samples within the window before exiting the while loop. The **`detectStop()`** function will re-configure the ADC to measure the speed potentiometer before returning to the main loop.

The preprocessor macros VWINDOW, DTIME, and GSAMP are used to set the voltage window, delay time, and number of good samples required. These constants may be modified to suit a particular motor system.

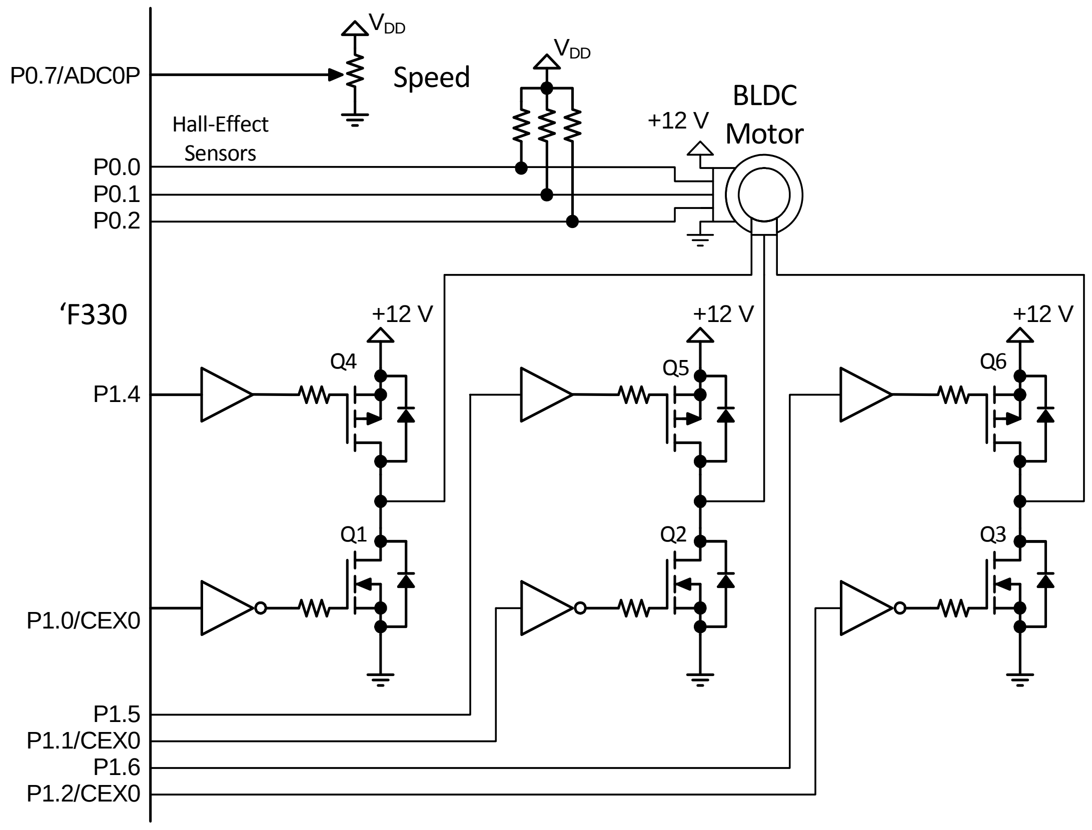


Figure 6. Brushless DC Motor Drive

5. Example 4—Brushless DC Motor

Brushless DC (BLDC) motors offer some advantages over conventional brush-commutated DC motors. The electronics and sensors effectively replace the role of the brushes, offering long life, reduced maintenance, and no brush noise. The torque-speed characteristics of a properly commutated BLDC motor are identical to the DC motor as shown in Figure 1. Thus, Brushless DC motors exhibit the same desirable qualities that make DC motors so well suited for variable speed control. This example provides simple open-loop control of a BLDC motor using Hall-effect sensors to control the motor commutation. The speed of the BLDC motor is controlled using a simple potentiometer. The characteristics of the BLDC motor controlled in this manner are similar to the DC motor in Example 1.

The hardware required for this example is illustrated in Figure 6. The C8051F330 was chosen for this example due to the requirement for at least 10 I/O pins. The motor is driven by six power transistors in a three-phase bridge configuration. The lower transistors Q1–3 are N-channel power MOSFETs. The upper three transistors are P-channel power MOSFETs. This simplifies the gate drive arrangement. Again, complementary gate drivers are used so that the power transistors are off in the default state. The gate drivers for the lower transistors Q1-3 are connected to P1.0–P1.2. and the upper transistors are connected to P1.4–P1.6. The analog speed input is on P0.7.

The Hall-Effect sensors are connected to P0.0–P0.2. Hall-Effect sensors have open-collector outputs and require pullup resistors. Check the motor specifications to ensure the Hall-effect sensors are configured properly. The open-collector outputs are usually 3 V compatible. However, the Hall-effect sensors also require a bias supply that typically requires more that 3.0 V. In most systems, the Hall-effect sensors can be powered off the motor supply voltage or the gate drive supply voltage.

When the MCU hits a breakpoint, the pins are effectively frozen in time. While this behavior is generally desirable, a breakpoint may leave the PWM outputs in the active state. A brushed DC motor will operate at full-speed when this occurs. However, a BLDC motor will stall with full voltage across one winding. The BLDC motor stall current is only limited by the internal resistance of the winding. This will most likely damage the power MOSFETs. The recommended procedure is to always disconnect the motor leads before single stepping code or using breakpoints. With the motor wires disconnected there is no path for current to flow.

The software for the BLDC motor example contains many new elements. A single PCA0 module is used in the 8-bit PWM mode. The crossbar is used as an output multiplexer to apply the PWM signal to P1.0, P1.1, or P1.2 as needed. The upper transistors are controlled by writing to P1.

The **`PORT_Init()`** function configures the crossbar and output pin assignments. The crossbar special function registers names and functionality for the ‘F330 are different than the ‘F300. The programmable counter array CEX0 output is enabled by writing 0x01 to the XBR1 SFR. The crossbar is configured to skip all P0 pins by writing 0xFF to the P0SKIP SFR. The P1SKIP SFR is initialized to output the CEX0 signal on P1.0. The P1SKIP SFR will be used later to multiplex the CEX0 signal to P1.0, P1.1, or P1.2. P1.0–P1.2 and P1.4–P1.6 are configured as push-pull outputs. P0.7 is configured as an analog input.

The programmable counter array time-base is configured to use SYSCLK/4, and the counter is started. However, the Module 0 mode SFR is not initialized for 8-bit PWM. The CEX0 signal, initially on P1.0, will remain high until the Hall-effect position is determined.

The **`main()`** function first initializes everything and sets the **`start`** flag bit. The main loop first checks the position of the Hall-effect sensors using the **`hallPosition()`** function. If the **`start`** flag bit is set or the Hall position has changed, the motor is commutated by calling the **`commutate()`** function. Next the speed input is read and the speed setting is written to the PWM output.

The **`hallPosition()`** function returns a zero on an error condition. This occurs if the Hall-effect inputs are all high or all low. If an error occurs, the main loop disables all outputs by calling the **`coast()`** function. The start bit is also set on an error condition to force a commutation on the next valid Hall position reading.

The **`readHalls()`** function reads and debounces the Hall-effect code on the Hall-effect input port pins. This function waits for three consecutive identical readings. This reduces the likelihood of an erroneous reading while the Hall-effect code is changing. The number of consecutive samples required for a good reading can be modified for a particular system by changing the GSAMP preprocessor macro.

The **`hallPosition()`** first reads the Hall-effect code by calling the **`readHalls()`** function described above. The Hall code pattern is stored in the constant array **`hallPattern[]`**. A single line *for* loop with post decrement is used to find the corresponding index for the matching Hall-effect code. The **`hallPosition()`** function returns a value 1 through 6 if it finds a matching pattern. If no match is found the **`hallPosition()`** function returns a zero value.

The **`commutate()`** function first disables the PWM by writing 0x00 to the PCA0CPM0 SFR. The upper transistors are also disabled by writing 0xff to P1. The **`commutate()`** function uses the index obtained from the **`hallPosition()`** function. Two constant arrays, **`skipPattern[]`** and **`P1Pattern[]`**, are used to store the patterns for the P1SKIP and P1 sfrs. The new values are written to the P1SKIP and P1 sfrs using the pattern index. Lastly, the 8-bit PWM is enabled by writing 0x42 to the PCA0CPM0 SFR.

The **`commutate()`** function is used to initialize the outputs on start-up, to change the state of the outputs when the Hall position changes, and to restart the motor after a Hall error has been corrected.

The patterns stored in **`hallPattern[]`**, **`skipPattern[]`** and **`P1Pattern[]`** may need to be modified to suit a particular motor system. There is no universal standard for the Hall-effect pattern or the commutation pattern. Consult the motor manufacturers data sheet for the particular motor you are using. Carefully check both patterns against the manufacturers data sheet. It may be necessary to swap the bit patterns for phase A and phase C to obtain the desired bit pattern. Also, check the correspondence between the Hall-effect pattern and the commutation pattern. It may be necessary to change the offset between the two patterns. The patterns listed are for a Pittman N2311A011 BLDC motor.

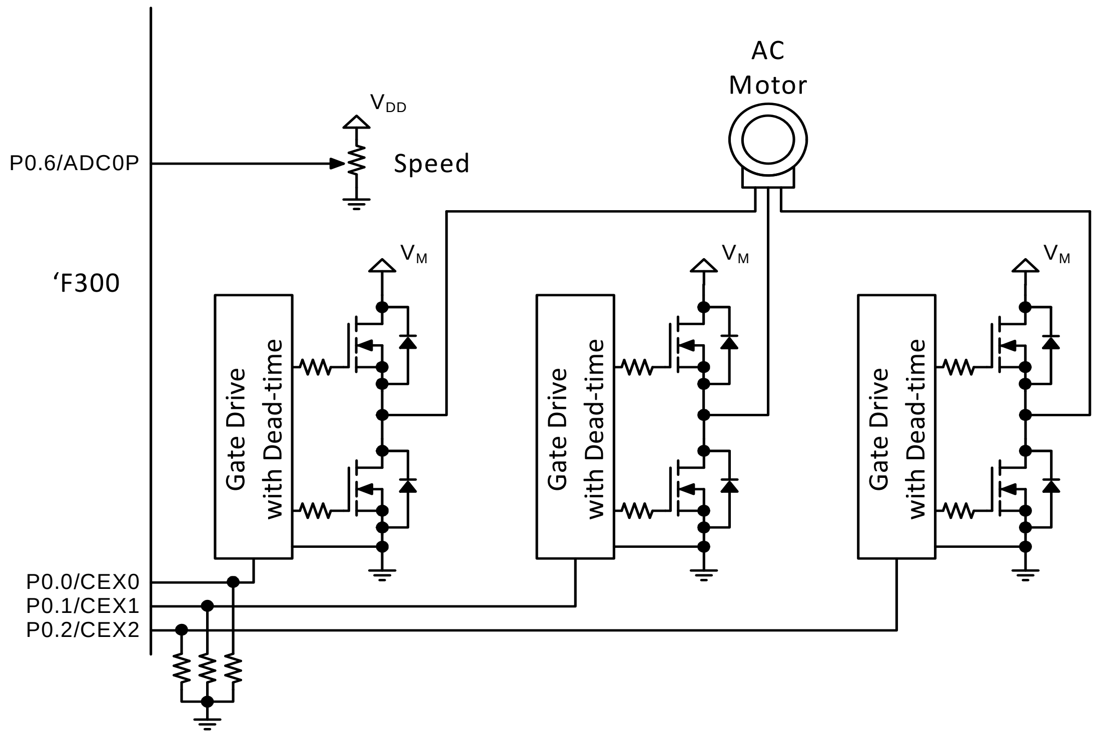


Figure 7. AC Induction Motor Drive

6. Example 5—AC Induction Motor

The previous examples have focused on small low-voltage motors. DC motors and BLDC motors offer competitive solutions for low-voltage motor drive systems. AC Induction motors are typically used only in off-line applications. The C8051F3xx family may be used to provide a cost-effective solution for constant V/Hz control of Fractional Horsepower motors. Fractional Horsepower motors range from 1/4 to 3/4 horsepower and normally have an operating voltage of 110 V_{AC} to 240 V_{AC}.

AC Induction motors can be used for a wide range of applications with radically different performance requirements. The simplest control method is called constant V/Hz control. This control methodology is used for variable speed or adjustable speed AC induction motor drive systems. AC inductions motors using constant V/Hz control can be used for fans, blowers, air handlers, pumps, submersible pumps, and compressors. A C8051F3xx MCU can be used to provide a low-cost solution for these applications.

At the other end of the performance spectrum, vector control may be used to provide a high-performance motion control system that meets or exceeds the performance of a DC servo drive. Vector control normally requires the use of a DSP to perform complex matrix algebra transforms. The intellectual appeal of Vector control tempts engineers to use vector control in applications that do not really require the performance. However the cost-conscious system designer will appreciate much lower cost of the constant V/Hz system.

The simplified schematic for Example 5 AC Induction motor is shown in Figure 7. A three-phase transistor bridge is used to drive the AC Induction motor. The power transistors might be power MOSFETs or insulated gate bipolar transistors (IGBTs). IGBTs will usually provide lower power losses for 230 V_{AC} applications greater than 1/4 HP. P0.0, P0.1, and P0.3 are used to control the gate drive of the three-phase bridge.

High-Voltage ICs may be used to provide a simple, low parts count, cost-effective gate drive. Dead-time is required to prevent cross-conduction and increased power losses. The switching time is limited by the performance of the power transistors and the circuit parasitics. The dead-time must also account for any mismatch in the turn-on and turn-off delay of the system. High-voltage ICs are available with built in dead-time for little or no additional cost.

The IR2103S is a 600 V half-bridge gate driver with a fixed dead-time of 520 ns and 3 V compatible inputs. The IR2104S provides the same features plus an active low shut-down that disables both outputs. The shut-down feature is very useful in more complex systems for both start-up and fault protection.

This example provides simple open loop V/Hz control for AC Induction motors using the C8051F300 MCU. The MCU reads the value of a speed control potentiometer and generates the three-phase sine wave PWM required to drive the power transistors. Three PCA modules configured for 8-bit PWM are used to generate three-phase PWM. The three-phase PWM waveforms are shown in Figure 8.

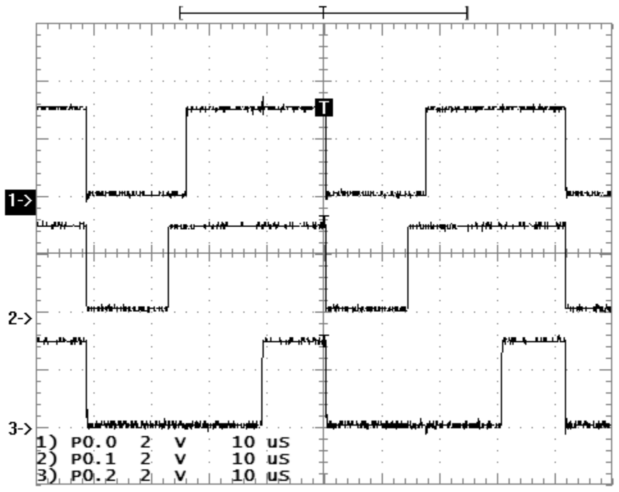


Figure 8. Three-Phase PWM

PCA0 outputs CEX0, CEX1, and CEX2 are enabled by writing 0xC0 to the XBR1 SFR. The XBR0 SFR is cleared to not skip any pins. This configures the PCA0 outputs on P0.0, P0.1, and P0.2. The first three outputs P0.0–P0.2 are configured as push-pull outputs by writing 0x07 to the P0MDOUT SFR. P0.6 is configured as input for the analog speed control. When the crossbar is enabled this will enable the three outputs.

All three PCA modules are configured for 8-bit PWM mode by writing 0x42 to their respective capture/compare mode registers. The PWM high times are initialized to 50%.

At startup, P0.0–P0.2 will be initially pulled high by the weak pullups by default. Pulldown resistors are used force the output low at start-up. This turns on the lower transistors and charges the bootstrap supply used for the high-side gate drive. The PCA is initialized before the crossbar is enabled. When the crossbar is initialized a 50% waveform will appear on outputs P0.0 through P0.2.

The main loop reads the ADC using averaging and stores the value in the global variable **`Volts`**. All sine wave generation and updating is done using interrupts.

Timer T0 is configured for 16-bit counter mode 1. The timer uses the 24.5 MHz SYSCLK divided by four. The timer is started and TF0 is set to a 1 to force an initial interrupt.

The Timer Interrupt Service Routine **`Timer_ISR()`** is used to generate a periodic interrupt with a period of about 1 ms. After setting up the next interrupt, the **`Timer_ISR()`** will call the **`Update()`** function.

The **`Update()`** function updates the three PWM duty cycles based on a sampled sine wave. The value of theta is calculated based on the relationships illustrated in Equation 11. First the global variable **`Volts`** is copied to the local variable **`omega`**. The variable **`omega`** is scaled so that a value of 0x04 corresponds to a sine frequency of about 1 Hz. The value of **`omega`** is integrated by adding **`omega`** to the global variable **`Sum`**. **`Sum`** is a 16-bit unsigned int data type. The upper byte of **`Sum`** is copied into **`theta`**.

$$\omega \propto V$$

$$\theta = \int \omega dt$$

$$\theta = \Sigma\omega$$

```
Sum = Sum + omega
theta = Sum / 256
```

Equation 1. Equation 1

The 8-bit variable **`theta`** is passed to the **`sineWave()`** function. The value returned from the **`sineWave()`** function is stored in PCA0CPH0. This sets the duty cycle of CEX0. The other two PWMs are updated using **`theta`** plus 0x55 and **`theta`** plus 0xAA. This generates three sine modulate PWM signals 120º apart.

The **`sineWave()`** function uses the **`sine[]`** look-up table containing 256 signed 8-bit values. The **`sine[]`** value corresponding to **`theta`** is multiplied by the **`Volts`** variable. The most significant byte of the product stored in an 8-bit variable. An offset of 0x80 is added to the output value to provide a sine wave centered about 50%.

7. Example 6—PWM using High-Speed Output Mode

The 8-bit PWM mode of the PCA provides sufficient resolution for most small motor drive applications. The PWM frequency is normally chosen to be just above the audible range. The optimum PWM frequency for small motors is in the range of 16 to 24 kHz. Integral-Horsepower AC Induction motors often employ lower switching frequencies to reduce switching losses. The 24 kHz frequency is suitable for most small motor drives.

The 8-bit PWM frequency can be set to 8, 24, or 96 kHz when operating from a system clock frequency of 24.5 MHz. An 8-bit PWM frequency of 16.0 or 19.1 kHz may be obtained by using T0 overflow as the PCA clock source.

Some applications may require more resolution than 8 bits. Higher resolution may be required to achieve speed regulation of better than 1% using DC or BLDC motors. AC Induction motor systems that require greater than 100 to 1 speed range may also require higher resolution.

Higher resolutions and arbitrary PWM frequencies can be obtained by using the high-speed output (HSO) mode of the programmable counter array to generate a PWM signal. The HSO mode can be used to generate PWM waveforms with up to 16-bit resolutions and 40 ns edge timing. This corresponds to an effective resolution of 10.25 bits or 0.0816% at 20.0 kHz.

The trade-off is that the software latency limits the minimum high-time and low-time. The MCU must have sufficient time to interrupt the current process and write the new values to the output compare registers before the next edge is scheduled to occur. The latency can be minimized by assigning the PCA0 to high priority, caching the edge timing data, and using an alternate register set for the interrupt service routine. The CPU expends a significant portion of its available processing capability servicing the frequent PCA interrupts.

The software for Example 6 reads the value of the speed control potentiometer and outputs a 20.0 kHz PWM waveform on P0.0. The system clock, ADC, and port initialization are identical to Example 1. The **`PCA0_Init()`** function configures the PCA to use the system clock and configures Module 0 for high-speed output mode. The PCA initialization also schedules the first PCA interrupt.

Two global variables **`NextEdge`** and **`cycle`** are used by the PCA0 Interrupt service routine. **`NextEdge`** is used to cache the edge timing data one edge ahead of time to reduce latency. The global bit **`cycle`** is used to keep track of which edge is to occur next. Since the HSO mode will toggle the output, a software flag bit is required. Using a flag bit is more robust than polling the output pin state because it is independent of compare matching.

The preprocessor macro LATENCY is set to a value just greater than the update latency to provide reliable operation. Preprocessor macro calculations are used to calculate PERIOD and HTSPAN. The high-time span HTSPAN is the PERIOD minus two times LATENCY.

The main loop polls the ADC using averaging. The value from the ADC is multiplied by the desired high-time span HTSPAN. The product is then incremented by adding LATENCY. The final result is then stored in a global variable **`HiTime`**. Temporary variables **`x`** and **`y`** are used to calculate the intermediate values. The scaling operation requires a **`long int`** data type. The final result is 16-bits. The global variable **`HiTime`** should not be used for intermediate calculations. The interrupt service routine might pickup the new value at any time and would use nonsensical values for the PWM. The PCA0 interrupt is also temporarily disabled during the **`HiTime`** update to ensure that an interrupt does not occur until both bytes have been updated.

The PCA interrupt service routine **`PCA0_ISR()`** first updates the PCA0CPx0 registers and then clears the PCA0 Module 0 capture/compare flag CCF0. Once the flag has been cleared, it is safe for the next interrupt to occur. Depending on the state of the **`cycle`** bit, the **`NextEdge`** is incremented by **`HiTime`** or **`Period`** minus **`HiTime`**.

A comparison of the HSO PWM is shown versus the 8-bit PWM in Figure 9. The frequency for the HSO PWM is exactly 20 kHz and the minimum high time is 1.8 µs. The frequency for the 8-bit PWM is 24 kHz and the minimum high-time is 160 ns.

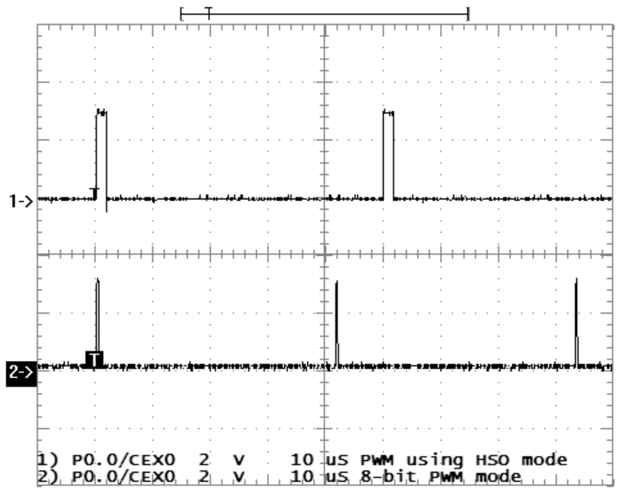


Figure 9. Minimum High-Time Comparison

8. Example 7—Center-aligned PWM

This example demonstrates how to use the PCA high-speed output mode to generate center-aligned PWM waveforms with dead-time. Center-aligned PWM with dead-time may be used for DC, BLDC, or AC induction motors. DC and BLDC motors that require active braking must use a PWM scheme that alternatively turns on the top and bottom transistors. Motors that provide positive and negative torque control in both forward and reverse directions are called servo motors or four-quadrant drives. Servo motors also require pulse-width modulating both upper and lower transistors. AC Induction motors always use this type of PWM scheme to generate sine waves. When pulse-width modulating both upper and lower transistors, dead-time is required between the activation of the upper and lower transistors. The dead-time function may be performed by the MCU or integrated into the gate drive.

Using center-aligned PWM has benefits. It is very easy to generate the required dead-time. The complementary PWM signal with dead-time may be obtained by adding a small number to the high-time of the first signal and inverting. Using center-aligned waveforms also has the benefit of doubling the frequency between phases and reducing the ripple current. This is particularly important for large motors with low inductance.

The desired center-aligned waveforms are illustrated in Figure 10. The period is measured with respect to the center of the high-time of the top PWM signal. When multiple PWM channels are used, all signals are aligned with respect to the center of the waveforms. The top and bottom PWM signals shown in Figure 10 are active high. Both top and bottom signals are inactive during the dead-time.

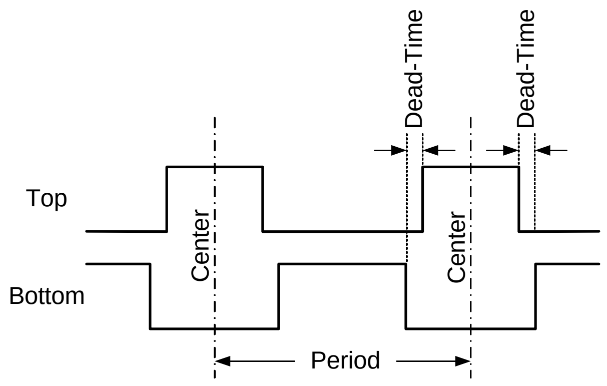


Figure 10. Center-Aligned PWM Signals

Three PCA modules are used to generate the desired waveforms. One module is used as a master. The master module is the only module that generates interrupts. The master channel generates 50% duty cycle waveform. The center of the high-times and low-times of the other channels are aligned with the rising and falling edges of the master channel as shown in Figure 11.

The F300 port I/O and pin assignments are similar to Example 5. The pin skip SFR XBR0 is cleared and no pins are skipped in the priority crossbar assignments. The XBR1 SFR is initialized to output CEX0, CEX1, and CEX2. The value 0x07 is written to P0MDOUT to configure P0.0, P0.1, P0.2 as push-pull outputs.

The main loop is identical to Example 6. The **`PCA0_Init()`** function initializes all three channels for high-speed output mode. The interrupt is enabled only on Module 0. The PCA initialization function schedules the first interrupt and next edges for a 50% duty cycle. The relative polarities of all signals are defined by the initialization. P0.2 is inverted because it does not have an edge scheduled for the first half-cycle.

The PCA interrupt service routine **`PCA0_ISR()`** is similar to Example 7. Three 16-bit PCA0CP compare special function registers must by updated before the interrupt flag is cleared. The LATENCY macro value must be increased to account for the extra instructions.

The **`NextEdge`** global variables are calculated for all three modules. The calculations are different depending on the state of the **`cycle`** bit.

The center-aligned module’s next edge values are calculated relative to the master module. If **`cycle`** is 1, the center-aligned **`NextEdge1`** and **`NextEdge2`** are calculated by first incrementing **`NextEdge0`**, then adding or subtracting the **`Hitime`** from **`NextEdge0`**, and then adding or subtracting the dead-time. If **`cycle`** is 0, **`NextEdge1`** and **`NextEdge2`** are calculated by adding or subtracting the Hitime from **`NextEdge0`**, adding or subtracting the dead-time, and incrementing **`NextEdge0`**. The minimum pulse width is reduced by adding half the LATENCY. This shifts the center-aligned waveforms slightly so that the edges are aligned with the master channel at the minimum high-time conditions.

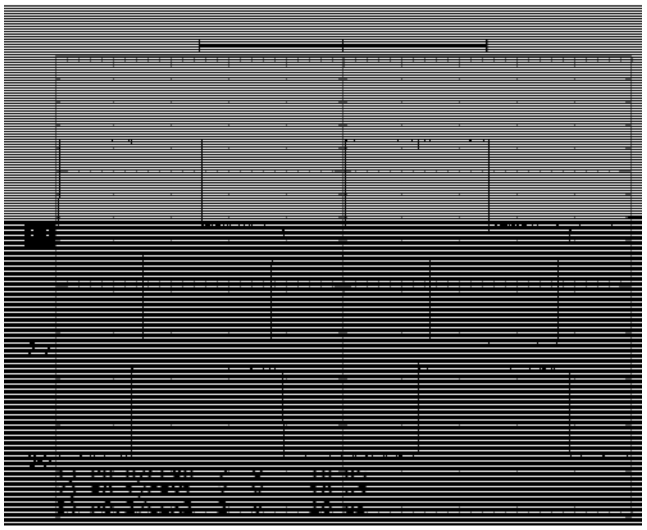


Figure 11. Center -Aligned PWM Waveforms

The measured center-aligned PWM waveforms are shown in Figure 11. The top waveform is the master channel output on P0.0. The scope is triggered off the master channel. The middle and bottom waveforms are P0.1 and P0.2. These signals may be used to drive the upper and lower transistors of a half-bridge. The dead-time is configured for inverting gate drivers. P0.1 and P0.2 are never low at the same time. There is a finite dead-time between the rising edge of P0.1 and the falling edge of P0.2.

The F300 has three PCA modules and may be used to provide two complementary center-aligned waveforms with dead-time. Two complementary PWM waveforms are sufficient to drive a DC motor using a half-bridge. This provides active braking and synchronous rectification. A simplified diagram of a half-bridge motor drive is shown in Figure 12.

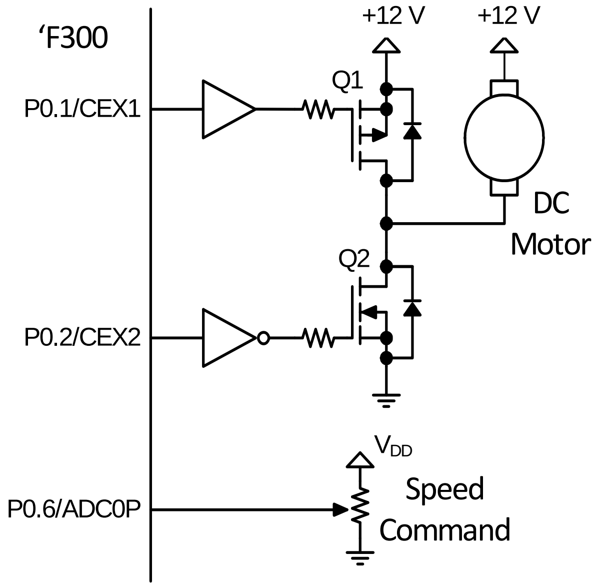


Figure 12. Half-Bridge DC Drive

One I/O pin can be saved by moving the master channel to CEX2 and configuring the crossbar to output only CEX0 and CEX1. The master module output is useful for debugging purposes, but may not be required in some systems.

The F310 has 5 PCA modules and may be used to generate four center-aligned PWM waveforms with dead-time. Four center-aligned PWM waveforms may be used to drive the DC servo motor as shown in Figure 13.

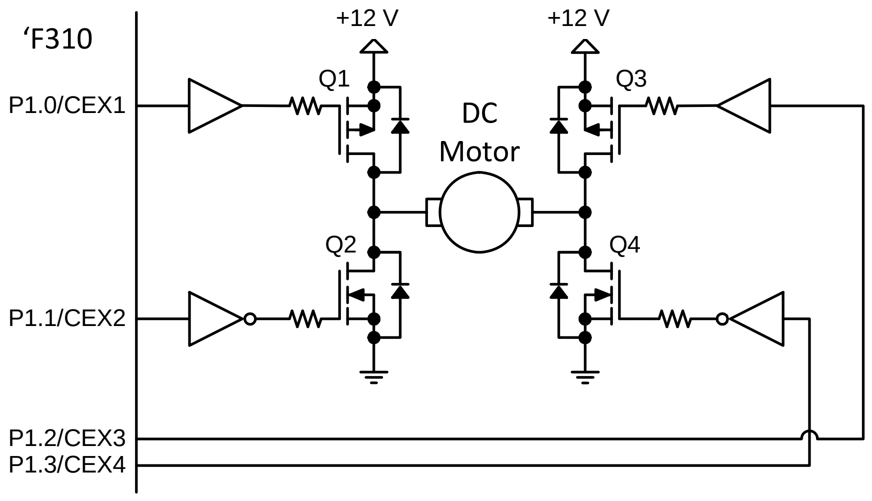


Figure 13. DC Servo Drive

9. Example 8—Quadrature Decode

Closed-Loop speed control requires position feedback. There are many different kinds of position sensors. Common examples are optical encoders, Hall-effect encoders, tachometers, and potentiometers.

This software example demonstrates a low-cost solution for the most common position sensor: the optical quadrature encoder. An optical encoder uses a semiconductor photo-detector and LED to detect slots in a disc or dark bands on a reflective wheel. Single channel optical encoders provide speed feedback but cannot detect which direction the motor is turning. Two-channel quadrature encoders are capable of providing both speed and direction.

The quadrature decode software can be combined with the DC motor control software used in Example 2 or the Center aligned PWM used in Example 7 for servo-positioning applications. This provides an integrated solution for closed loop motor control to reduce external parts count and reduce system cost.

The two outputs of a quadrature encoder, CHA and CHB, are 90º out of phase as shown in Figure 14. If the motor is turning in the forward direction, CHA will be leading CHB. If the motor is turning in the reverse direction, CHB will be leading CHA. The direction of rotation may be detected by taking the exclusive OR of the two signals. The results of taking the exclusive OR of CHA and CHB is indicated by the letters T and F in Figure 14. When the motor is turning in the forward direction, immediately after any edge detection on CHA, the XOR of CHA and CHB will be true. Immediately after any edge detection on CHB, while turning in the forward direction, CHA XOR CHB will be false.

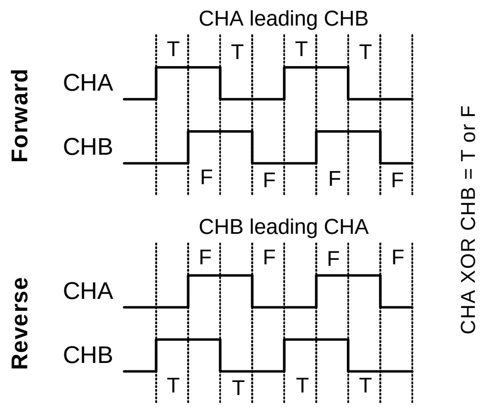


Figure 14. Quadrature Decoder Operation

Using this information a simple algorithm can be obtained using edge triggered interrupt events. On CHA edge interrupts, the position should be incremented if CHA XOR CB is true, or decremented if false. Conversely, on CHB edge interrupts the position should be incremented if CHA XOR CHB is false, or decremented if true.

The hardware configuration for software Example 8 requires that quadrature encoder CHA be connected to P0.0 and CHB be connected to P0.1. Pullup resistors are typically required for most quadrature encoders. Some encoders are specified to source only a few microamps. These will also require pullup resistors. The default weak pullups on P0 are typically not strong enough to drive the encoder signals high with an appropriate rise time.

Ensure that the quadrature encoder is compatible with 3 V CMOS logic. The open-collector outputs of most encoders should work with a pullup resistor to 3 V. The pullup resistor value should be decreased to keep the sink current approximately the same value. The encoder also requires a voltage power supply for the LEDs and internal circuitry. Some encoders may require a regulated 5.0 V supply. However, the open-collector outputs can still be used with pullups to a 3 V supply.

The software for Example 8 uses the UART with an ASCII terminal to display the position of the quadrature encoder. The position is stored in the global variable **`Position`**. The position is updated by two interrupt service routines for external interrupts INT0 and INT1.

The UART is enabled and P0.4 is configured for push-pull output. P0.0 and P0.1 are skipped as a precautionary measure. This will be required if any other peripherals are enabled in the crossbar.

The external interrupt initialization function **`EINT_Init()`** configures INT0 and INT1 to use P0.0 and P0.1 respectively. Both INT0 and INT1 are configured for edge activated interrupts. The initial trigger polarity for each channel is determined by polling P0.0 and P0.1. The external interrupts are configured for high priority and enabled.

The external interrupt service routines are identical except for respective change in the interrupt flag, polarity bit, and count direction. First the trigger polarity bit is toggled. Then a nested **`if...else`** statement is used to test the state of both polarity bits. For INT0, if both polarity bits are true or both polarity bits are false, the position will be incremented. Otherwise, the position will be decremented. For INT1, if both polarity bits are true or both polarity bits are false, the position will be decremented. Otherwise, the position will be incremented. This is the equivalent of a logical exclusive OR function. This implementation uses simple bit tests and is very code efficient.

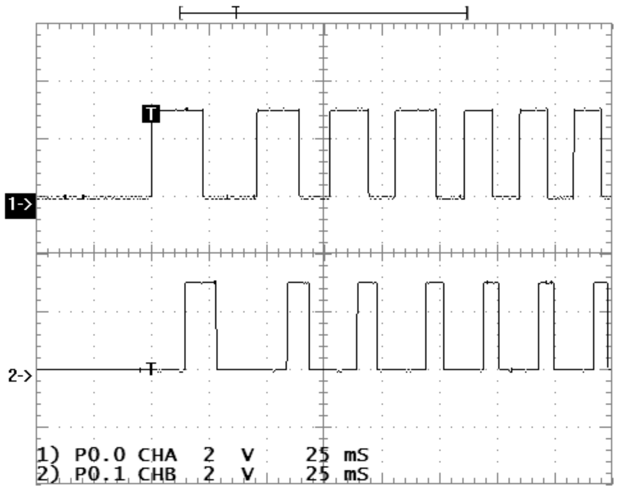


Figure 15. Quadrature Decode Measurements

The measured waveforms for a quadrature encoder are shown in Figure 15. This type of quadrature decode using interrupts is a viable solution up to speeds of about 50,000 counts per second. The number of counts per second is four times the number of pulses per second. There are four edges in each pulse. This is suitable for a medium-speed motor (<8000 RPM) with a low-resolution encoder (100 ppr) or a low-speed motor (<1500 RPM) with a high-resolution encoder (500 ppr). This performance range covers many consumer and automotive applications. High-performance industrial servo drives typically require much higher count rates up to 1 million counts per second. These types of applications will require a hardware based quadrature decoder interface.

Appendix A—DC Motor Code

```
//-----------------------------------------------------------------------------
// Example 1
// DC Motor Control
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 12MAR04
//
// This program provides simple DC motor control using the PCA 8-bit PWM Mode.
// The ADC is used to read the potentiometer voltage on P0.6. The ADC uses
// polled mode and 64 sample averaging. The 8-bit PWM is configured to operate
// at 24 kHz. The PWM high-time varies from a minimum of 160 ns to a maximum of
// 100%.
//
// Target: C8051F30x
//
// Tool chain: KEIL Eval 'c'
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f300.h>                 // SFR declarations

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void SYSCLK_Init (void);               // Initialize SYSCLK
void PORT_Init (void);                 // Initialize XBR and Port Pins
void PCA0_Init (void);                 // Initialize PCA0
void ADC0_Init (void);                 // Initialize ADC
unsigned char readVin(void);           // read ADC using polling
unsigned char avgVin(void);            // returns avg ADC reading


//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
void main (void)
{
   PCA0MD &= ~0x40;                    // Disable Watchdog Timer

   SYSCLK_Init ();                     // Initialize system clock
   PORT_Init ();                       // Initialize crossbar and GPIO
   ADC0_Init();                        // Initialize ADC for polled mode
   PCA0_Init ();                       // PCA0 for 8-bit PWM

   EA = 1;                             // enable global interrupts

   while (1)
   {
      PCA0CPH0 = avgVin();             // get avg reading and output to PWM
   }
}
```

```
//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use the internal 24.5MHz
// oscillator as its clock source.  Also enables missing clock detector reset
// and the VDD Monitor.
//
void SYSCLK_Init (void)
{
   OSCICN = 0x07;                       // configure internal oscillator for
   RSTSRC = 0x06;                       // enable missing clock detector
                                        // and VDD Monitor.
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - /PWM - active low PWM signal
// P0.1 -
// P0.2 -
// P0.3 -
// P0.4 -
// P0.5 -
// P0.6 - Analog Input
// P0.7 - C2D
//
void PORT_Init (void)
{
   XBR0     =  0x00;                     // don’t skip any pins
   XBR1     =  0x40;                     // Enable CEX0 on P0.0
   P0MDOUT  =  0x01;                     // Enable P0.0 as a push-pull output
   P0MDIN   = ~0x40;                     // Configure P0.6 for analog input
   XBR2    |=  0x40;                     // Enable crossbar

}


//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------
//
//
void PCA0_Init (void)
{

   PCA0MD = 0x02;                       // PCA uses sysclk/4, no PCA interrupt

   PCA0CPM0 = 0x42;                     // Module 0, 8-bit PWM Mode

   PCA0L = 0x00;                        // reset the timer
   PCA0H = 0x00;
   PCA0CPL0 = 0x00;
   PCA0CPH0 = 0x00;                     // Initialize to minimum duty
   CR = 1;                              // Start PCA0 timer
}
```

```
//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------

void ADC0_Init (void)
{
   ADC0CN = 0x40;                      // Low-power tracking mode;
                                       // ADC0 conversions are initiated
                                       // on writes to AD0BUSY;
   AMX0SL = 0xf6;                      // select P0.6, single-ended
   ADC0CF = 0x81 ;                     // AD0SC=4 gain =1
   REF0CN = 0x0a;                      // ADC uses Vdd for full scale
   EIE1 &= ~0x04;                      // disable ADC0 EOC interrupt
   AD0EN = 1;                          // enable ADC
}

unsigned char readVin(void)
{
   AD0INT = 0;                         // clear ADC0 end-of-conversion
   AD0BUSY = 1;                        // initiate conversion
   while (!AD0INT);                    // wait for conversion to complete
   return ADC0;
}

unsigned char avgVin(void)
{
   unsigned char i, result;
   unsigned int sum;

   sum = 0;
   for (i = 64; i != 0; i--)           // repeat 64 times
   {
      sum += readVin();                // read ADC and add to sum
   }
   result = (unsigned char)(sum>>6);   // divide by 64 and cast to uchar
   return result;                      // return average reading
}
```

Appendix B—DC Motor with Reversing Code

```
//-----------------------------------------------------------------------------
// Example 2
// DC Motor Control with Reversing
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 12MAR04
//
// This program provides DC motor control with hard reversing using the PCA
// 8-bit PWM Mode. A single PCA module is used to generate an 8-bit PWM. The
// pin skip register XBR0 is used to multiplex the PWM between two outputs
// P0.0 and P0.1. These two outputs are used to drive the lower transistors in
// an H-Bridge configuration. P0.2 and P0.3 are used to drive the upper
// transistors.
//
// The ADC is used to read the poteniometer voltage on P0.6. The ADC uses polled
// mode and 64 sample averaging.
//
// When the pushbutton is pressed and held the PWM duty cycle will be set to
// zero and all transistors will be turned off. This will allow the motor to
// coast. Releasing the switch will cause the motor to reverse. If the switch
// is released before the motor comes to a complete stop, the motor will
// abruptly stop and reverse directions.
//
// Target: C8051F30x
//
// Tool chain: KEIL Eval 'c'
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f300.h>                 // SFR declarations

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void SYSCLK_Init (void);               // Initialize SYSCLK
void PORT_Init (void);                 // Initialize XBR and Port Pins
void PCA0_Init (void);                 // Initialize PCA0
void ADC0_Init (void);                 // Initialize ADC
unsigned char readVin(void);           // read ADC using polling
unsigned char avgVin(void);            // returns avg ADC reading
void coast (void);
void reverse (void);
//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------
bit Fwd;
sbit SW1 = P0^5;
//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
void main (void)
```

```
{

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

   SYSCLK_Init ();                     // Initialize system clock
   PORT_Init ();                       // Initialize crossbar and GPIO
   ADC0_Init();                        // Initialize ADC for polled mode
   PCA0_Init ();                       // PCA0 for 8-bit PWM
   Fwd = 1;                            // start in forward direction
   EA = 1;                             // enable global interrupts

   while (1)
   {
      if(SW1)                          // if not pressed
      {

         PCA0CPH0 = avgVin();          // get avg reading and output to PWM
      }
      else
      {
         coast();                      // if pressed, coast
         while(!SW1);                  // wait for button to be released
         reverse();                    // reverse directions
      }
   }
}

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use the internal 24.5MHz
// oscillator as its clock source.  Also enables missing clock detector reset
// and the VDD Monitor.
//
void SYSCLK_Init (void)
{
   OSCICN = 0x07;                      // Configure internal oscillator for
                                       // highest frequency.
   RSTSRC = 0x06;                      // Enable missing clock detector
                                       // and VDD Monitor.
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - /Abot - active low PWM signal drives bottom transistor
// P0.1 - /Bbot - active low PWM signal drives bottom transistor
// P0.2 - /Atop - active low signal drives top transistor
// P0.3 - /Abot - active low signal drives top transistor
// P0.4 -
// P0.5 - Switch
// P0.6 - Vin
// P0.7 -
//
void PORT_Init (void)
{
   XBR0    = ~0x01;                    // skip all except P0.0
   XBR1    =  0x40;                    // Enable CEX0 on P0.0
   P0MDOUT =  0x0f;                    // P0.0 - P0.3 are push-pull outputs
   P0MDIN  = ~0x40;                    // Configure P0.6 for analog input
   P0      = ~0x08;                    // P0.3 low
   XBR2   |=  0x40;                    // Enable crossbar
}


//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------
//
//
void PCA0_Init (void)
{

   PCA0MD = 0x02;                      // PCA uses sysclk/4, no interrupt

   PCA0CPM0 = 0x42;                    // Module 0, 8-bit PWM Mode

   PCA0L = 0x00;                       // reset the timer
   PCA0H = 0x00;
   PCA0CPL0 = 0x00;                    // Initialize to minimum duty
   PCA0CPH0 = 0x00;
   CR = 1;                             // Start PCA0 timer

}

//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------

void ADC0_Init (void)
{
   ADC0CN = 0x40;                      // Low-power tracking mode;
                                       // ADC0 conversions are initiated
                                       // on writes to AD0BUSY;
   AMX0SL = 0xf6;                      // select P0.6, single-ended
   ADC0CF = 0x81;                      // AD0SC=4 gain =1
   REF0CN = 0x0a;                      // ADC uses Vdd for full scale
   EIE1 &= ~0x04;                      // disable ADC0 EOC interrupt
   AD0EN = 1;                          // enable ADC
}

unsigned char readVin(void)
{
   AD0INT = 0;                         // clear ADC0 end-of-conversion
   AD0BUSY = 1;                        // initiate conversion
   while (!AD0INT);                    // wait for conversion to complete
   return ADC0;
}


unsigned char avgVin(void)
{
   unsigned char i, result;
   unsigned int sum;
   sum = 0;
   for (i = 64; i != 0; i--)          // repeat 64 times
   {
      sum += readVin();               // read ADC and add to sum
   }
   result = (unsigned char)(sum>>6);  // divide by 64 and cast to uchar
   return result;                     // return average reading
}

void coast(void)
{
    PCA0CPH0 = 0;                     // disable PWM
    P0 = 0xff;                        // force all outputs high
}

void reverse (void)
{
   Fwd=!Fwd;                          // toggle direction flag
   CR = 0;                            // Stop PCA0 timer
   PCA0CPH0 = 0;                      // clear duty cycle
   PCA0CPM0 = 0;                      // disable PWM
   P0 = 0xff;                         // force all outputs high
   XBR2 &= ~0x40;                     // disable Crossbar
   if (Fwd)
   {
      XBR0 = ~0x01;                   // don’t skip P0.0
      P0   = ~0x08;                   // P0.3 low
   }
   else
   {
      XBR0 = ~0x02;                   // don’t skip P0.1
      P0   = ~0x04;                   // P0.2 low
   }
   XBR2 |= 0x40;                      // enable Crossbar
   PCA0CPM0 = 0x42;                   // Module 0, 8-bit PWM Mode
   CR = 1;                            // restart PCA0 timer
}
```

Appendix C—DC Motor with Soft Reversing Code

```
//-----------------------------------------------------------------------------
// Example 3
// DC Motor Control with Soft Reversing
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 12MAR04
//
// This program provides dc motor control with reversing control using the PCA
// 8-bit PWM Mode. A single PCA module is used to generate an 8-bit PWM. The
// pin skip register XBR0 is used to multiplex the PWM between two outputs
// P0.0 and P0.1. These two outputs are used to drive the lower transistors in
// an H-Bridge configuration. P0.2 and P0.3 are used to drive the upper
// transistors.
//
// The ADC is used to read the potentiometer voltage on P0.6. The ADC uses
// polled mode and 64 sample averaging.
//
// The ADC is also used to sense the motor voltage during reversal. When the
// Reverse pushbutton is pressed, the PWM duty will be set to zero. The
// differential voltage across the motor is measured. The once this voltage
// drops below a limit, the upper transistors are reversed and the PWM will be
// applied to the other phase.

//
// Target: C8051F30x
//
// Tool chain: KEIL Eval ‘c’
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f300.h>                  // SFR declarations

//-----------------------------------------------------------------------------
// Typdefs
//-----------------------------------------------------------------------------

typedef union                           // union used for writing to TL0 & TH0
    {
        struct
        {
            unsigned char hi;
            unsigned char lo;
        } b;
        unsigned int w;
    }udblbyte;

//-----------------------------------------------------------------------------
// MACROS
//-----------------------------------------------------------------------------

#define VWINDOW (5*256/100)             // set window to +/- 5%
```

```
#define DTIME (245000/48)               // set delay time to 10 ms per sample
#define GSAMP 10                        // stop for 10 good samples (100ms)

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------

void SYSCLK_Init (void);                // Initialize SYSCLK
void PORT_Init (void);                  // Initialize XBR and Port Pins
void PCA0_Init (void);                  // Initialize PCA0
void ADC0_Init (void);                  // Initialize ADC
void Timer_Init (void);
unsigned char readVin(void);            // read ADC using polling
unsigned char avgVin(void);             // returns avg ADC reading
void coast (void);                      // disable PWM and outputs
void detectStop(void);                  // wait for motor to stop
void reverse (void);
void delay(unsigned int);

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

bit Fwd;
sbit SW1 = P0^7;

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main (void)
{
   PCA0MD &= ~0x40;                     // Disable Watchdog Timer

   SYSCLK_Init ();                      // Initialize system clock
   PORT_Init ();                        // Initialize crossbar and GPIO
   ADC0_Init();                         // Initialize ADC for polled mode
   PCA0_Init ();                        // PCA0 for 8-bit PWM
   Timer_Init();
   Fwd = 1;
   EA = 1;                              // enable global interrupts

   while (1)
   {
      if(SW1)
      {

         PCA0CPH0 = avgVin();           // get avg reading and output to PWM
      }
      else
      {
         coast();
         while(!SW1);                   // wait for button
         detectStop();
         reverse();
      }
   }
}
```

```
//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------

void SYSCLK_Init (void)
{
   OSCICN = 0x07;                      // configure internal oscillator for
   RSTSRC = 0x06;                      // enable missing clock detector
                                       // and VDD Monitor.
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - /Abot - active low PWM signal drives bottom transistor
// P0.1 - /Bbot - active low PWM signal drives bottom transistor
// P0.2 - /Atop - active low signal drives top transistor
// P0.3 - /Abot - active low signal drives top transistor
// P0.4 - VA
// P0.5 - VB
// P0.6 - Vin
// P0.7 - Switch
//

void PORT_Init (void)
{
   XBR0     =  0x70;                    // skip P0.4, P0.5, & P0.6
   XBR1     =  0x40;                    // Enable CEX0 on P0.0
   P0MDOUT  =  0x0f;                    // P0.0 - P0.4 are push-pull outputs
   P0MDIN   = ~0x70;                    // Configure P0.6 for analog input
   XBR2     =  0x40;                    // Enable crossbar
   P0       = ~0x08;                    // P0.3 low

}


//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------

void PCA0_Init (void)
{

   PCA0MD = 0x02;                      // PCA uses sysclk/4, no interrupt

   PCA0CPM0 = 0x42;                    // Module 0, 8-bit PWM Mode

   PCA0L = 0x00;                       // reset the timer
   PCA0H = 0x00;
   PCA0CPL0 = 0x00;
   PCA0CPH0 = 0x00;                    // Initialize to minimum duty
   CR = 1;                             // Start PCA0 timer

}

//-----------------------------------------------------------------------------
// ADC0_Init
```

```
//-----------------------------------------------------------------------------
void ADC0_Init (void)
{
   ADC0CN = 0x40;                    // Low-power tracking mode;
                                     // ADC0 conversions are initiated
                                     // on writes to AD0BUSY;
   AMX0SL = 0xf6;                    // select P0.6, single-ended
   ADC0CF = 0x81 ;                   // AD0SC=4 gain =1
   REF0CN = 0x0a;                    // ADC uses Vdd for full scale
   EIE1 &= ~0x04;                    // disable ADC0 EOC interrupt
   AD0EN = 1;                        // enable ADC
}
//-----------------------------------------------------------------------------
// Timer_Init
//-----------------------------------------------------------------------------
void Timer_Init (void)
{
   CKCON = 0x02;                      // T0 uses sysclk/48
   TMOD =  0x01;                      // T0 mode 1, 16-bit counter
}
//-----------------------------------------------------------------------------
// read_Vin()
//-----------------------------------------------------------------------------
unsigned char readVin(void)
{
   AD0INT = 0;                       // clear ADC0 end-of-conversion
   AD0BUSY = 1;                      // initiate conversion
   while (!AD0INT);                  // wait for conversion to complete
   return ADC0;
}
//-----------------------------------------------------------------------------
// detectStop()
//-----------------------------------------------------------------------------

void detectStop(void)
{
   unsigned char g;

   AMX0SL = 0x54;                    // select P0.4 - P0.5
   g = 0;
   while (g < GSAMP)                 // wait for GSAMP good samples
   {
      AD0INT = 0;                    // clear ADC0 end-of-conversion
      AD0WINT = 0;                   // clear window detector
      AD0BUSY = 1;                   // initiate conversion
      while (!AD0INT);               // wait for conversion to complete
      if(AD0WINT)
         g++;                        // count number of good samples
      else
         g=0;                        // start over if outside window
      delay(DTIME);                  // wait 10 ms
   }

   AMX0SL = 0xf6;                    // select P0.6, single ended
}
//-----------------------------------------------------------------------------
// avgVin()
//-----------------------------------------------------------------------------
```

```
unsigned char avgVin(void)
{
   unsigned char i, result;
   unsigned int sum;

   sum = 0;
   for (i = 64; i != 0; i--)          // repeat 64 times
   {
      sum += readVin();               // read ADC and add to sum
   }
   result = (unsigned char)(sum>>6);  // divide by 64 and cast to uchar
   return result;                     // return average reading
}
//-----------------------------------------------------------------------------
// coast()
//-----------------------------------------------------------------------------

void coast(void)
{
    PCA0CPH0 = 0;                     // disable PWM
    P0 = 0xff;                        // force all outputs high
}
//-----------------------------------------------------------------------------
// reverse()
//-----------------------------------------------------------------------------

void reverse (void)
{
   Fwd=!Fwd;
   CR = 0;                            // Stop PCA0 timer
   PCA0CPH0 = 0;                      // clear duty cycle
   PCA0CPM0 = 0;                      // disable PWM
   P0 = 0xff;                         // all high
   XBR2 &= ~0x40;                     // disable Crossbar
   if (Fwd)
   {
      XBR0 = ~0x01;                   // don't skip P0.0
      P0   = ~0x08;                   // P0.3 low
   }
   else
   {
      XBR0 = ~0x02;                   // don't skip P0.1
      P0   = ~0x04;                   // P0.2 low
   }
   XBR2 |= 0x40;                      // enable Crossbar
   PCA0CPM0 = 0x42;                   // Module 0, 8-bit PWM Mode
   CR = 1;                            // restart PCA0 timer
}

//-----------------------------------------------------------------------------
// delay()
//-----------------------------------------------------------------------------
void delay(unsigned int d)
{
   udblbyte t;
   TR0 = 0;                           // stop Timer0
   t.w = -d;                          // take 2's complement
   TL0 = t.b.lo;                      // write lo byte first
   TH0 = t.b.hi;                      // write hi byte second
   TF0 = 0;                                    // clear overflow flag
   TR0 = 1;                                    // start Timer0
   while(!TF0);                                // wait for overflow
   TR0 = 0;                                    // stop timer
}
```

Appendix D—Brushless DC Motor Code

```
//-----------------------------------------------------------------------------
// Example 4
// BLDC Motor Control
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 12MAR04
//
// This program provides Brushless DC motor control using the PCA 8-bit PWM
// Mode. A single PCA module is used to generate an 8-bit PWM. The pin skip
// register XBR0 is used to multiplex the PWM between three outputs P1.0,
// P1.1, & P1.2 These three outputs are used to drive the lower transistors in
// an 3-Phase Bridge configuration. P0.4, P0.5, & P0.6 are used to drive the
// upper transistors.
//
// The ADC is used to read the potentiometer voltage on P0.6. The ADC uses
// polled mode and 64 sample averaging.
//
// P0.0, P0.1, and P0.2 are used for hall effect sensor inputs. This pins are
// polled to determine the rotor position. The readHalls() function requires
// three identical samples and returns the hall code. The corresponding state
// of the motor is found from the HallPattern. This state is then used to
// commutate the motor.
//
// It is safe to single-step and use breakpoints ONLY with the motor wires
// disconnected. Do not single step through the code with the motor wires
// connected! The PWM outputs may remain active while the CPU is stopped.
// In particular do not single step past the lines that enable the PWM.
// These lines are marked with comments in ALL CAPS.
//
// Target: C8051F33x
//
// Tool chain: KEIL Eval ‘c’
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f330.h>                 // SFR declarations
//-----------------------------------------------------------------------------
// MACROS
//-----------------------------------------------------------------------------

#define GSAMP 3

//-----------------------------------------------------------------------------
// Hall-effect and commutation patterns
//-----------------------------------------------------------------------------

const unsigned char code hallPattern[7]=
    { 0x00, 0x01, 0x03, 0x02, 0x06, 0x04, 0x05};

const unsigned char code skipPattern[7]=
    {~0x01,~0x01,~0x01,~0x02,~0x02,~0x04,~0x04};
```

```
const unsigned char code P1Pattern[7]=
    {~0x00,~0x20,~0x40,~0x40,~0x10,~0x10,~0x20};


//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void SYSCLK_Init (void);
void PORT_Init (void);
void PCA0_Init (void);
void PCA0_ISR (void);
void ADC_Init (void);
unsigned char readVin(void);
unsigned char avgVin(void);
unsigned char readHalls(void);
unsigned char hallPosition(void);
void commutate(unsigned char);
void coast(void);

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
void main (void) {

   unsigned char h,p;
   bit start;

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

   SYSCLK_Init ();                     // initialize system clock


   PORT_Init ();                       // initialize i/o
   PCA0_Init ();                       // configure PCA0 to 8-bit PWM
   ADC_Init();                         // initialize i/o

   EA = 1;                             // enable global interrupts

   p = 0;                              // clear p
   start = 1;                          // set start bit

   while (1)
   {
      h = hallPosition();              // h equals hall position
      if(h)                            // if good position
      {
         if ((h != p)||(start))        // if new position or start
         {
            p = h;                     // update p
                                       // DO NOT SINGLE-STEP PAST THE NEXT
                                       // LINE WITH MOTOR WIRES CONNECTED!!!
            commutate(p);              // commutate motor, enables PWM
            start = 0;
         }
         PCA0CPH0 = avgVin();          // get avg reading and output to PWM
      }
      else
      {
         coast();                     // coast until good reading
         start = 1;                   // set start bit to restart motor
      }
   }

}

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

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

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - HA
// P0.1 - HB
// P0.2 - HC
// P0.3 -
// P0.4 -
// P0.5 -
// P0.6 -
// P0.7 - Vin - analog input
//
// P1.0 - Abottom - push-pull output
// P1.1 - Bbot - push-pull output
// P1.2 - Cbot - push-pull output
// P1.3 -
// P1.4 - Atop - push-pull output
// P1.5 - Btop - push-pull output
// P1.6 - Ctop - push-pull output
// P1.7 -
//
void PORT_Init (void)
{
   XBR0     = 0x00;                   // enable nothing on XBR0
   XBR1     = 0x01;                   // enable PCA CEX0
   P0SKIP   = 0xFF;                   // skip all pins on P0
   P0MDIN   =~0x80;                   // P0.7 analog input
   P1SKIP   =~0x01;                   // skip all except P1.0
   P1MDOUT  = 0x77;                   // enable P1 outputs
   XBR1    |= 0x40;                   // enable crossbar
   P1       = 0xff;                   // P1 all high
}


//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------
void PCA0_Init (void)
{

   PCA0MD = 0x02;                     // PCA uses sysclk/4, no CF int
   PCA0CPM0 = 0x00;                    // clear mode, pin high
   PCA0L = 0x00;                       // reset the timer
   PCA0H = 0x00;
   PCA0CPH0 = 0x00;                    // initial to 0%
   CR = 1;                             // START PCA0 timer
}

//-----------------------------------------------------------------------------
// coast function
//-----------------------------------------------------------------------------

void coast(void)
{
   PCA0CPM0 = 0x00;                    // disable PWM
   P1 = 0xff;                          // disable upper transistors
}

//-----------------------------------------------------------------------------
// readHalls function
//-----------------------------------------------------------------------------
// reads and debounces Hall-Sensor inputs

unsigned char readHalls(void)
{
   unsigned char g,h,i;
   g = 0;
   h = 0;
   while (g<GSAMP)                     // while less that 3 good samples
   {
     i = P0 & 0x07;                    // read halls
     if (h == i)                       // if the same
         g++;                          // one more good
     else
         g = 0;                        // else start over
     h = i;                            // update h
   }
   return h;                           // return good hall code
}

//-----------------------------------------------------------------------------
// hallPosition function
//-----------------------------------------------------------------------------

unsigned char hallPosition (void)
{
   unsigned char h,p;
   h = readHalls();                    // get debounced hall reading

   // find corresponding pattern index
   for (p=6;(h != hallPattern[p])&&(p!=0);p--);
   return p;
}

//-----------------------------------------------------------------------------
// hallPosition function
//-----------------------------------------------------------------------------

void commutate (unsigned char i)
{
   PCA0CPM0 = 0x00;                    // disable PWM
   P1 = 0xFF;
   XBR1 &= ~0x40;                      // disable crossbar
   P1SKIP = skipPattern[i];
   P1 = P1Pattern[i];
   XBR1 |= 0x40;                       // enable crossbar
                                       // DO NOT SINGLE-STEP PAST THE NEXT
                                       // LINE WITH MOTOR WIRES CONNECTED!!!
   PCA0CPM0 = 0x42;                    // enable 8-bit PWM mode
}

//-----------------------------------------------------------------------------
// ADC functions
//-----------------------------------------------------------------------------

void ADC_Init(void)
{
    AMX0P       = 0x07;                // positive input P0.7
    AMX0N       = 0x11;                // single ended mode
    ADC0CF      = 0xC4;                // 1MHz clock, left justified
    ADC0CN      = 0x80;                // configure ADC for polled mode
    REF0CN      = 0x08;                // use Vdd as ADC full scale

}

unsigned char readVin(void)
{
   AD0INT = 0;                         // clear ADC0 end-of-conversion
   AD0BUSY = 1;                        // initiate conversion
   while (!AD0INT);                    // wait for conversion to complete
   return ADC0H;
}

unsigned char avgVin(void)
{
   unsigned char i, result;
   unsigned int sum;

   sum = 0;
   for (i = 64; i != 0; i--)           // repeat 64 times
   {
      sum += readVin();                // read ADC and add to sum
   }
   result = (unsigned char)(sum>>6);   // divide by 64 and cast to uchar
   return result;                      // return average reading
}
```

Appendix E—AC Induction Motor Code

```
//-----------------------------------------------------------------------------
// Example 5
// AC Motor Control
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 12MAR04
//
// This program provides AC Induction motor control using the PCA 8-bit
// PWM mode. Three PCA channels are used to generate three-phase PWM.
// The three PWMs are output on P0.0-0.2. The falling edge of the three
// PWM signals are edge-aligned. Each PWM has a frequency of 24 kHz and
// a low-time that varies independently from 160ns to 100%. These
// signals can be used to drive a induction motor using a three-phase
// bridge. A gate drive with built in dead-time is required.
//
// A potentiometer is connected to P0.6. The value of the Pot controls
// the speed of the motor. The speed is controlled using a constant V/
// Hz profile. The sine wave frequency varies from DC up to 60 Hz. The
// modulation depth varies from zero to 100%.
//
// Target: C8051F33x
//
// Tool chain: KEIL Eval ‘c’
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f300.h>                  // SFR declarations
//-----------------------------------------------------------------------------
// Macros
//-----------------------------------------------------------------------------
#define DTIME 5981                      // delay time 1 ms update rate
                                        // 24500000/4/1024
//-----------------------------------------------------------------------------
// Typdefs
//-----------------------------------------------------------------------------
typedef union                           // union used for writing to TL0 & TH0
    {
        struct
        {
            unsigned char hi;
            unsigned char lo;
        } b;
        unsigned int w;
    }udblbyte;

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void SYSCLK_Init (void);
void PORT_Init (void);
void PCA0_Init (void);
void ADC0_Init (void);
void Timer_Init(void);
unsigned char readVin(void);
unsigned char avgVin(void);
unsigned char sineWave(unsigned char);
void Timer_ISR (void);
void Update(void);
//-----------------------------------------------------------------------------
// Sine Wave Table
//-----------------------------------------------------------------------------

const signed char code sine[256]=
{
   0x00, 0x03, 0x06, 0x09, 0x0C, 0x10, 0x13, 0x16, 0x19, 0x1C, 0x1F, 0x22,
   0x25, 0x28, 0x2B, 0x2E, 0x31, 0x34, 0x36, 0x39, 0x3C, 0x3F, 0x42, 0x44,
   0x47, 0x49, 0x4C, 0x4E, 0x51, 0x53, 0x56, 0x58, 0x5A, 0x5C, 0x5E, 0x60,
   0x62, 0x64, 0x66, 0x68, 0x6A, 0x6C, 0x6D, 0x6F, 0x70, 0x72, 0x73, 0x74,
   0x76, 0x77, 0x78, 0x79, 0x7A, 0x7B, 0x7B, 0x7C, 0x7D, 0x7D, 0x7E, 0x7E,
   0x7E, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7E, 0x7E, 0x7D, 0x7D,
   0x7C, 0x7C, 0x7B, 0x7A, 0x79, 0x78, 0x77, 0x76, 0x75, 0x74, 0x72, 0x71,
   0x70, 0x6E, 0x6C, 0x6B, 0x69, 0x67, 0x65, 0x63, 0x61, 0x5F, 0x5D, 0x5B,
   0x59, 0x57, 0x54, 0x52, 0x50, 0x4D, 0x4B, 0x48, 0x45, 0x43, 0x40, 0x3D,
   0x3B, 0x38, 0x35, 0x32, 0x2F, 0x2C, 0x29, 0x27, 0x24, 0x20, 0x1D, 0x1A,
   0x17, 0x14, 0x11, 0x0E, 0x0B, 0x08, 0x05, 0x02, 0xFE, 0xFB, 0xF8, 0xF5,
   0xF2, 0xEF, 0xEC, 0xE9, 0xE6, 0xE3, 0xE0, 0xDC, 0xD9, 0xD7, 0xD4, 0xD1,
   0xCE, 0xCB, 0xC8, 0xC5, 0xC3, 0xC0, 0xBD, 0xBB, 0xB8, 0xB5, 0xB3, 0xB0,
   0xAE, 0xAC, 0xA9, 0xA7, 0xA5, 0xA3, 0xA1, 0x9F, 0x9D, 0x9B, 0x99, 0x97,
   0x95, 0x94, 0x92, 0x90, 0x8F, 0x8E, 0x8C, 0x8B, 0x8A, 0x89, 0x88, 0x87,
   0x86, 0x85, 0x84, 0x84, 0x83, 0x83, 0x82, 0x82, 0x81, 0x81, 0x81, 0x81,
   0x81, 0x81, 0x81, 0x82, 0x82, 0x82, 0x83, 0x83, 0x84, 0x85, 0x85, 0x86,
   0x87, 0x88, 0x89, 0x8A, 0x8C, 0x8D, 0x8E, 0x90, 0x91, 0x93, 0x94, 0x96,
   0x98, 0x9A, 0x9C, 0x9E, 0xA0, 0xA2, 0xA4, 0xA6, 0xA8, 0xAA, 0xAD, 0xAF,
   0xB2, 0xB4, 0xB7, 0xB9, 0xBC, 0xBE, 0xC1, 0xC4, 0xC7, 0xCA, 0xCC, 0xCF,
   0xD2, 0xD5, 0xD8, 0xDB, 0xDE, 0xE1, 0xE4, 0xE7, 0xEA, 0xED, 0xF0, 0xF4,
   0xF7, 0xFA, 0xFD, 0x00
};

//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------
unsigned char Volts;                  // output voltage

unsigned int Sum;                     // integral of omega

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------


void main (void)
{

   PCA0MD &= ~0x40;                   // disable watchdog timer

   SYSCLK_Init ();                    // initialize system clock
   PCA0_Init ();                      // initialize PCA0 for 8-bit PWM
   PORT_Init ();                      // initialize crossbar and GPIO
   ADC0_Init();                       // initialize ADC for polled mode
   Timer_Init();                      // initialize T0 for update timebase
   EA = 1;                          // enable global interrupts

   Sum = 0;                         // clear 16-bit integral of omega

   while (1)
   {

      Volts = avgVin();             // set output voltage to pot setting
   }
}


//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
void SYSCLK_Init (void)
{
   OSCICN = 0x07;                   // configure for 24.5 MHz
   RSTSRC = 0x06;                   // enable missing clock detector
                                    // and VDD Monitor.
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - !PWMA - phase A PWM signal - push-pull output
// P0.1 - !PWMB - phase B PWM signal - push-pull output
// P0.2 - !PWMC - phase C PWM signal - push-pull output
// P0.3 -
// P0.4 -
// P0.5 -
// P0.6 - Analog Input
// P0.7 -
//
void PORT_Init (void)
{
   XBR0    =  0x00;                 // skip nothing
   XBR1    =  0xc0;                 // enable CEX0-2
   P0MDOUT =  0x07;                 // P0.0-2 push-pull output
   P0MDIN  = ~0x40;                 // P0.6 analog input
   XBR2    =  0x40;                 // enable crossbar
}

//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------

void PCA0_Init (void)
{

    PCA0MD = 0x02;                  // use SYSCLK/4 for 24kHz PWM
    PCA0L  = 0x00;                  // clear PCA Counter/Timer Low Byte
    PCA0H  = 0x00;                  // clear PCA Counter/Timer High Byte


    //Module 0
    PCA0CPM0 = 0x42;                // module 0 8-bit PWM no interrupts
    PCA0CPL0 = 0x80;                   // initialize for 50%
    PCA0CPH0 = 0x80;                   // initialize for 50%

    //Module 1
    PCA0CPM1 = 0x42;                   // module 1 8-bit PWM no interrupts
    PCA0CPL1 = 0x80;                   // initialize for 50%
    PCA0CPH1 = 0x80;                   // initialize for 50%

    //Module 2
    PCA0CPM2 = 0x42;                   // module 2 8-bit PWM no interrupts
    PCA0CPL2 = 0x80;                   // initialize for 50%
    PCA0CPH2 = 0x80;                   // initialize for 50%

    PCA0CN   = 0x40;                   // enable PCA0

}



//-----------------------------------------------------------------------------
// Timer_Init
//-----------------------------------------------------------------------------

void Timer_Init (void)
{
   CKCON = 0x01;                       // T0 uses sysclk/4
   TMOD =  0x01;                       // T0 mode 1
   TR0 = 1;                            // enable timer
   ET0 = 1;                            // enable interrupts
   TF0 = 1;                            //force interrupt

}

//-----------------------------------------------------------------------------
// Timer_ISR
//-----------------------------------------------------------------------------

void Timer_ISR(void) interrupt 1
{
   udblbyte t;
   TR0 = 0;                            // stop Timer0
   t.w =  -DTIME;                      // 2s complement delay-time
   TL0 = t.b.lo;                       // write lo byte first
   TH0 = t.b.hi;                       // write hi byte second
   TF0 = 0;                            // clear overflow flag
   TR0 = 1;                            // start Timer0
   Update();                           // update sinewave
}

//-----------------------------------------------------------------------------
// Update function
//-----------------------------------------------------------------------------

void Update (void)
{
      unsigned int omega;              // angular frequency
      unsigned char theta;             // sine wave angle

      omega = Volts;                   // constant V/Hz control
      omega <<=4;                     // scale omega for 1 ms update rate

      Sum += omega;                   // integrate omega

      theta = Sum >>8;                // theta is upper byte

      PCA0CPH0 = sineWave(theta);     // output sinewaves

      PCA0CPH1 = sineWave(theta + 0x55);

      PCA0CPH2 = sineWave(theta + 0xaa);
}

//-----------------------------------------------------------------------------
// sineWave function
//-----------------------------------------------------------------------------

unsigned char sineWave(unsigned char q)
{
      signed char s;                  // signed sine
      unsigned char o;                // output value
      unsigned int p;                 // 16 bit product

      s = sine[q];                    // get value from table

      p = Volts * (signed int)s;      // multiply by v

      o = p>>8;                       // throw away low byte

      o += 0x80;                      // center sinewave at 50%

      return o;                       // return sinewave value
}

//-----------------------------------------------------------------------------
// ADC functions
//-----------------------------------------------------------------------------
void ADC0_Init (void)
{
   ADC0CN = 0x40;                     // Low-power tracking mode;
                                      // ADC0 conversions are initiated
                                      // on writes to AD0BUSY;
   AMX0SL = 0xf6;                     // select P0.6, single-ended
   ADC0CF = 0x81 ;                    // AD0SC=4 gain =1
   REF0CN = 0x0a;                     // ADC uses Vdd for full scale
   EIE1 &= ~0x04;                     // disable ADC0 EOC interrupt
   AD0EN = 1;                         // enable ADC
}

unsigned char readVin(void)
{
   AD0INT = 0;                        // clear ADC0 end-of-conversion
   AD0BUSY = 1;                       // initiate conversion
   while (!AD0INT);                   // wait for conversion to complete
   return ADC0;
}

unsigned char avgVin(void)
```

```
{
   unsigned char i, result;
   unsigned int sum;

   sum = 0;
   for (i = 64; i != 0; i--)           // repeat 64 times
   {
      sum += readVin();                // read ADC and add to sum
   }
   result = (unsigned char)(sum>>6);   // divide by 64 and cast to uchar
   return result;                      // return average reading
}
```

Appendix F—PWM using High-Speed Output Mode Code

```
//-----------------------------------------------------------------------------
// Example 6
// PWM using PCA High Speed Output Mode
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 12MAR04
//
// This example demonstrates using the High Speed Output (HSO) mode to
// generate a PWM signal.
//
// The program reads the value of a potentiometer connected to P0.6 and uses
// this value to set the duty Cycle of the PWM. This example could be used
// to drive a dc motor.
//
// Target: C8051F30x
//
// Tool chain: KEIL Eval 'c'
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f300.h>                  // SFR declarations

//-----------------------------------------------------------------------------
// typedefs
//-----------------------------------------------------------------------------

typedef union                           // union used for writing to TL0 & TH0
    {
        struct
        {
            unsigned char hi;
            unsigned char lo;
        } b;
        unsigned int w;
    }udblbyte;

//-----------------------------------------------------------------------------
// MACROS (all caps)
//-----------------------------------------------------------------------------

#define SYSCLK       24500000           // SYSCLK frequency in Hz
#define PERIOD (SYSCLK/20000)
#define DEADTIME 25                     // desired Dead-Time in clocks
#define LATENCY 45                      // worst case latency in clocks
#define HTSPAN (PERIOD - 2*LATENCY)     // high-time span
//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------

unsigned int HiTime;
unsigned int  NextEdge;
```

```
bit Cycle;
//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void SYSCLK_Init (void);
void PORT_Init (void);
void PCA0_Init (void);
void PCA0_ISR (void);
void ADC0_Init (void);
unsigned char readVin (void);
unsigned char avgVin (void);

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
void main (void) {

   unsigned long x;
   unsigned int y;



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

   SYSCLK_Init ();                     // Initialize system clock to
                                       // 24.5MHz internal oscillator
   PORT_Init ();                       // Initialize crossbar and GPIO
   PCA0_Init ();
   ADC0_Init();

   EA = 1;                             // enable global interrupts

   while (1)
   {
      x = avgVin();                    // get avg. ADC reading
      x *= HTSPAN;                     // multiply by span
      y = x>>8;                        // through away low byte
      y += LATENCY;                    // add minimum latency
      EIE1 &= ~0x08;                   // disable PCA interrupt
      HiTime =  y;                     // coherent update of global hitime
      EIE1    |= 0x08;                 // enable PCA interrupt
   }
}

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

void SYSCLK_Init (void)
{
   OSCICN = 0x07;                      // configure for 24.5 MHz
   RSTSRC = 0x04;                      // enable missing clock detector
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - PWM - CEX0 - push-pull output
// P0.1 -
// P0.2 -
// P0.3 -
// P0.4 -
// P0.5 -
// P0.6 - Analog Input
// P0.7 -
//
void PORT_Init (void)
{
   XBR0     =  0x00;                   // skip nothing
   XBR1     =  0x40;                   // Enable CEX0 on P0.0
   P0MDOUT  =  0x01;                   // enable CEX0 as a push-pull output
   P0MDIN   = ~0x40;                   // enable ADC input on P0.6
   XBR2    |=  0x40;                   // enable crossbar

}

//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------

void PCA0_Init (void)
{
   udblbyte output;

   PCA0MD   = 0x08;                    // PCA uses sysclk, no CF int
   PCA0CPM0 = 0x4D;                    // High Speed Output Mode, enable ECCF0
   PCA0L    = 0x00;                    // reset the timer
   PCA0H    = 0x00;
   output.w = PERIOD/2;                // schedule first interrupt
   PCA0CPL0 = output.b.lo;
   PCA0CPH0 = output.b.hi;
   NextEdge = PERIOD;                  // initialize NextEdge
   EIE1    |= 0x08;                    // enable PCA0 interrupts
   EIP1    |= 0x08;                    // set PCA to high priority

   CR = 1;                             // start PCA0 timer
}

//-----------------------------------------------------------------------------
// PCA0_ISR
//-----------------------------------------------------------------------------
void PCA0_ISR (void) interrupt 9 using 1
{
   static bit cycle = 0;
   udblbyte output;

   output.w = NextEdge;
   PCA0CPL0 = output.b.lo;             // write lo byte first
   PCA0CPH0 = output.b.hi;             // write hi byte second
   PCA0CN &= ~0x87;                    // clear all PCA flags
   if (cycle)
      {
      NextEdge += PERIOD;
      NextEdge -= HiTime;
      }
   else
   {
      NextEdge += HiTime;
   }
   cycle = !cycle;
}


//-----------------------------------------------------------------------------
// ADC functions
//-----------------------------------------------------------------------------

void ADC0_Init (void)
{
   ADC0CN = 0x00;                      // normal tracking mode;
                                       // ADC0 conversions are initiated
                                       // on writes to AD0BUSY;
   AMX0SL =  0xf6;                     // select P0.6
   ADC0CF =  0x81 ;                    // AD0SC=4 gain =1
   REF0CN =  0x0a;                     // ADC uses Vdd for full scale
   EIE1  &= ~0x04;                     // disable ADC0 EOC interrupt
   AD0EN   =  1;                       // enable ADC
}

unsigned char readVin(void)
{
   AD0INT = 0;                         // clear ADC0 end-of-conversion
   AD0BUSY = 1;                        // initiate conversion
   while (!AD0INT);                    // wait for conversion to complete
   return ADC0;
}

unsigned char avgVin(void)
{
   unsigned char i, result;
   unsigned int sum;

   sum = 0;
   for (i = 64; i != 0; i--)           // repeat 64 times
   {
      sum += readVin();                // read ADC and add to sum
   }
   result = (unsigned char)(sum>>6);   // divide by 64 and cast to uchar
   return result;                      // return average reading
}
```

Appendix G—Center Aligned PWM Code

```
//-----------------------------------------------------------------------------
// Example 7
// Center-Aligned PWM with Dead-Time
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 12MAR04
//
// This program reads the voltage at P0.6 and outputs center-aligned PWM
// with dead-time on P0.1 and P0.2.
//
// Target: C8051F30x
//
// Tool chain: KEIL Eval ‘c’
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f300.h>                  // sfr declarations

typedef union                           // union used for writing to PCA0CPxx
    {
        struct
        {
            unsigned char hi;
            unsigned char lo;
        } b;
        unsigned int w;
    }udblbyte;
//-----------------------------------------------------------------------------
// Macros
//-----------------------------------------------------------------------------

#define SYSCLK       24500000           // SYSCLK frequency in Hz
#define PERIOD (SYSCLK/20000/2)
#define DEADTIME 25                     // desired dead-time in clocks
#define LATENCY 45                      // worst case latency in clocks
#define HTSPAN (PERIOD - 2*LATENCY)     // high-time span

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

unsigned int HiTime;                    // global PWM high-time
unsigned int nextEdge0;                 // CEX0 next edge time
unsigned int nextEdge1;                 // CEX1 next edge time
unsigned int nextEdge2;                 // CEX2 next edge time

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------

void SYSCLK_Init (void);
void PORT_Init (void);
void PCA0_Init (void);
void PCA0_ISR (void);
void ADC0_Init (void);
unsigned char readVin(void);
unsigned char avgVin(void);

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main (void)
{

   unsigned long x;
   unsigned int y;



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

  SYSCLK_Init ();                // Initialize system clock to
// 24.5MHz internal oscillator
   PORT_Init ();                       // Initialize crossbar and GPIO
   PCA0_Init ();
   ADC0_Init();

   EA = 1;                             // enable global interrupts

   while (1)
   {
      x = avgVin();                    // get avg. ADC reading
      x *= HTSPAN;                     // multiply by span
      y = x>>8;                        // through away low byte
      y += LATENCY;                    // add minimum latency
      EIE1    &= ~0x08;                // disable interrupt while updating
      HiTime =  y;                     // coherent update of global hitime
      EIE1    |= 0x08;                 // re-enable interrupt
   }
}

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

void SYSCLK_Init (void)
{
   OSCICN = 0x07;                      // 24.5MHz internal oscillator
   RSTSRC = 0x04;                      // enable missing clock detector
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - CEX0
// P0.1 - CEX1
// P0.2 - CEX2
// P0.3 -
// P0.4 -
// P0.5 -
// P0.6 - Analog Input
// P0.7 - C2D
//

void PORT_Init (void)
{
   XBR0     =  0x00;                  // skip nothing
   XBR1     =  0xC0;                  // Enable CEX0-2 on P0.0-2
   P0MDOUT |=  0x07;                  // enable CEX0-2 as a push-pull output
   P0MDIN   = ~0x40;                  // configure P0.6 for analog input
   XBR2     =  0x40;                  // Enable crossbar

}

//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------

void PCA0_Init (void)
{
   udblbyte output;                   // used to write to PCA0CPxx

   PCA0MD = 0x08;                     // use system clock
   PCA0CN = 0x00;                     // clear PCA control register

   //Module 0
   PCA0CPM0 = 0x4D;                   // HSO mode, enable interrupts
   output.w = PERIOD;                 // schedule first low-high transition
   PCA0CPL0 = output.b.lo;            // for one full period
   PCA0CPH0 = output.b.hi;

   //Module 1
   PCA0CPM1 = 0x4C;                   // HSO mode, disable interrupts
   output.w = (PERIOD/2+DEADTIME);    // schedule first low-high transition
   PCA0CPL1 = output.b.lo;            // before CEX0 sets polarity to
   PCA0CPH1 = output.b.hi;            // low on CEX0 H-L transition

   //Module 2
   PCA0CPM2 = 0x4C;                   // HSO mode, disable interrupts
   output.w = (2*PERIOD);             // schedule first low-high transition
   PCA0CPL2 = output.b.lo;            // after CEX0 sets polarity to
   PCA0CPH2 = output.b.hi;            // high on CEX0 H-L transition


   HiTime = PERIOD/2;                 // init HighTime to 50%
   nextEdge0 = (2*PERIOD);            // init next CEX0 to 50
   nextEdge1 = (3*PERIOD/2-DEADTIME); // init next CEX1 and subtract DT
   nextEdge2 = (3*PERIOD/2+DEADTIME); // init next CEX1 and add DT

   PCA0L  = 0x00;                     // clear PCA Low Byte
   PCA0H  = 0x00;                     // clear PCA High Byte

   EIP1 |= 0x08;                      // set PCA to high priority
   EIE1 |= 0x08;                      // Enable PCA0, interrupts

   CR = 1;                            // start PCA0 timer
```

```
}

//-----------------------------------------------------------------------------
// PCA0_ISR
//-----------------------------------------------------------------------------

void PCA0_ISR (void) interrupt 9 using 1
{
   static bit cycle = 0;

   udblbyte output;
   unsigned int t ;

   output.w = nextEdge1;                 // output next edge on CEX1
   PCA0CPL1 = output.b.lo;
   PCA0CPH1 = output.b.hi;
   output.w = nextEdge2;                 // output next edge on CEX2
   PCA0CPL2 = output.b.lo;
   PCA0CPH2 = output.b.hi;
   output.w = nextEdge0;                 // output next edge on CEX0
   PCA0CPL0 = output.b.lo;
   PCA0CPH0 = output.b.hi;
   PCA0CN &= ~0x87;                      // clear all PCA flags

   cycle = !cycle;                       // toggle Cycle
   if (cycle)
   {
      nextEdge0 += PERIOD;               // pre-increment nextEdge0
      t = nextEdge0 - HiTime;            // calculate next edges
      nextEdge1 = t + (LATENCY/2 + DEADTIME);
      nextEdge2 = t + (LATENCY/2 - DEADTIME);
   }
   else
   {
      t = nextEdge0 + HiTime;            // calculate next edges
      nextEdge1 = t + (LATENCY/2 - DEADTIME);
      nextEdge2 = t + (LATENCY/2 + DEADTIME);
      nextEdge0 += PERIOD;               // post increment nextEdge0
   }
}

//-----------------------------------------------------------------------------
// ADC functions
//-----------------------------------------------------------------------------

void ADC0_Init (void)
{
   ADC0CN = 0x00;                        // use polled mode
   AMX0SL = 0xf6;                        // select P0.6, single ended
   ADC0CF = 0x81 ;                       // AD0SC=4, gain =1
   REF0CN = 0x0a;                        // ADC uses Vdd for full scale
   EIE1 &= ~0x04;                        // disable ADC0 EOC interrupt
   AD0EN = 1;                            // enable ADC
}

unsigned char readVin(void)
{
   AD0INT = 0;                           // clear ADC0 end-of-conversion
   AD0BUSY = 1;                          // initiate conversion
   while (!AD0INT);                     // wait for conversion to complete
   return ADC0;                         // return reading
}

unsigned char avgVin(void)
{
   unsigned char i, result;
   unsigned int sum;

   sum = 0;
   for (i = 64; i != 0; i--)            // repeat 64 times
   {
      sum += readVin();                 // read ADC and add to sum
   }
   result = (unsigned char)(sum>>6);    // divide by 64 and cast to uchar
   return result;                       // return average reading
}
```

Appendix H—Quadrature Decode Code

```
//-----------------------------------------------------------------------------
// Example 8
// Quadrature Decode
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories Inc.
//
// AUTH: KAB
// DATE: 12MAR04
//
// This example provides a quadrature decode interface using the
// external interrupts INT0 and INT1. The external interrupts are
// assigned to P0.0 and P0.1.
//
// The interrupt service routines first complement the appropriate
// interrupt polarity bit and then increment or decrement the global
// variable Position as needed. A nested if...else statement is used
// to determine the appropriate action depending on the status of
// the  polarity bits.
//
// The Position variable is output on the UART each time the Position
// changes. The Position can be monitored using a standard Terminal
// application such as Hyperterm. The Terminal should be configured
// for 9600 bps.
//
// Target: C8051F30x
//
// Tool chain: KEIL Eval ‘c’
//
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f300.h>                    // sfr declarations
#include <stdio.h>                        // printf()
//-----------------------------------------------------------------------------
// Macros
//-----------------------------------------------------------------------------

#define SYSCLK       24500000            // sysclk frequency in Hz
#define BAUDRATE     9600                // baud rate of UART in bps

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------

void main (void);
void SYSCLK_Init (void);
void PORT_Init (void);
void UART0_Init (void);
void EINT_Init (void);
void INT0_ISR (void);
void INT1_ISR (void);

//-----------------------------------------------------------------------------
```

```
// Global VARIABLES
//-----------------------------------------------------------------------------
unsigned int Position;                  // position of quadrature encoder
sbit CHA = P0^0;                        // quadrature channel A pin state
sbit CHB = P0^1;                        // quadrature channel B pin state
//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main (void)
{

   unsigned int p;                      // last position

   PCA0MD = 0x00;                       // disable watchdog timer

   PORT_Init ();                        // initialize GPIO and Crossbar
   SYSCLK_Init ();                      // initialize System Clock
   EINT_Init();                         // initialize External Interrupt
   UART0_Init ();                       // initialize UART

   Position = 0;                        // clear initial position

   EA = 1;                              // enable global interrupts

   while (1)
   {
      while(p == Position);             // do nothing until moved

      p = Position;                     // update p

      printf("%u     \r", p);           // display position
   }
}

//-----------------------------------------------------------------------------
// Initialization Subroutines
//-----------------------------------------------------------------------------

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

void SYSCLK_Init (void)
{
   OSCICN = 0x07;                       // configure for 24.5 MHz
   RSTSRC = 0x04;                       // enable missing clock detector
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - INT0 - CHA input
// P0.1 - INT1 - CHB input
// P0.2 -
// P0.3 -
// P0.4 - UART TX - push-pull output
// P0.5 - UART RX
// P0.6 -
// P0.7 - C2D
//
void PORT_Init (void)
{
   XBR0    =  0x03;                    // skip P0.0 & P0.1
   XBR1    =  0x03;                    // enable uart
   P0MDOUT =  0x10;                    // TX output
   XBR2   |=  0x40;                    // enable crossbars

}

//-----------------------------------------------------------------------------
// EINT_Init
//-----------------------------------------------------------------------------
void EINT_Init (void)
{
    IT01CF  =  0x10;                   // INT0=P0.0, INT1=P0.1
    if(CHA==0)                         // check CHA state
      IT01CF|=0x08;                    // if low, trigger high
    if(CHB==0)                         // check CHB state
      IT01CF|=0x80;                    // if low, trigger high
    TCON   |=  0x05;                   // set IT0 & IT1 for edge trigger
    TCON   &= ~0x0a;                   // clear IE0 and IE1 flags
    IP     |=  0x05;                   // INT0 and INT1 high priority
    IE     |=  0x05;                   // Enable INT0 and INT1

 }

//-----------------------------------------------------------------------------
// INT0_ISR, INT1_ISR
//-----------------------------------------------------------------------------
//
// These two interrupt service routines occur when the respective encoder
// channel changes state. The interrupt service routines are identical
// except for the respective change in the interrupt flag, polarity bit, and
// count direction. The nested if...else statement performs the logical
// equivalent of an exclusive OR function using the polarity bits as the
// quadrature state.
//
void INT0_ISR (void) interrupt 0
{

   IT01CF ^= 0x08;                     // toggle edge select trigger


   if ((IT01CF&0x08)==0x08)
      if((IT01CF&0x80)==0x80)
         Position--;                   // if both decrement
      else
         Position++;                   // if different increment
   else
      if((IT01CF&0x80)==0x80)
         Position++;                   // if different increment
      else
         Position--;                   // if neither decrement
}
```

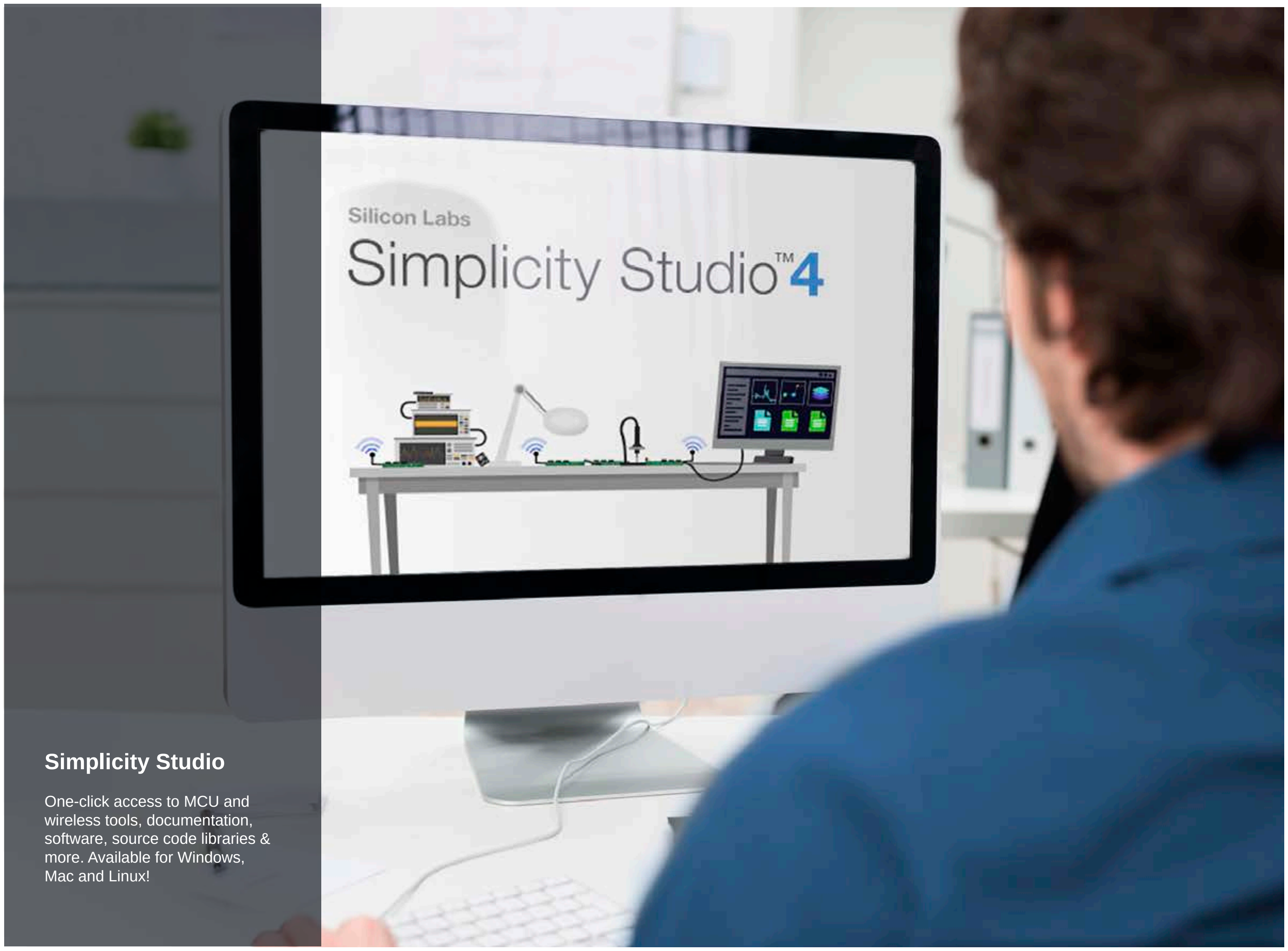
```
void INT1_ISR (void) interrupt 2
{

   IT01CF ^= 0x80;                    // toggle edge  select trigger

   if ((IT01CF&0x08)==0x08)
      if((IT01CF&0x80)==0x80)
         Position++;                  // if both increment
      else
         Position--;                  // if different decrement
   else
      if((IT01CF&0x80)==0x80)
         Position--;                  // if different decrement
      else
         Position++;                  // if neither increment
}

//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------
//
// Standard UART0 Init from software examples
//
void UART0_Init (void)
{
   SCON0 = 0x10;                      // SCON0: 8-bit variable bit rate
                                      //        level of STOP bit is ignored
                                      //        RX enabled
                                      //        ninth bits are zeros
                                      //        clear RI0 and TI0 bits
   if (SYSCLK/BAUDRATE/2/256 < 1) {
      TH1 = -(SYSCLK/BAUDRATE/2);
      CKCON |=  0x10;                 // T1M = 1; SCA1:0 = xx
   } else if (SYSCLK/BAUDRATE/2/256 < 4) {
      TH1 = -(SYSCLK/BAUDRATE/2/4);
      CKCON &= ~0x13;                 // T1M = 0; SCA1:0 = 01
      CKCON |=  0x01;
   } else if (SYSCLK/BAUDRATE/2/256 < 12) {
      TH1 = -(SYSCLK/BAUDRATE/2/12);
      CKCON &= ~0x13;                 // T1M = 0; SCA1:0 = 00
   } else {
      TH1 = -(SYSCLK/BAUDRATE/2/48);
      CKCON &= ~0x13;                 // T1M = 0; SCA1:0 = 10
      CKCON |=  0x02;
   }

   TL1 = TH1;                         // init Timer1
   TMOD &= ~0xf0;                     // TMOD: timer 1 in 8-bit auto-reload
   TMOD |=  0x20;
   TR1 = 1;                           // START Timer1
   TI0 = 1;                           // Indicate TX0 ready
}
```



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