STEPPER MOTOR REFERENCE DESIGN

1. Introduction

Stepper motors are used in a wide variety of applications. They are prevalent in consumer office equipment such as printers, plotters, copiers, and scanners. Stepper motors are also used in automotive applications for electronic throttle control, dashboard indicators, and climate control systems. Stepper motors are also found in industrial equipment such as robotics, electronic component handlers, testers, dispensers, and other manufacturing equipment.

Stepper motors are often controlled using special function ICs that provide limited control functionality. Such ICs often employ a rudimentary step forward and back interface to the microprocessor that limits system performance. Other stepper motor systems are PC card based and use a host PC to provide high performance control.

In embedded systems it is much better to use a small microcontroller to directly control the stepper motor. A very small microcontroller such as the C8051F300 is capable of providing a high performance motion control solution. The microcontroller implements a linear-velocity profile, generates the precise timing required, and outputs the stepping pattern used to drive the motor. The microcontroller directly drives the power MOSFETs and no addition gate drive circuitry is required.

The microcontroller also provides serial communications for remote control and distributed systems. This reference design uses a RS232 port operating at 57600 bps. This demonstrates the feasibility of using serial control. It is equally feasible to use SMBus, I2C, RS485, or some more advanced UART based network protocol. The C8051F300 is housed in a very small form factor MLP11 package, measuring only 3 mm square. The entire stepper motor drive can easily be integrated onto the back of a small stepper motor. A system with multiple motors may use a single small microcontroller for each motor.

The C8051F300 is ideally suited for driving a stepper motor. The small form factor lends itself to integrated motor solutions. The on chip UART and SMBus provide serial communication and control. The calibrated internal oscillator eliminates the cost and pin-count of using an external crystal, while providing an accurate time base for high speed UART and precise motor timing. The low-pin count package has enough pins to drive the stepper motor and RS232 transceiver, with two additional I/O pins left over for special functions.

This reference design demonstrates a high performance stepper motor system using the C8051F300. The reference design provides for both stand-alone demo operation and UART control. The reference design may also be used as a platform for stepper motor code development using the C2D two-wire on-chip debug and Flash programming interface. The reference design is complete with schematic, bill of materials, printed circuit board artwork, code flowcharts, and source code. The software is also available for download from the Silicon Laboratories web site.

2. Using the Stepper Motor Reference Design

2.1. Quick Start

The recommended stepper motor listed in the Bill of Materials is the GBM model number 42BYG205, available from Jameco Electronics®. Connect the GBM 42BYG205 stepper motor to the stepper motor reference design using the color code shown in Table 1.

Table 1. GBM 42BYG205 Color Code

Color	Name
red	A+
yellow	Acommon
blue	A-
green	B+
orange	Bcommon
brown	B-

Connect the 9 V DC power supply to the 2.1 mm power connection on the stepper motor reference design. Plug the power supply into 120 VAC power source. The LED labeled “PWR” should illuminate.

Press the function switch labeled “FUNC”. The stepper motor should turn four turns. The green status LED labeled “STAT” should illuminate while the motor is

Copyright © 2008 by Silicon Laboratories

turning. Press the function switch again. The motor will rotate four turns the other direction.

It the LED does not illuminate, check the power connection. If the motor does not turn, check the motor wiring.

If using a stepper motor other than the GBM 42BYG205, follow the color code provided with that particular stepper motor. Note that there is no standard color code for stepper motor wiring. It is best to double-check the wiring with a digital multi-meter. A 30 Ω stepper motor should measure 60 Ω from A+ to A- and 30 Ω from A+ or A- to Acommon. Phase B should measure similarly. A high impedance should be obtained when measuring from any phase A wire to any phase B wire.

2.2. Setting up HyperTerminal

Connect a DB9 serial modem cable to the stepper motor reference design RS232 connector. Connect the other end to a serial port on the back of a Windows PC. Note which COM port is connected to the Stepper Motor Reference Design.

Open HyperTerminal from the start menu. *Start>Programs>Accessories>Communication>HyperTerminal*. When prompted for a new connection name, type in *StepperMotor* or some other descriptive name. In the next dialog box, click on the *connect using* pull-down menu and select the appropriate COM port (e.g. COM4). Click on OK to exit the new connection dialog box. In the next dialog box choose 57600 bits per second, 8 data bits, no parity, 1 stop bit, and no flow control.

Now hit return a few times. A prompt sign and a new line should be displayed each time the return key is depressed. If prompt is not displayed, double-check the connections and the serial port settings. Make sure the stepper motor board is plugged in and powered up. To assist in debugging, test points are conveniently provided for the TX and RX connections on the stepper motor reference design.

2.3. Command Line Operation

The command line parser understands three commands. The commands are ***p*** for position, ***a*** for acceleration, and ***s*** for status. Each command must start with a lowercase letter. The position and acceleration commands are followed by a number string.

Type ***s*** and the stepper motor will display the current position and acceleration parameter. The text following the prompt sign is always user input.

```
>s
Position:
0
Acceleration:
80
>
```

The stepper motor will turn one complete rotation in 400 steps. Type ***p4000*** and then hit enter. The stepper motor will turn 10 rotations and then stop, While moving, the terminal will display the message ***Moving...*** and the current position of the motor. The display is updated periodically while moving. When the move is complete the number will stop at the final position and the terminal will display the message ***done!*** and a command prompt sign ***>***.

```
>p4000
Moving...
Position:
4000
done!
>
```

Now type ***a120*** and hit enter. The terminal will display the new acceleration to verify the parameter change.

```
>a120
Acceleration:
120
>
```

Now type ***p0*** and hit return. The stepper motor will rotate ten turns the other direction at a slower rate.

A smaller number results in a faster acceleration and a faster top speed. If you set the acceleration factor far too small the motor will stall at the maximum slewing speed. If the acceleration parameter marginally too small, the motor will have very low torque in the slewing region.

The parser will ignore any non-numeric characters between the command letter and the first number. For example ***p1000***, ***p 1000***, ***position 1000***, and ***pig 1000*** will all be interpreted as a position 1000 command. The parser does not understand capital letters.

The number parsing is terminated by the first non-numeric character. So it doesn't really matter what you type after the number. It could be a carriage return, space, period, or any non-numeric character.

The number parser for the position expects an unsigned 16-bit integer. You can enter any position from 0 to 65535. If you enter 65536, it will be interpreted as a zero. The acceleration parser expects an unsigned 8-bit

integer. The range is 0 to 255. If you enter 256 it will be interpreted as a zero. The number 257 will be interpreted as a one. Entering a zero or a very small integer may produce unpredictable results.

3. Theory of Operation

3.1. Motor Basics

The primary distinguishing feature of stepper motors is the manner in which they are driven. Stepper motors are moved in discrete steps. This is in contrast to other types of motors such as d.c. and brushless d.c. motors which are typically controlled using continuous mode analog control methodologies. The position of a stepper motor may be expressed using an integer. The stepping rate in steps per second is typically used to describe the angular velocity.

Because stepper motors are driven in discrete steps, they excel at absolute positioning applications. The most commonly available stepper motors move in precise increments of 1.8° or 0.9° per step.

Stepper motors are controlled directly. The primary command and control variable is the step position. This is in contrast to d.c. motors where the control variable is the motor voltage and the command variable may be either position or velocity. A d.c. motor requires a feedback control system and controls the position indirectly. A stepper motor system is normally operated “open loop”.

3.2. Stepper Motor Construction

Stepper motors may be classified by their motor construction, drive topology, and stepping pattern.

There are several different types of stepper motor construction. These include variable reluctance, permanent magnet, and hybrid permanent magnet. This reference design is applicable to the permanent magnet and hybrid two or four phase stepper motors. Permanent magnet stepper motors are very inexpensive and have a large stepping angle of 7.5° to 18°. Permanent magnet stepper motors are often used in inexpensive consumer products. Hybrid stepper motors are a bit more expensive and have stepping angles of 1.8° or 0.9°. Hybrid stepper motors are predominant in industrial motion control applications. Variable reluctance motors typically have three or five phases and require a different drive topology. Variable reluctance stepper motors are not addressed in this reference design.

The most common type of stepper motor construction used for industrial motion control is the hybrid permanent magnet motor. The rotor is constructed using a cylindrical permanent magnet oriented with the north-south polarity along the rotor axis. Two laminated end caps are used with many teeth around the periphery. The north and south teeth are staggered to provide many effective poles using a single permanent magnet. The stator laminates typically have four large forks. Each fork has many teeth. The teeth for the two windings are also staggered to line up with the appropriate teeth on the rotor. Using this clever arrangement, a 200-pole motor can be constructed using a single permanent magnet and only four stator windings.

3.3. Drive Types

The two common drive topologies for stepper motors are unipolar and bipolar. A unipolar drive uses four transistors to drive the two phases of the stepper motor. The motor has two center-tapped windings with six wires emanating from the motor as shown in Figure 1. This type of motor is sometimes rather confusingly called a *four-phase* motor. This is not an accurate representation as the motor really has only two phases. A more accurate description would be a two-phase, six-wire stepper motor. A six-wire stepper motor is also often called a *unipolar* stepper motor. However, a six-wire stepper motor could be used with either a unipolar or bipolar drive.

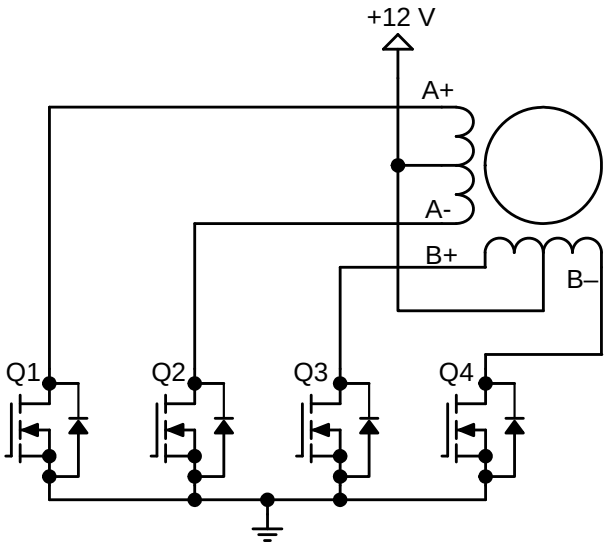


Figure 1. Unipolar Stepper Motor Drive

The center tap of the motor winding is connected to the positive voltage supply. Each coil can be energized in either direction by turning on the appropriate transistor. The center-tapped winding acts as a transformer. So the voltage on the unused switch will be twice the supply voltage.

A clamped unipolar drive circuit is shown in Figure 2. When Q1 is turned on, current will flow from the +12V supply, through the A winding, through Q1 to ground. When Q1 is turned off, the current will tend to continue to flow through the winding inductance. The drain voltage of Q1 will rise above the supply voltage. The center-tapped winding acts as a transformer. Thus, when the voltage on A+ reaches 24 V, the voltage on the A- terminal will go below ground and be clamped by the internal diode of Q2. There is also a considerable amount of uncoupled inductance in each winding. This will cause an additional overshoot voltage when the transistor is turned off. The four diodes D1-D4 and the clamp zener D5 form an effective clamp circuit to limit the overshoot voltage. The zener voltage should be slightly higher than twice the maximum supply voltage.

A bipolar stepper motor drive uses eight transistors to drive the two phases as shown in Figure 3. A stepper motor with four wires or six wires may be used with a bipolar drive. A four-wire motor can only be used with a bipolar drive. The four-wire motor might be marginally less expensive in high volume applications. The bipolar stepper motor drive uses twice as many transistors as the unipolar stepper motor drive. The four lower transistors can be usually be driven directly from the microcontroller. The upper transistors require a more expensive high-side drive. The bipolar drive transistors only need to withstand the motor supply voltage. The bipolar drive does not require a clamp circuit like the unipolar drive.

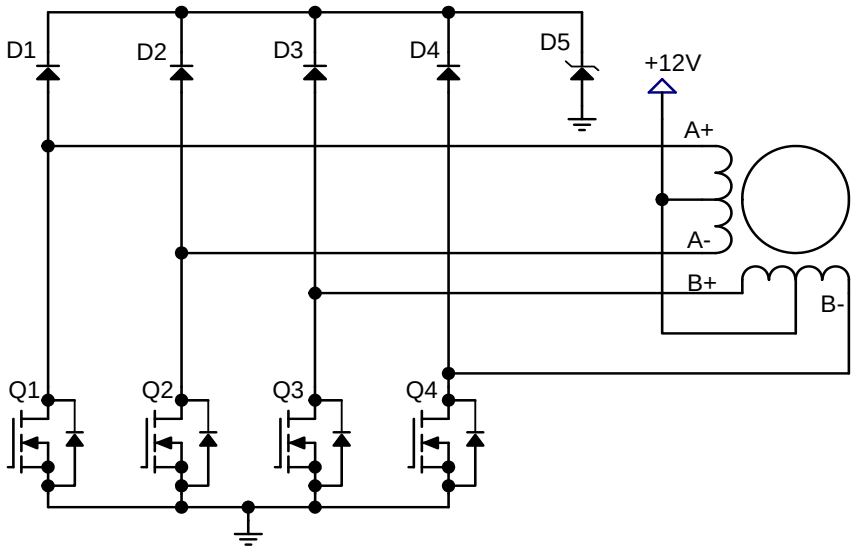


Figure 2. Unipolar Stepper Motor Drive with Zener Clamp

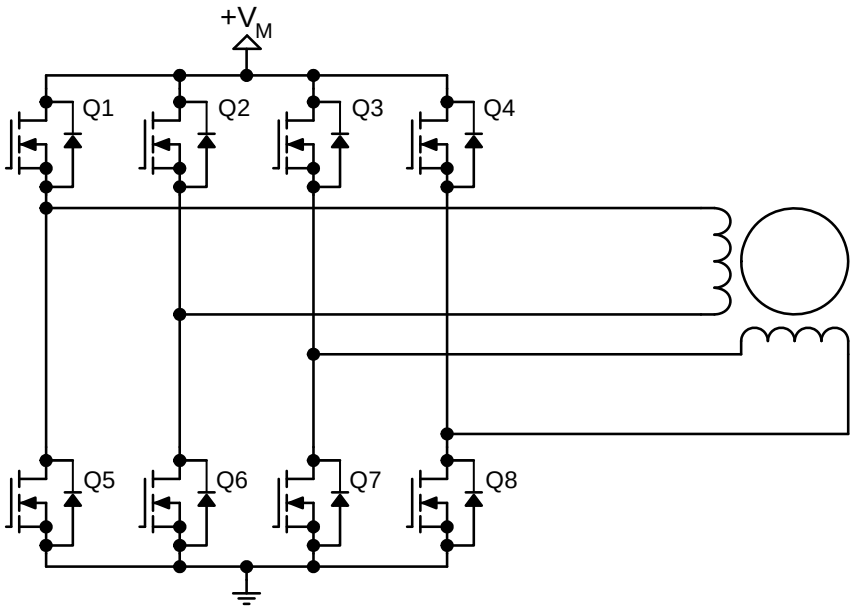


Figure 3. Bipolar Stepper Motor Drive

The performance differences between unipolar and bipolar drives are subtle. The unipolar drive only uses half of the actual motor windings at any one time. Thus, the bipolar stepper motor should theoretically have much better performance for a given motor volume. In practice, this is not always the case. Often the six-wire stepper motors have a lower phase resistance and consequently a higher holding torque for a particular motor size. The trade-offs of bipolar versus unipolar are summarized in Table 2.

Table 2. Bipolar vs. Unipolar Trade-offs

	Bipolar	Unipolar
number of transistors	8	4
number of high-side drivers	4	0
number of clamps	0	4
transistor voltage	1 x Vs	2.5 x Vs
winding usage	100%	50%
motor wires	4	6

3.4. Stepping Patterns

The two possible stepping patterns for stepper motors are full-step and half-step. A full-step pattern has four states and moves the motor one full step for each state. A 1.8° stepper motor will move 1.8° for each state and 7.2° for the full pattern. A full step pattern is shown in Table 3. In the full-step pattern, two transistors are always on. The first two columns indicate whether the A and B phase voltages are positive +, negative -, or high impedance *z*. The next four columns indicate the state of the four transistors for the unipolar stepper motor shown in Figure . The last column is the state of all four transistors expressed in hexadecimal for use with microcontrollers.

Table 3. Unipolar Full-Step Pattern

A	B	Q1	Q3	Q2	Q4	Hex
-	-	0	0	1	1	0x03
-	+	0	1	1	0	0x06
+	+	1	1	0	0	0x0C
+	-	1	0	0	1	0x09

Note that the transistor order in Table 3 has been rearranged listing Q3 before Q2 to yield a clear pattern. The polarity of the A and B windings is only important in determining if the rotation of the motor is clockwise or counter clockwise. Swapping the polarity of either phase will change the direction of the motor. Swapping A and B windings will result in no change of rotation.

Table 4 is the stepping pattern for a half step stepper motor. The half step stepping pattern has eight states.

Four of these states only have one transistor on at any one time. The half-step pattern allows a positioning accuracy of 0.9° for a 1.8° stepper motor. Note that the holding current and consequently the holding torque will be less for the states with only one transistor on at a time.

Table 4. Unipolar Half-Step Pattern

A	B	Q1	Q3	Q2	Q4	Hex
z	-	0	0	0	1	0x01
-	-	0	0	1	1	0x03
-	z	0	0	1	0	0x02
-	+	0	1	1	0	0x06
z	+	0	1	0	0	0x04
+	+	1	1	0	0	0x0C
+	z	1	0	0	0	0x08
+	-	1	0	0	1	0x09

The step pattern for a bipolar stepper motor is similar. Normally the diagonally opposite transistors in each H-bridge are turned on simultaneously. Thus the stepper pattern can be readily understood by listing the upper transistors first in the same order as the unipolar drive. Then the lower transistors are listed in the order corresponding to the same pattern (13246857). The end result is that the upper and lower nibbles of the hexadecimal stepping pattern are identical.

Table 5. Bipolar Full-Step Pattern

A	B	Q1	Q3	Q2	Q4	Q6	Q8	Q5	Q7	Hex
0	-	0	0	0	1	0	0	0	1	0x11
-	-	0	0	1	1	0	0	1	1	0x33
-	0	0	0	1	0	0	0	1	0	0x22
-	+	0	1	1	0	0	1	1	0	0x66
0	+	0	1	0	0	0	1	0	0	0x44
+	+	1	1	0	0	1	1	0	0	0xCC
+	0	1	0	0	0	1	0	0	0	0x88
+	-	1	0	0	1	1	0	0	1	0x99

3.5. Stepping Algorithm

A Linear-Velocity Profile is shown in Figure 4. The velocity ramps up and down in the shape of a trapezoid. The three distinct phases are named the acceleration phase, the constant velocity or slewing phase, and the deceleration phase. The resulting angular step position curve *n* is a smooth s-shaped curve. The acceleration is constant in the acceleration and deceleration phases. The acceleration is zero in the slewing phase.

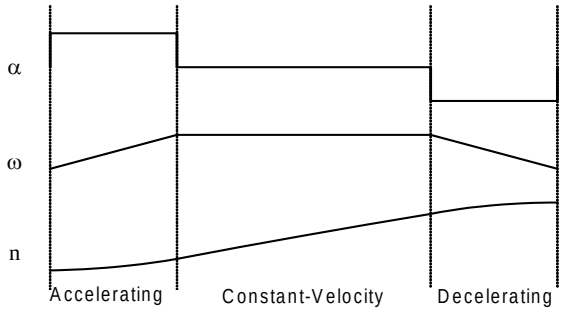


Figure 4. Linear Velocity Profile

We would like to derive a table or step periods that will be used to commutate the motor. The step period T_n is defined as the difference in time between two adjacent steps in Equation 1. The step period will be used to control the microcontroller timer.

Equation 1.

$$T_n = t_{n+1} - t_n$$

The angular acceleration is defined in Equation 2 and the step position is defined in Equation 3. These equations are straightforward textbook definitions for angular velocity and position. Since the stepper motor position moves in discrete steps the step position is an integer denoted by the letter *n* instead of θ.

Equation 2.

$$\omega = \alpha t_n$$

Equation 3.

$$n = \frac{1}{2}\alpha t_n^2$$

Solving Equation 3 for time gives the results shown in Equation 4. This is the absolute time required to provide a linear acceleration profile. This would be useful if we were working in absolute time and scheduling each commutation point based on a cumulative count from the beginning. However, we would like to use a relative count for each step period.

Equation 4.

$$t_n = \sqrt{\frac{2n}{\alpha}}$$

The definition of the step period from Equation 1 is used with the results in Equation 4 to provide an equation for the step period listed in Equation 5. The constant acceleration term has been factored out and is called T_0 as defined in Equation 6. The value of T_0 will determine the step period of the initial step with n equal to zero. Thus, we can use a single table for the relationship and let T_0 be a variable. This means one table can be used with any stepper motor.

Equation 5.

$$T_n = t_{n+1} - t_n = T_0(\sqrt{n+1} - \sqrt{n})$$

Equation 6.

$$T_0 = \sqrt{\frac{2}{\alpha}}$$

3.6. Common Mistakes

As demonstrated in the preceding section, the linear velocity profile is not as simple as it would first appear. The values stored in the linear velocity table must closely follow the non-linear equation shown in Equation 5. Often engineers in a hurry to get hardware up and running do not use the proper relation for the stepper motor table. This is a very common mistake that is very easy to make.

The most common mistake is to have the step period decrease linearly with the step number. For example, one might have an initial step period of 256 timer ticks and decrease the step period by one each time. This results in a non-linear velocity that is increasing hyperbolically as the step period approaches zero. Such a profile will hardly move at first and then the velocity will increase much too quickly.

The second most common mistake is to have the step period decrease with the inverse of the step number. This results in a velocity that is linear with respect to the step number. But this ignores the fact that the step period is constantly changing. The velocity should be plotted against the cumulative time, not the step number. If the velocity is plotted against the absolute time, the resulting curve is a second order function. That is, the velocity is increasing with the square of time. This profile also starts out too slow and ends up accelerating too fast.

3.7. Linear-Velocity vs. Linear-Acceleration

Many engineers hold a preconceived notion that a forth-order linear-acceleration profile will provide much better dynamic performance than a linear velocity profile. A linear-acceleration order profile has an acceleration that is trapezoidal in nature and velocity shaped in an s-curve.

A linear-acceleration profiler provides only marginally better dynamic performance in some systems. Only a few applications can actually benefit from a forth-order profile. For example, a printer head driven by an elastic band might benefit from the improved smoothness. The linear-acceleration profiler will have a smoother transition between the acceleration and slewing phases. The maximum step rate is dictated by motor parameters and will be the same in either case.

The linear velocity profile has several advantages over the forth-order profile: The linear velocity profile can be implemented using a single table. The step table is fixed. It can use a single multiply function to provide a variable acceleration. Using the table avoids having to calculate complex functions like a square root. Calculating the profile is very simple.

In contrast the linear-acceleration profile is much more complex. A single fixed table cannot be used. The initial conditions of each acceleration phase depend on the length of all prior phases.

The recommended solution is to always start out using a properly implemented linear-velocity profiler. This will be the best solution for most applications. Verify that the velocity is actually linear and evaluate the dynamic response of the system. If the dynamic response of the system does not meet the requirements, consider using a linear-acceleration profile.

3.8. Interrupt Based Algorithm

When developing an algorithm to control a stepper motor using a small microcontroller, it is important to consider the manner in which the code will be executed. A simple sequential algorithm could accomplish the task. The sequential algorithm might calculate the current step time and figure out what to do next depending on the acceleration phase. However, such an algorithm would end up writing to the timer and then waiting until the timer times out. It would spend most of the time just waiting on the timer.

Fortunately most MCU timers are capable of generating interrupts. Thus, we can set up the timer to generate an interrupt after one step period. When the interrupt occurs, the MCU should commutate the motor and update the timer with the next step period.

Now considering that we want to make the stepper motor control interrupt based, we must use a different paradigm. The timer interrupt service routine should be small, fast, robust, and only do what must be done on each commutation period. Anything that can be calculated once beforehand will be done outside the interrupt service routine. Values may be stored in global variables to be accessed by the interrupt service routine.

Using this scheme there are two basic pieces of code. The first is the profiler or the ***move()*** function. The second is the timer interrupt service routine. The profiler is called from the main loop and is executed in the foreground. The profiler calculates the global variables based on the target location and the current position of the motor. The function is named ***move()*** so that the user code makes sense in plain English.

This reference design uses a simple divide-by-four profiler. This means that the total number of steps is divided by four. The motor will accelerate and decelerate for one forth of the total number of steps. The remainder of the steps will be at a constant velocity. Some actual profiles are shown in Figure 5. Note that the total time accelerating for short profiles is much more that one-forth the total time. This is due to the effect of the variable step period.

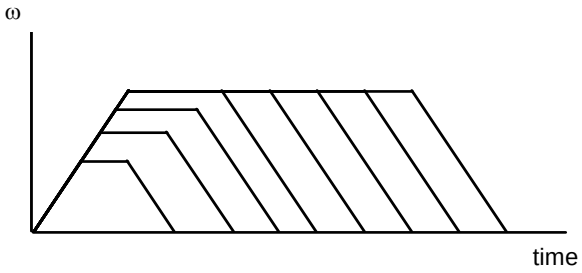


Figure 5. Different Profiles

The constant-acceleration and deceleration phases are accomplished by incrementing and decrementing an index for the stepper motor table. Incrementing the table index by one each step will accelerate the motor. Decrementing the index each step will decelerate the motor. The maximum index and the corresponding minimum period determine the top speed of the motor for a particular profile.

4. Hardware Design

The Stepper motor reference design hardware consists of four sections: the C8051F300 microcontroller, the power electronics, the voltage regulator, and the serial interface. The full schematic is included in Appendix A. The Bill of Materials is in Appendix B and the printed circuit board artwork is in Appendix C.

4.1. Microcontroller

The reference design features the C8051F300 microcontroller. This microcontroller is housed in a tiny 3 mm by 3 mm 11-lead micro lead package (MLP). This package is small enough to be integrated into the smallest motor. The C8051F30x family includes five devices with various options. The ‘F300 and ‘F301 include a calibrated internal oscillator. The internal oscillator is calibrated to within ±2% at test. This is close enough to use the internal oscillator for the UART with baud rates up to 115.2 kbps. The reference design does not utilize 8-bit 500 ksps ADC in the ‘F300. Thus, the design could use either the ‘F300 or the ‘F301 which does not include the ADC. The ADC on the F300 might prove useful in some designs for monitoring the dc motor voltage, stepper motor current, or the stepper motor temperature.

The unipolar stepper motor drive requires four outputs to drive the transistors. P0.0 through pin P0.3 are used to drive the power MOSFETs. The pins have been

swapped during the layout phase for optimum routing to the MOSFET gates. The stepping pattern has been changed accordingly. The port output current is sufficient to drive most small power MOSFETs with sub-microsecond switching times.

The I/O pins of the MCU are by default configured as inputs with a weak pull-up transistor. This has the inadvertent effect of turning on all four transistors momentarily when the MCU is first powered up. This is usually not a problem for unipolar stepper motors since the current will be limited by the winding resistance of the motor. Bipolar drive will require either pull-down resistors or inverting gate drivers to accommodate the default pull-ups.

P0.6 is used to drive an LED indicator that illuminates while the motor is moving. The C2 data signal is pin shared with the active-low switch “SW1” on pin 10. The switch is used to initiate a pre-programmed move so that the board can be used without the serial interface when desired.

The C2 reset signal is pin shared with a manual reset button on pin 8. Momentarily pressing the reset button will reset the MCU and turn all of the output transistors off. Note that the MCU will be held in reset for as long as the reset button is held down. As a result, all four output transistors will be turned on while the reset button is depressed.

4.2. Power Electronics

The power MOSFET selected for the stepper motor reference design is the Fairchild FDS8926A. These are small low on-resistance power MOSFETs in an SO8 package. These MOSFETs were chosen for their 30 V rating, 3 V gate, and SO8 package. A maximum drain to source rating of 30 V is required to drive the 12 V stepper motor. The MOSFET chosen should be compatible with a 3 V MCU. The relevant on-resistance rating of the FDS8926A is 38 mΩ at 2.5 V. These MOSFETs should easily handle 2-3 A in this application.

As a practical matter, the large dc wall-mounted transformer used with the reference design has a rating of 1 A at 9 V. The open load voltage of this supply is about 12 V. The output voltage decreases to 9 V at 1 A. This is sufficient to drive a small Nema 17 stepper motor with a voltage rating of 12 V or 9.6 V and a resistance of 30 Ω or greater. A regulated lab supply can be used to drive larger stepper motors up to 3 A.

The 330 Ω resistors were chose to provide turn-off time of just under 1 μs. During the drain to source rise time, the gate to source voltage is about 1.5 V. This is called the plateau voltage and will depend on the threshold and transconductance of the chosen MOSFET. The plateau voltage may be obtained from the gate charge curve as shown in Figure 6. Neglecting the V_{OL} of the MCU, there is about 1.5 V across the gate drive resistor. The 330 Ω resistors provide about 5 mA of gate drive current during turn off. The gate to drain gate charge Q_{gd} of this MOSFET is about 5 nC. This gives a switching time of about 1 μs. The measured values are very close to this value. The measured V_{OL} of the MCU is about 100 mV at this current.

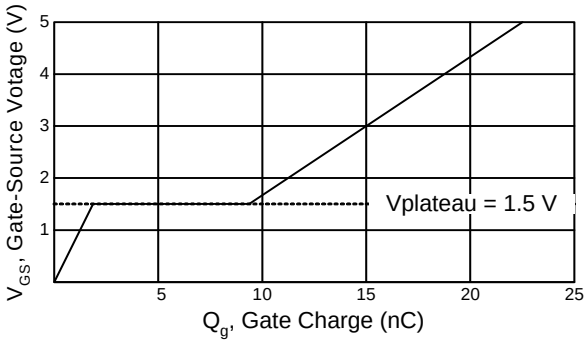


Figure 6. MOSFET Gate Charge

It is not advisable to turn off the MOSFETs too fast. Using no resistor or too small of a resistor will result in much faster turn-off. The turn-off time will determine the rate of the change in current or di/dt. The overshoot voltage at turn-off depends on the unclamped inductance and the di/dt. Excessive overshoot voltage could damage the power MOSFETs. An excessively fast turn-off may also hinder the ability of the Zener clamp to effectively protect the Power MOSFET. The resistor also protects the MCU from excessive currents and voltage transients.

The stepper motor reference design uses a clamp circuit to protect the power MOSFETS from excessive overshoot voltage. A zener voltage of 27 V was chosen to protect the 30 V MOSFETs. This clamp circuit is a cautionary measure to protect the MOSFETs when used with different motors and voltage supplies. In many fixed applications it is possible to just let the MOSFETs avalanche during turn-off. The MOSFETs must be capable of handling the peak current and energy stored in the unclamped inductance. The current and motor inductance will vary greatly from motor to motor. The clamp circuit is useful when a single drive board might be used with various motors.

4.3. Voltage Regulator

The voltage regulator is an LM2973-3.3. This is a 3.3 V low-dropout regulator in a SOT223 package. The maximum continuous input voltage for this device is 26 V. It is important to consider the maximum open circuit voltage of the power supply when selecting a voltage

regulator. The total worst-case V_{DD} current draw for the board is about 25 mA. This condition occurs when both LEDS are on and the serial port is running at 115.2 kbps. This results in a worst-case power dissipation in the regulator of about 300 mW with the input voltage at 15 V. The large tab of the SOT223 is connected to a large ground plane to improve heatsinking.

4.4. Serial Port

The serial port transceiver is a Sipex SP3223U. This is a 3.3 V RS232 transceiver. The TX and RX pins of the transceiver are connected to P0.4 and P0.5 of the C8051F300. Test points are provided for the TX and RX connections. The extra transceiver channel is used to loop RTS back to CTS. This ensures that the terminal will work when hardware handshaking is enabled.

4.5. PC Board Layout

The two-layer printed circuit board layout is divided into two routing areas with logic circuits on the left and power circuits on the right. The logic circuitry uses a ground plane top and bottom. The minimum clearance from pad to pad for the C8051F300 MLP11 footprint is about 8 mils. The actual PC board layout uses 10 mil traces and a 10 mil ground plane clearance. The design rule checker also uses a 10 mil clearance. Only the pads of the MLP package have less than 10 mils clearance.

Most PC board fabrication companies can readily manufacture boards with 8 mil clearances; including most quick-turn PCB companies. But this does limit the copper weight to 0.5 oz. raw stock with a finished plated weight of 1 oz.

Using a finished copper weight of 1 oz., the high current conductors must be very wide. A copper width of 100 mils (2.5 mm) will give a temperature rise of 10 °C at 5 A. It is not practical to use conductors this wide with the small SO8 package, so copper pour regions were used for the motor current conductors. The part placement is optimized to provide large copper pour areas for the ground, 12 V supply, clamp, and motor outputs. This has the benefit of using all available copper for current conduction and heatsinking. The inductance for the clamp circuit is also minimized by using a large ground plane area on the top and a large 12 V plane on the bottom.

The PC board uses a single point grounding scheme with separate grounds for the digital and power sections. The grounds are connected together using a ground test point footprint. This is a contrived mechanism to permit the integrated PC board software to manage the grounds separately and ensure there are no ground loops. A single design rule error may be generated for the ground test point.

5. Software Design

5.1. Port Configuration

The C8051F300 family of products has a very useful feature call the Digital Crossbar. Using the Digital Crossbar, the end user can select which of the many peripherals can access the port pins.

In the stepper motor reference design P0.0 through P0.3 are used to drive the stepper motor. The corresponding bits are set in the XBR0 register causing the Crossbar to skip these pins. This means that P0.0 through P0.3 are not available as digital I/O for any of the internal peripherals. Bits 0 through 3 are also set in the P0MDOUT register. This configures the pins as push-pull outputs so that they can drive the power MOSFETs both high and low.

The stepper motor I/O pins can be controlled by either setting the bits P0.0 to P0.3 one at a time or by writing a byte to the P0 register. We would like the pins to change state simultaneously, so we will write a byte to the P0 register. Caution must be exercised when writing to the P0 register as this affects all of the port pins. Any pins configured as push-pull outputs will be driven high or low. Pins that are used as inputs may also be used as open drain outputs. So if we wish these pins to remain inputs, we must set the corresponding bits to 1 when writing to P0.

Port pins P0.4 and P0.5 are allocated to the UART by setting Bit 0 and Bit 1 in the XBR1 register. This puts the UART in control of these pins. Writing to P0 will have no effect on pins P0.4 and P0.5. Unlike other peripherals that are assigned based on priority, the UART pins are always assigned to P0.4 and P0.5 when enabled. When using the UART, the corresponding pin skip bits in XBR0 should not be set. Also Bit 4 of P0MDOUT is set to configure the TX pin as a push-pull output.

The two remaining pins of P0 are used for a status LED and a control push-button switch. P0.6 was selected for the LED. The corresponding Bit 6 is set in XBR0 to skip P0.6. Bit 6 in P0MDOUT is also set to configure the port pin as a push-pull output. This is not absolutely necessary since the LED is only driven when low.

P0.7 is used as the push-button switch input. No special action is required to configure this pin as an input. However, one must be careful not to allocate pins to any other peripherals. Since P0.0 through P0.6 have already been allocated, P0.7 is next on the allocation chain. There is no pin skip bit for P0.7. For example, enabling the SYSCLK output would force the SYSCLK on P0.7. So for P0.7 to be reserved as an input, bits 4 through 7 of XBR1 must be zero.

5.2. User Interface

The primary user interface is an ASCII terminal with a baud rate of 57,600 bps. The main loop of the code parses the characters from the terminal and performs the appropriate action. The flowchart for the main loop is shown in Figure D1.

The main loop first checks to see if the motor is moving. The LED bit is used to turn on the LED and is also used as a state variable. The LED is turned on by the move function and is turned off by the Timer ISR on completion of the move. While the motor is moving the main loop will periodically update the position to the terminal display. This is accomplished by overwriting the previous position. The position is written to the terminal followed by a carriage return without the normal line feed character. This overwrites the previous number. The position is written to the terminal using the ***puts()*** put string function. A string of four space characters are used to blank trailing characters. This is necessary when moving from a large 5-digit number to a small 1-digit number. A short delay is used to prevent the transmitter buffer from overflowing.

Next the main loop checks the ***doneFlag***. The Timer ISR sets the ***doneFlag*** when the move is complete. If ***doneFlag*** is set, the main loop will call the ***doneMsg()*** function. The ***doneMsg()*** function displays the word "done" and readies the terminal for the next command. The ***doneFlag*** is cleared by ***doneMsg()***.

Next the main loop enters the command parser. The command parser is implemented using a large switch statement. The parser gets one character from the terminal and uses a case statement to do different things depending on the character.

If the character is a carriage return, a newline and prompt will be transmitted to the terminal. Repeatedly entering a carriage return will display multiple prompt characters, each on a new line.

If the character is a ***p***, the parser will get a 16-bit number from the terminal using the ***getuint()*** function, and move to the new position.

If the character is an ***a***, the parser will get an 8-bit number from the terminal using the ***getuchar()*** function, store the new value, and display the new value to the screen for confirmation.

If the character is an ***s***, the main loop will display the current status information - position and acceleration. The status command does not wait for any additional characters.

If the parser does not recognize the character, an invalid character message will be displayed.

5.3. Move Function

A flowchart for the move function is shown in Figure D2. The move function first checks to see if the motor is already at the target position. If the motor is already at position zero and you enter the command p0, the motor will not move and the done message will be displayed. This is necessary to ensure that the algorithm works properly. The smallest valid move is 1 step in either direction. The current position is the value stored in the global variable ***Position***. The desired position, passed as a parameter to the ***move()*** function, is called ***targetPosition***. The forward flag is set or cleared depending on whether the target is greater than the position. Both ***targetPosition*** and ***Position*** are unsigned integers. The smaller is subtracted from the larger to get the variable ***length***.

The global variable ***TableMax*** defines how many steps the motor will accelerate and decelerate. While accelerating the ***TableIndex*** will be incremented until it reaches ***TableMax***. If ***length*** is less than 1024, the value for ***TableMax*** is calculated by dividing ***length*** by four. If ***length*** is greater than 1024, ***TableMax*** is set to 255. This is the maximum allowable value for ***TableIndex***.

Taking ***length***, subtracting two times ***TableMax*** for the acceleration/deceleration steps, and subtracting one more for the initial step gives the number of constant velocity slewing steps. This ensures that the total number of steps is exactly equal to the length.

The move function then schedules the timer to execute the first step after a short delay. After the initial step, the Timer Interrupt service routine ***Timer_ISR()*** will control the stepper motor timing from then on.

5.4. Timer ISR

The timer interrupt service routine moves the motor according to the prescribed profile. The interrupt service routine will first commutate the motor. Then the ***PatternIndex*** is incremented or decremented according to the stepper motor direction. The ***PatternIndex*** is used as a pointer to the step pattern. It is not to be confused with the ***TableIndex*** that points to the linear velocity table. The step pattern index for a unipolar stepper motor is a modulus 8 counter. It counts from zero to seven. The step pattern is written to P0. Bit 7 is set to a 1 to ensure that P0.7 remains configured as an input.

The timer interrupt service routine determines what to do next depending on the current state of the motor. The four possible states are *Accelerating*, *Decelerating*, *Slewing*, or *Done*. If the motor state is *Done*, the timer ISR will disable further timer interrupts, turn off the LED,

and set the ***doneFlag***. The ***doneFlag*** is used as a handshaking mechanism for the main loop to display the done message.

If the motor state is not *Done*, the timer ISR will calculate the value for the next interrupt and write to the TL0 and TH0 special function registers. Next the ***TableIndex*** or ***SlewCount*** is modified according to the motor State. During *Acceleration*, the ***TableIndex*** is incremented. During *Deceleration* the ***TableIndex*** is decremented. If the motor state is *Slewing* the ***SlewCount*** will be decremented.

5.5. UART Functions

The UART functions implement fully buffered I/O using the C8051 receiver and transmitter. The benefit of using buffered I/O is that most short messages will be sent or received in a non-blocking manner. This means that the CPU does not have to wait for each byte to be transmitter or received. The user code does not write or read data directly to the serial port. Instead the user functions write or read data to the buffers. A UART interrupt service routine transmits characters from the write buffer and places received characters into the read buffer.

The UART functions are modeled after the ANSI ***stdio*** library functions. Each function has been optimized for use in a small microcontroller. This is more code efficient than including the entire ***stdio*** library and rewriting the bottom layer functions.

The put string ***puts()*** function is used to display messages on the terminal. There are two versions of the get integer & put integer functions, one for 8-bit unsigned char data and one for 16-bit unsigned int data.

The middle layers of the serial protocol are the put character ***putc()*** and get character ***getc()*** functions. These are the lowest level functions that should be called by the user code.

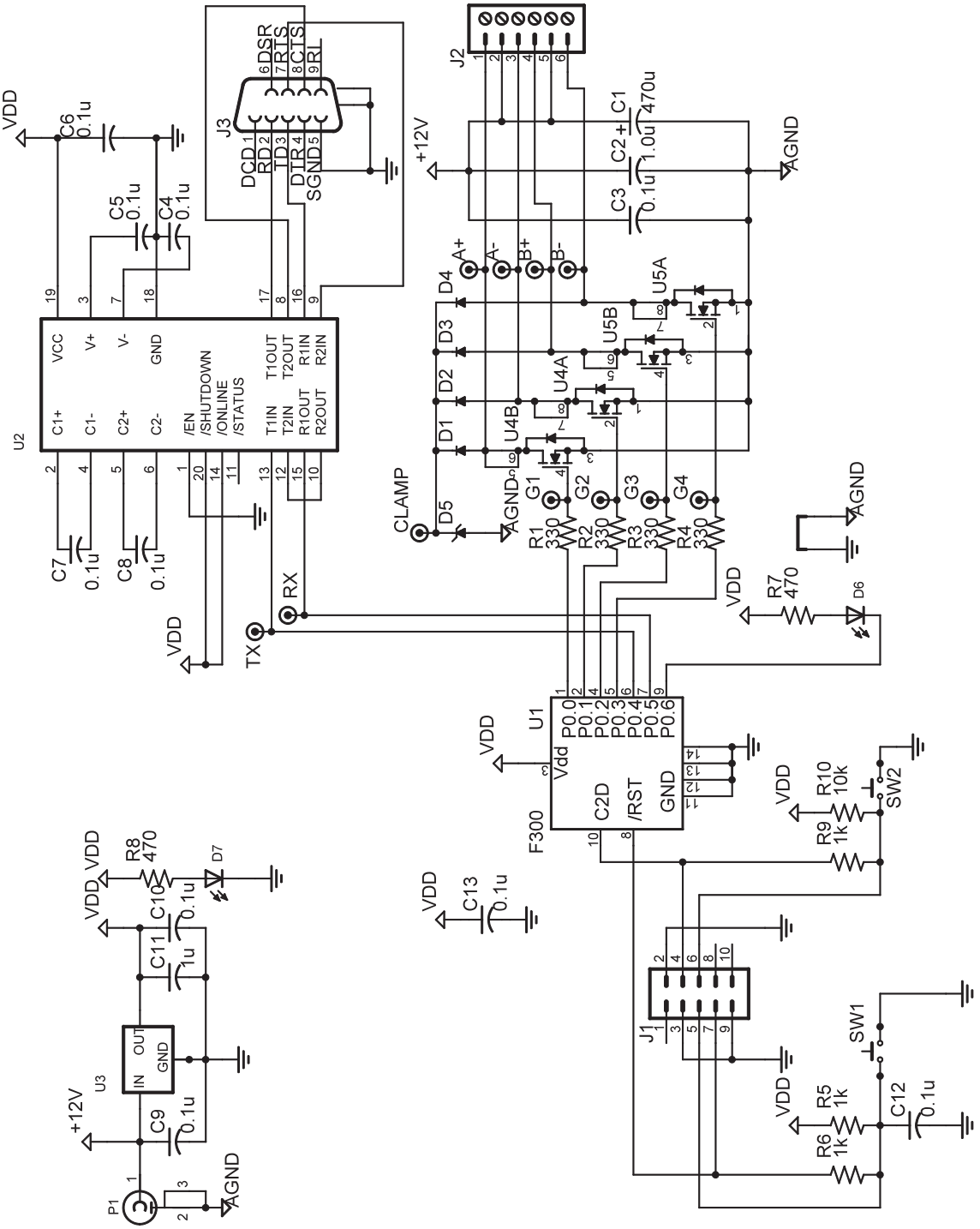
The bottom layers of the serial communications protocol are the write character ***writec()*** and read character ***readc()*** functions. These functions write and read directly to the read and write buffers. These functions are called from the ***getc()*** and ***putc()*** functions. The user code should not use ***readc()*** and ***writec()*** functions directly. The ***readc()*** and ***writec()*** functions momentarily disable UART interrupts while accessing the read and write buffers to avoid contentions. The ***writec()*** function also restarts the transmitter if it is idle.

A single UART Interrupt service routine ***UART_ISR*** is used for both transmit and receive interrupts. The ISR must check both the transmitter and receiver to determine which needs servicing. If the write buffer is empty, the ***UART_ISR*** will set the ***TX_Idle*** flag bit and clear the transmit interrupt flag without writing to ***SBUF0***. This will disable the transmitter until restarted. The ***TX_Idle*** flag bit is used for handshaking with the ***writec()*** function. The ***writec()*** function checks the status of the ***TX_Idle*** flag bit and restarts the transmitter if necessary.

6. References

1. Takashi Kenjo, “Stepping motors and their Microprocessor Controls”, Oxford University Press, 1984.
2. Ken Berringer, “Linear Velocity Control of Stepper Motors”,
Incremental Motion Systems Symposium, June 1999

APPENDIX A - SCHEMATIC



APPENDIX B - BILL OF MATERIALS

Qty	Designator	Description	Value	pkg	mfr PN	Manufacturer
1	C1	electrolytic capacitor	470 u		EEU-FC1E471	Panasonic
2	C2, C11	chip capacitor	1. 0u	1206		
10	C3-10, C12, C13	chip capacitor	0.1 u	805		
4	R1, R2, R3, R4	chip resistor	330	805		
1	R10	chip resistor	10 k	805		
3	R5, R6, R9	chip resistor	1 k	805		
2	R7, R8	chip resistor	470	805		
4	D1, D2, D3, D4	Schottkey diode	30 V	SMB	10BQ30	International Rectifier
1	D5	Zener Diode	27 V	SMA	SMAZ27-13	Diodes Inc
2	D6, D7	LED				
1	U1	Microcontroller		MLP11	C8051F301	Silicon Laboratories
1	U2	RS232 Transceiver		TSSOP20	SP3223E	Sipex
1	U3	3.3V Regulator		SOT223	LM2937	National
2	U4, U5	Power MOSFET		SO8	FDS8926A	Fairchild
2	SW1, SW2	Switch			EVQ-PAD04M	Panasonic
1	P1	Power Connector			RAPC722	Switchcraft
1	J1	Shrouded Header			2510-6002UB	3M
1	J2	Terminal Block			1729160	Phoenix Contact
1	J3	DB9 Connector			745781	Amp
1	T1	unregulated 9VDC Wall Transformer			830AS09100	Tamura

APPENDIX C - PCB ARTWORK

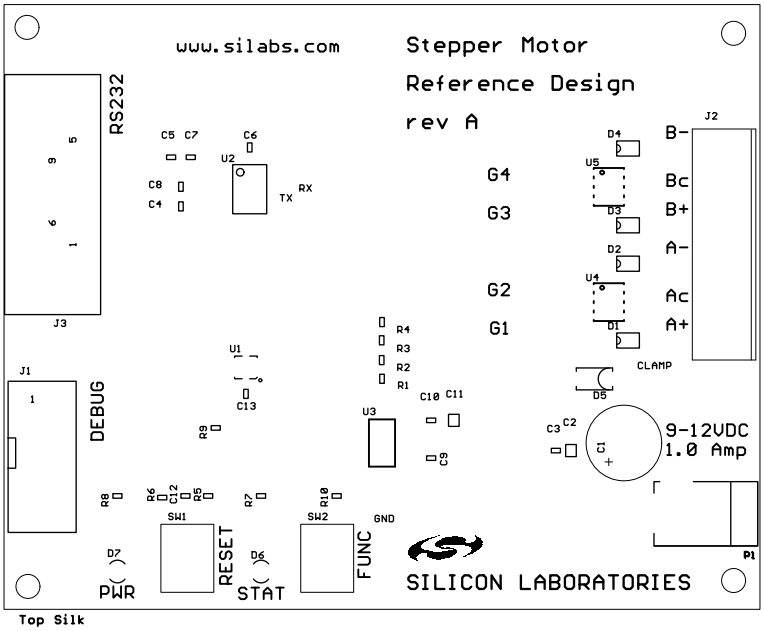


Figure C1. Silk Layer

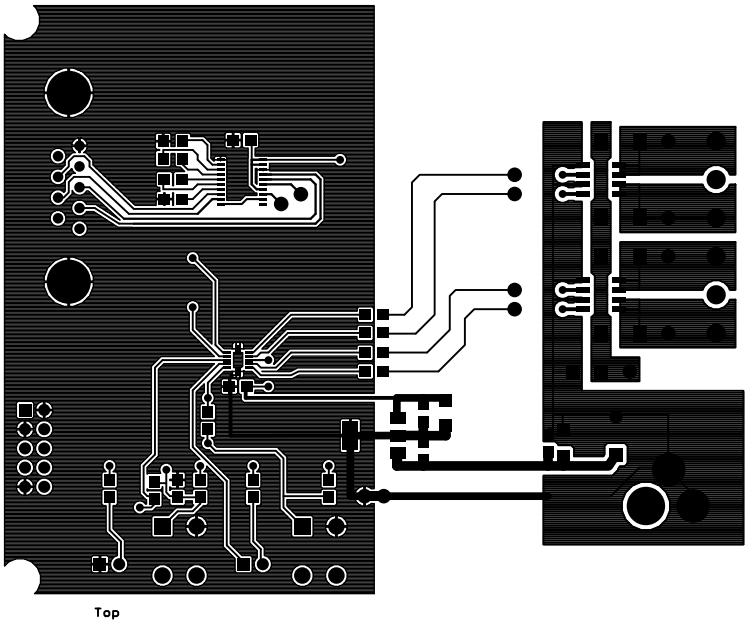


Figure C2. Top Layer

r

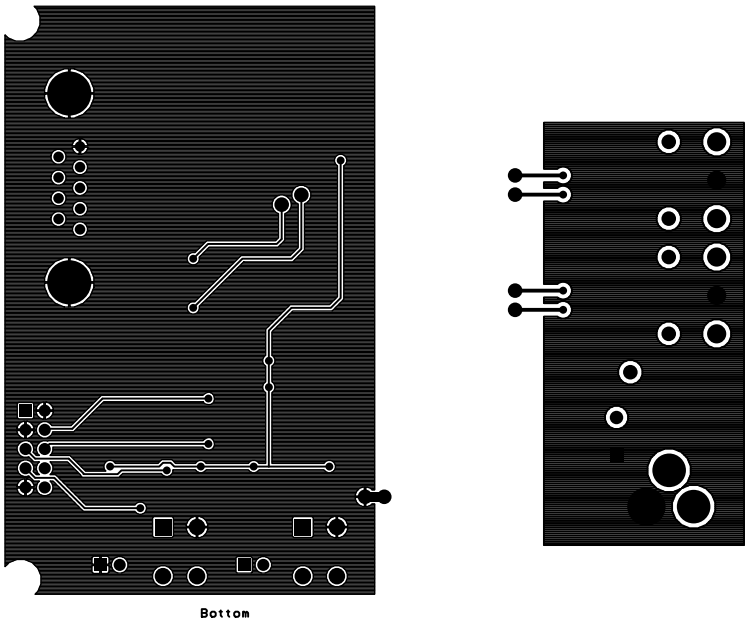


Figure C3. Bottom Laye

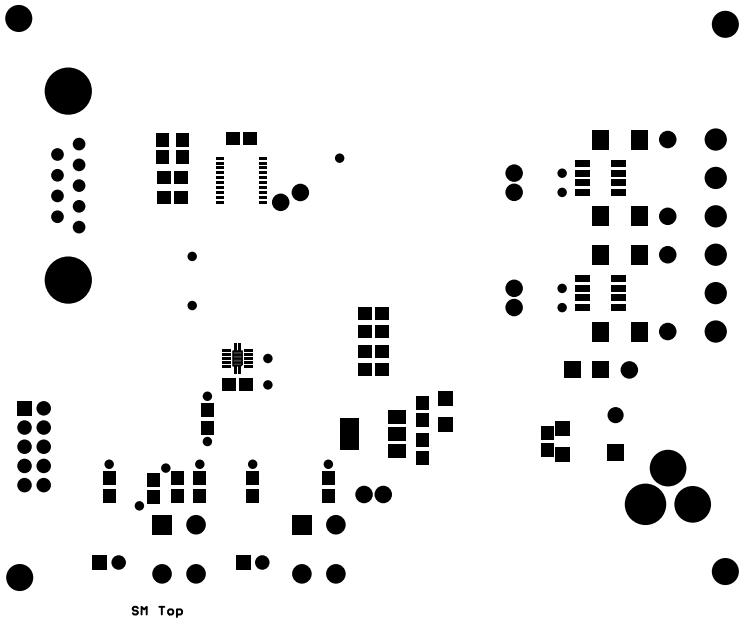


Figure C4. Top Solder Mask

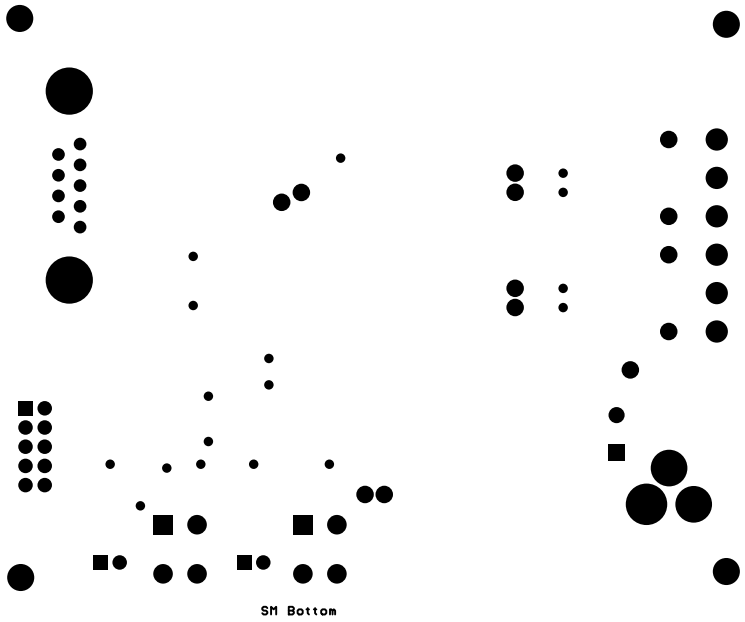


Figure C5. Bottom Solder Mask

APPENDIX D - CODE FLOWCHARTS

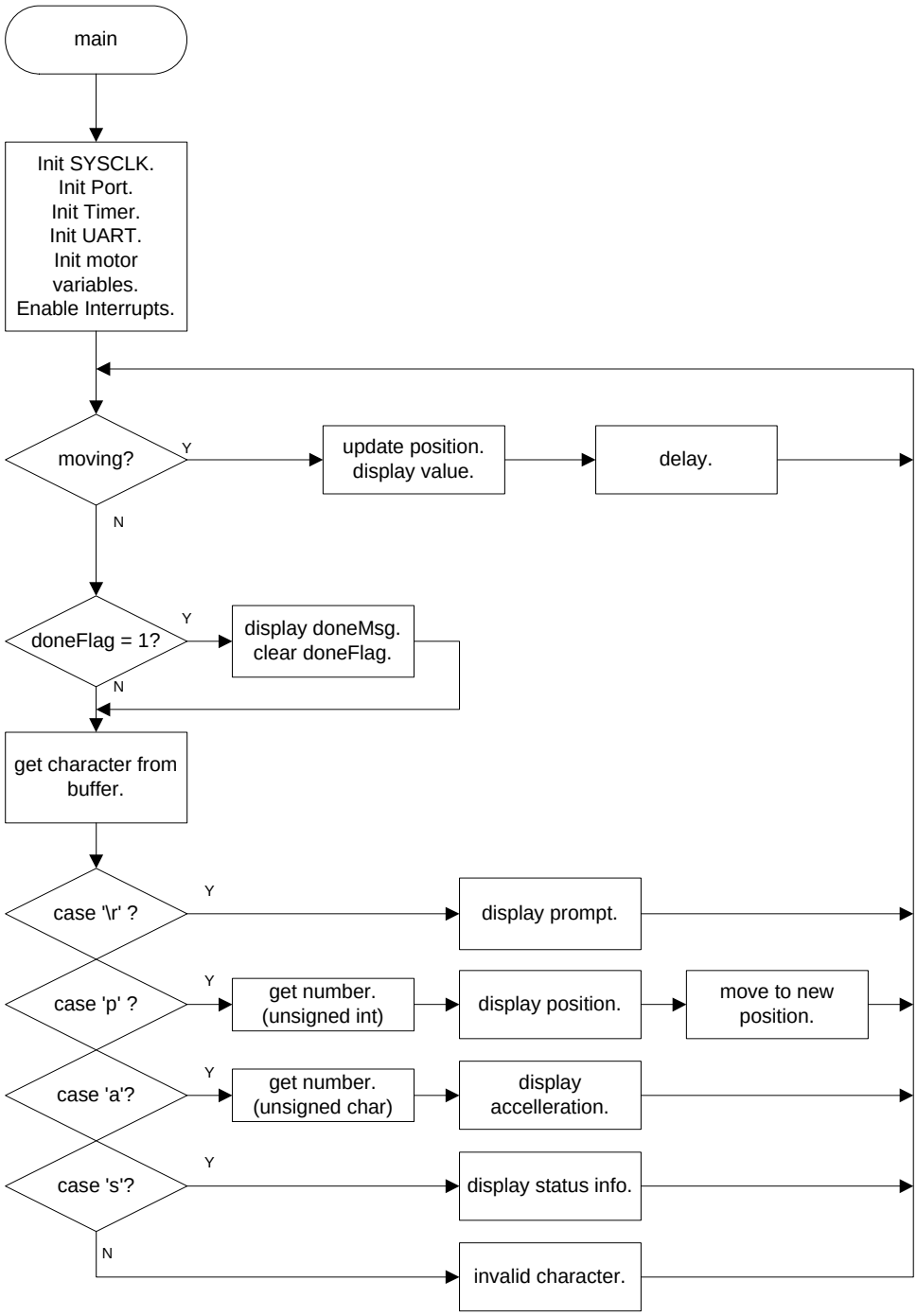


Figure D1. Main Loop

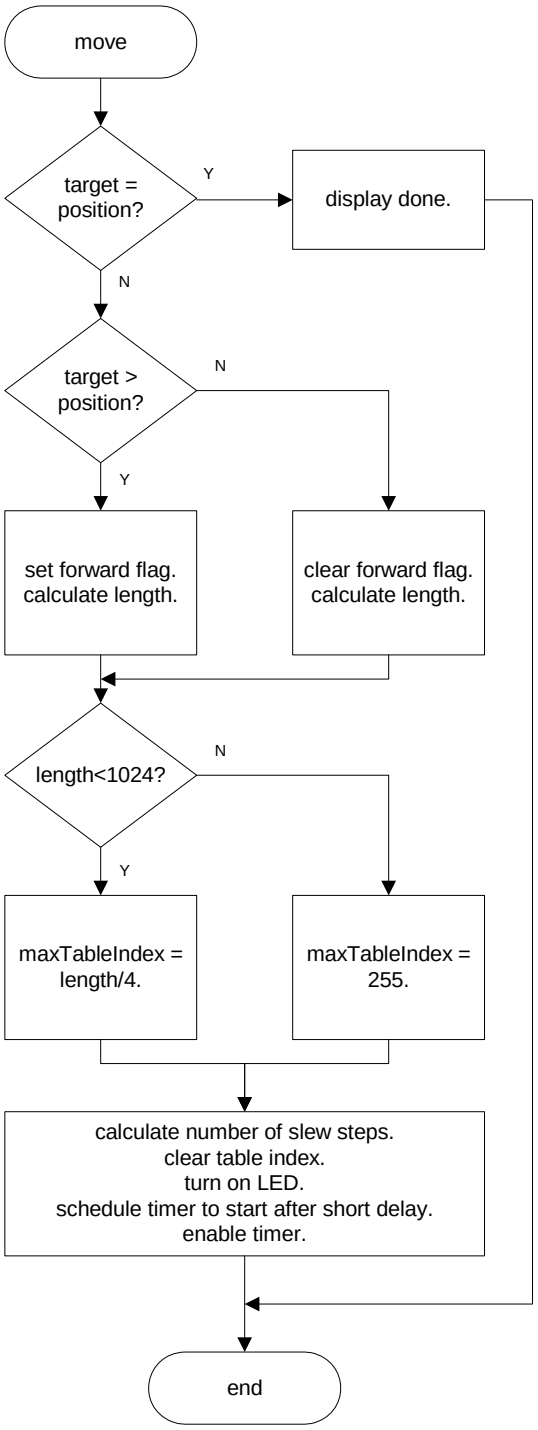


Figure D2. Move (Profiler)

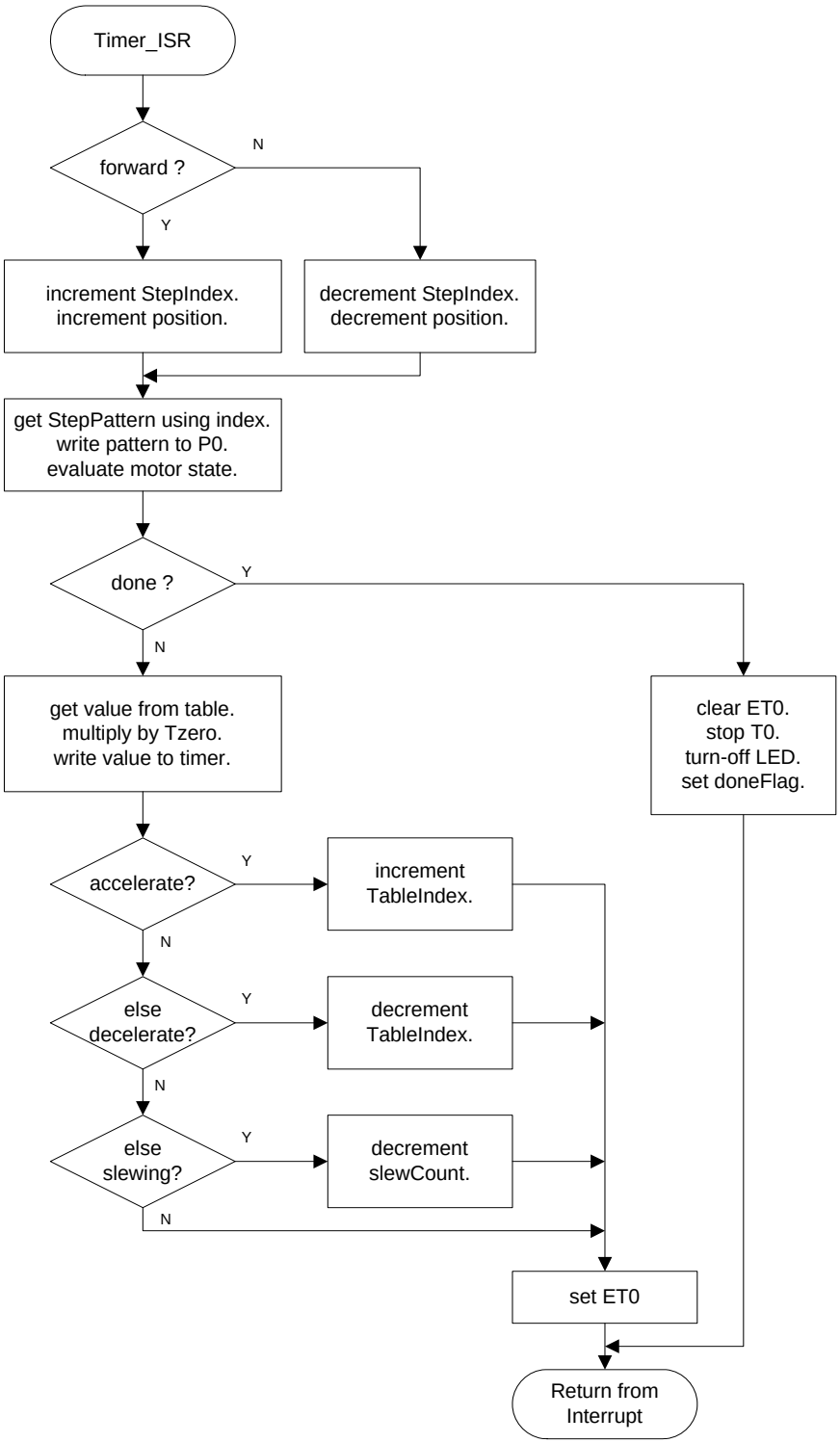


Figure D3. Timer ISR

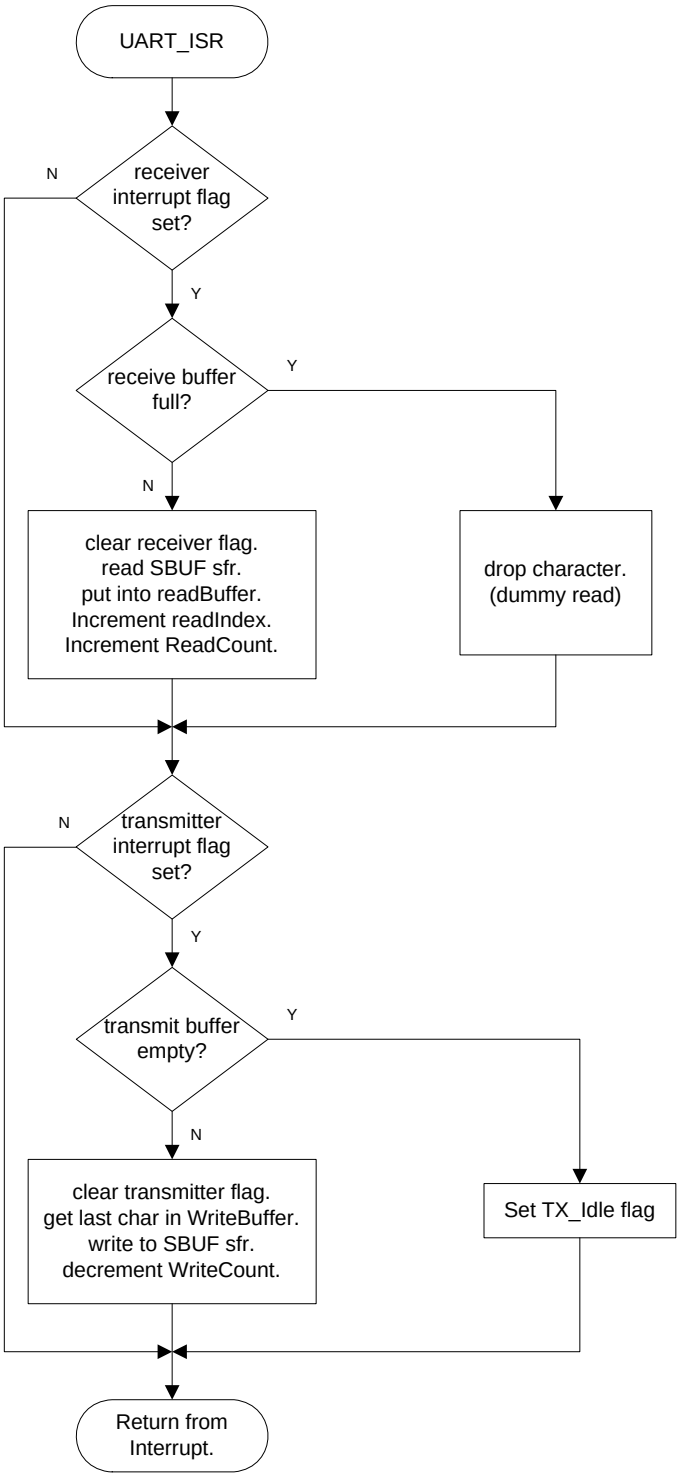


Figure D4. UART ISR

APPENDIX E - CODE LISTING

```
//-----------------------------------------------------------------------------
// Stepper.c
//-----------------------------------------------------------------------------
// Copyright 2003 Silicon Laboroatories Inc.
//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f300.h>                  // include SFR declarations
//-----------------------------------------------------------------------------
// defines and typedefs
//-----------------------------------------------------------------------------
#define SYSCLK       24500000           // SYSCLK frequency in Hz
#define BAUDRATE     57600              // Baud rate of UART in bps

#define READ_BUFFER_SIZE 16             // change UART receive buffer size here
#define WRITE_BUFFER_SIZE 16            // change UART receive buffer size here

#define ON 0                            // LED is active low
#define OFF 1
#define ACCELERATE 3                    // motor states
#define SLEWING 2
#define DECELERATE 1
#define DONE 0
#define INIT_TZERO 80                   // Tzero used at start-up and with
                                        // manual mode, change value here for
                                        // different stepper motors

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

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
//
// half step stepper motor step pattern
// transistors are in order  B-,B+,A-,A+
//
// 0x01 = 0001
// 0x05 = 0101
// 0x04 = 0100
// 0x06 = 0110
// 0x02 = 0010
// 0x0A = 1010
// 0x08 = 1000
// 0x09 = 1001

const unsigned char code StepPattern[8]=
   {0x01,0x05,0x04,0x06,0x02,0x0A,0x08,0x09};
//
```

```
// stepper motor linear velocity profile
// 255 * [sqrt(n+1)-sqrt(n)]
// n = 0 to 255
//
const unsigned char code StepTable[256]=
{
   0xFF, 0x69, 0x51, 0x44, 0x3C, 0x36, 0x32, 0x2E,
   0x2B, 0x29, 0x27, 0x25, 0x24, 0x22, 0x21, 0x20,
   0x1F, 0x1E, 0x1D, 0x1C, 0x1C, 0x1B, 0x1A, 0x1A,
   0x19, 0x19, 0x18, 0x18, 0x17, 0x17, 0x17, 0x16,
   0x16, 0x16, 0x15, 0x15, 0x15, 0x14, 0x14, 0x14,
   0x14, 0x13, 0x13, 0x13, 0x13, 0x12, 0x12, 0x12,
   0x12, 0x12, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11,
   0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
   0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F,
   0x0E, 0x0E, 0x0E, 0x0E, 0x0E, 0x0E, 0x0E, 0x0E,
   0x0E, 0x0E, 0x0E, 0x0D, 0x0D, 0x0D, 0x0D, 0x0D,
   0x0D, 0x0D, 0x0D, 0x0D, 0x0D, 0x0D, 0x0D, 0x0D,
   0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C,
   0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C,
   0x0C, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B,
   0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B,
   0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0A, 0x0A,
   0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A,
   0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A,
   0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A,
   0x0A, 0x0A, 0x0A, 0x09, 0x09, 0x09, 0x09, 0x09,
   0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
   0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
   0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
   0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
   0x09, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
   0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x07, 0x07
};
//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
//
// main and user interface functions
void SYSCLK_Init (void);               // initialize port pins and crossbar
void PORT_Init (void);                 // initialize system clock and watchdog
void Timer_Init (void);                // initialize timer for Motor and UART
void UART_Init (void);                 // initialize UART
void Motor_Init(void);                 // initialize Motor variables
void doneMsg(void);                    // display done message
void error(void);                      // display error message
void delay (void);                     // delay used for position updating
void INT0_ISR (void);                  // external interrupt used for SW2
//
// stepper motor control functions
void move (unsigned int);
void Timer_ISR (void);
unsigned int MUL8x8(unsigned char, unsigned char);
```

```
//
//  BUFFERED UART FUNCTION PROTOTYPES
//--Top Level - User Functions ------------------------------------------------

// put a string into transmit buffer, used to display text messages
void puts (char *);

// converts a 16 bit number to a string of ASCII characters
// and stores the characaters into the transmit buffer
void putuint(unsigned int);

// converts an 8 bit number to a string of ASCII characters
// and stores the characaters into the transmit buffer
void putuchar(unsigned char);

// retrieves ASCII characters from the receiver buffer
// and converts a string of digits into an 8-bit number
unsigned char getuchar(void);

// retrieves ASCII characters from the receiver buffer
// and converts a string of digits into an 16bit number
unsigned int getuint(void);

void newline(void);                    // outputs carriage return & line feed


//----------Middle Level - Primitive User Functions ----------------------------

// retrieves one character from the receive buffer using readc and echoes
// the character to the transmit buffer
char getc (void);

// stores one character into the transmit buffer using writec
void putc (char);

void readClear(void);                  // clears read buffer

void writeClear(void);                 // clears write buffer

//----------------Lowest level - Hardware access functions --------------------
char readc (void);                     // pulls one character directly
                                       // from the receive buffer

void writec (char);                    // writes one character directly
                                       // to the transmit buffer


//-----------------------------------------------------------------------------
// Global VARIABLES
//-----------------------------------------------------------------------------
//
// user interface variables
sbit LED     = P0 ^ 6;                 // bit set high to turn LED on
//
// Stepper Motor Variables
unsigned int Position;                 // current motor position
unsigned char TableIndex;              // index for stepper motor table
unsigned char MaxTableIndex;           // maximum index for desired profile
unsigned int SlewCount;                // down counter for slewing phase
```

```
unsigned char Tzero;                    // initial period, sqrt(2/alpha)
unsigned char PatternIndex;             // index for step pattern, modulus 8 counter
bit Forward;                            // 1 for forward, 0 for reverse
bit doneFlag;                           // done flag used for Timer ISR handshake
unsigned char motorState;               // motor state variable
//
// UART Global Variables

// UART receiver buffer
char idata readBuffer[READ_BUFFER_SIZE];
// UART transmitter buffer
char idata writeBuffer[WRITE_BUFFER_SIZE];
unsigned char writeIndex;               // points to next free write byte
unsigned char readIndex;                // points to next free read byte
unsigned char writeCount;               // number of bytes in write buffer
unsigned char readCount;                // number of bytes in read buffer
bit TX_Idle;                            // flag set when TX buffer is empty

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
//
// The main function initializes everything and then services the user
// interface. The user interface parses characters from the UART and
// executes basic commands. There are three interrupt service routines
// which function in the background - The Timer_ISR which controls the
// stepper motor; The UART_ISR which moves characters to/from the buffers;
// and the INT0_ISR that handles the pushbutton switch commands.
//
void main (void)
{
   char theChar;                        // the character to be parsed
   unsigned int newP;                   // new position to move to

   SYSCLK_Init ();                      // Init system clock to 24.5MHz
   PORT_Init ();                        // Init crossbar and GPIO
   Timer_Init();                        // Init T0 for stepper motor and
                                        // T1 for Baud rate
   UART_Init();                         // Init UART
   Motor_Init();                        // Init motor global variables
   EA = 1;                              // enable global interrupts

   putc('>');                           // display prompt

    while(1)                            // main loop
    {
      if(LED==ON)                       // display position while moving
      {
         putuint(Position);             // display position
         puts("    ");                  // blank out trailing characters
         putc('\r');                    // overwrite line by not using linefeed
         delay();                       // delay sets display update rate
      }
      else
      {
         if(doneFlag)
         {
            doneMsg();                  // if done display message
         }
```

```
         theChar = getc();             // get character to be parsed
         switch(theChar)               // parse character
         {
            case '\r':                 // if return character
               putc('\n');             // linefeed with no carriage return
               putc('>');              // display prompt
               break;
            case 'p':                  // p for new position
               newP = getuint();       // get number as unsigned integer
               newline();
               puts("Moving...");      // display moving status indicator
               newline();
               puts("Position:");      // display position tag
               newline();
               move(newP);             // initiate move to new position
               break;
            case 'a':                  // a for change acceleration
               Tzero = getuchar();     // get number as unsigned character
               newline();
               puts("Acceleration:");  // confirm acceleration changed
               newline();
               putuint(Tzero);         // display new acceleration
               newline();
               putc('>');
               break;
            case 's':                  // s for display status
               newline();
               puts("Position:");      // display position tag
               newline();
               putuint(Position);      // display position value
               newline();
               puts("Acceleration:");  // display acceleration tag
               newline();
               putuint(Tzero);         // display acceleration value
               newline();
               putc('>');              // display prompt
               break;
            default:
               error();                // display error message
         }
      }
    }
}

//-----------------------------------------------------------------------------
// 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.
//
void SYSCLK_Init (void)
{
   OSCICN = 0x07;                      // set SYSCLK to OSC frequency
   RSTSRC = 0x04;                      // enable missing clock detector
   PCA0MD &= ~0x40;                    // disable watchdog timer
}

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

```
// PORT_Init
//-------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - A+
// P0.1 - A-
// P0.2 - B+
// P0.3 - B-
// P0.4 - Txd
// P0.5 - Rxd
// P0.6 - LED
// P0.7 - C2D/Switch
//
void PORT_Init (void)
{
   XBR0 = 0x4F;                        // Crossbar Register 1
   XBR1 = 0x03;                        // Crossbar Register 2
   XBR2 = 0x40;                        // Crossbar Register 3
   P0 = 0xC0;                          // Turn off MOSFETs, Set P0.6 & P0.7
   P0MDOUT = 0x5F;                     // Output configuration for P0
   P0MDIN = 0xFF;                      // Input configuration for P0
   IP = 0x02;                          // T0 high priority, others low
   IT01CF = 0x70;                      // use P0.7 for INT1
   IT1 = 1;                            // edge sensitive interrupt
   IE1 = 0;                            // clear external interrupt
   EX1 = 1;                            // enable external interrupt 1
}
//-------------------------------------------------------------------------
void Timer_Init (void)
{
   CKCON = 0x12;                       // T1 uses sysclk, T0 uses /48,
   TMOD = 0x21;                        // T1 mode 2, T0 mode 1
}
//-------------------------------------------------------------------------
void UART_Init(void)
{
   SCON0   = 0x10;                     // mode 1, 8-bit UART, disable receiver
   TH1    = 0x2C;                      // set Timer1 reload value for 57600
   TR1    = 1;                         // start Timer1
   TX_Idle = 1;                        // set idle flag
   readClear();                        // zero out read buffer
   writeClear();                       // zero out write buffer
   ES0 = 1;                            // enable UART interrupt
}
//-------------------------------------------------------------------------
void Motor_Init()
{
   Tzero = INIT_TZERO;                       // initialize acceleration
   Position = 0x0000;                  // zero position
   LED = OFF;                          // turn off LED
   doneFlag = 0;                       // clear done flag
}

//-------------------------------------------------------------------------
void error (void)                      // used by main
{
   newline();
   puts("invalid character");          // display error message
   newline();
```

```
   putc('>');
}
//-----------------------------------------------------------------------------
void delay (void)                       // used by main
{
   unsigned char i,j;

   for(i=255;i>0;i--)                   // simple delay for display update
   {
      for(j=255;j>0;j--);
   }
}
//-----------------------------------------------------------------------------
void doneMsg(void)                      // called from Timer_ISR
{
   putc('\r');                          // overwrite final position
   putuint(Position);                   // display int as string
   newline();
   puts("done!");                       // indicate move complete
   newline();
   putc('>');                           // display prompt
   doneFlag=0;

}
//-----------------------------------------------------------------------------
void INT0_ISR (void) interrupt 2        // external interrupt 0 used for switch
{
      if(LED==OFF)                      // ignore switch if moving
      {
         if(Position==0)                // if home move to 100
         {
            move(1600);
         }
         else                           // if not home move to home
         {
            move(0);
         }
      }
}

//-----------------------------------------------------------------------------
// move
// This function calculates the profile and initializes the stepper
// motor. Function will abort if the target equals the Position or if
// the motor is still moving. Forward is set to 1 if target>Position.
// the length is calculated as the absolute value of Position minus
// target. For short moves less than 1024, MaxTableIndex is length
// divided by 4. For long moves, MaxTableIndex is set to 255.
//
// slewCount = length - 2 * MaxTableIndex - 1
//
// The slewcount is calculated by subtracting MaxTableIndex twice and
// decrementing. The TableIndex is initialized to zero.
//
void move (unsigned int target)
{
   unsigned int length;                 // used to calculate length of move

   if (target != Position)              // abort if already there
```

```
   {
      if (target > Position)
      {
         Forward = 1;                  // set forward flag
         length = target - Position;   // subtract smaller (position)
      }
      else
      {
         Forward = 0;                  // clear forward flag
         length = Position - target;   // subtract smaller (target)
      }
      if (length < 1024)               // if short move
         MaxTableIndex = length>>2;    // divide length by 4
      else
         MaxTableIndex = 0xff;         // else max is 255
      SlewCount = length;              // build value in SlewCount
      SlewCount -= MaxTableIndex;      // subtract MaxTableIndex twice
      SlewCount -= MaxTableIndex;
      SlewCount--;                     // Subtract one to account first step
      TableIndex = 0;                  // init table index
      motorState = ACCELERATE;               // init motor state
      TL0 = -100;                      // move starts in 100 ticks
      TH0 = 0xFF;                      // extend sign
      LED = ON;                        // turn on LED
      ET0 = 1;                         // enable Timer0 interrupts
      TR0 = 1;                         // start Timer0
   }
      else
   {
      doneFlag=1;                       // if done display message
   }
}

//-------------------------------------------------------------------------
// Timer_ISR()
// This is the timer interrupt service routine for the stepper motor.
// First the PatternIndex and Position are incremented or decremented
// depending on the direction of the motor. Then the new pattern is
// output to the stepper motor. Nested if..else statements are used to
// determine the if the motor is in the acceleration, slewing, or
// deceleration phase. The TableIndex and SlewCount are modified
// accordingly. When the move is complete, further output compares and
// interrupts are disabled.
//
void Timer_ISR (void) interrupt 1
{
   unsigned char TableValue;
   unsigned int offset;
   udblbyte time;

   if (Forward)                        // if forward
   {
      PatternIndex++;                  // increment step pattern
      PatternIndex &= 0x07;            // fix modulus 8 counter
      Position++;                      // increment Position
   }
   else
   {
      PatternIndex--;                  // decrement step pattern
```

```
      PatternIndex &= 0x07;            // fix modulus 8 counter
      Position--;                      // increment Position
   }
   // output step pattern, set bit 7 because it is an input
   P0 = StepPattern[PatternIndex] | 0x80;

   // determine motor state based on counter values
   if (SlewCount == 0)
      if (TableIndex == 0)
         motorState = DONE;
      else
         motorState = DECELERATE;
   else
      if (TableIndex < MaxTableIndex)
         motorState = ACCELERATE;
      else
         motorState = SLEWING;

   if (motorState == DONE)
   {
      ET0     = 0;                     // disable T0 interrupts
      TR0     = 0;                     // stop T0
      LED = OFF;                       // turn off LED
      doneFlag = 1;                    // display done message
   }
   else
   {
      // get value from table, multiply by T zero
      TableValue = StepTable[TableIndex];
      offset = MUL8x8(Tzero, TableValue);
      TR0 = 0;                         // stop Timer0
      time.b.lo = TL0;                 // read lo byte first
      time.b.hi = TH0;                 // read hi byte second
      time.w = time.w - offset;        // calculate new time
      TL0 = time.b.lo;                 // write lo byte first
      TH0 = time.b.hi;                 // write hi byte second
      TR0 = 1;                         // start Timer0
     if (motorState == DECELERATE)     // if decelerating
         TableIndex--;                 // decrement table index
      else if(motorState == ACCELERATE)// if accelerating
         TableIndex++;                 // increment table index
      else if (motorState == SLEWING)  // if slewing
         SlewCount--;                  // decrement slewCount
      ET0     = 1;                     // enable Timer0 interrupts
   }
}

//-----------------------------------------------------------------------------
unsigned int MUL8x8(unsigned char a, unsigned char b)
{
   unsigned int ab;
   ab = (unsigned int) a * b;          // cast a to int to trick compiler
   return ab;                          // return int
}
//-----------------------------------------------------------------------------
void puts(char *string)                // puts a string into send buffer
{
   while(*string)                      // while not null character
   {
```

```
      putc(*string);                  // put character at pointer in buffer
      string++;                       // increment pointer
   }
}
//-----------------------------------------------------------------------------
unsigned int getuint(void)             // get string and convert to int
{
   char theChar;
   unsigned int i;

   i=0;                                // build value in i
   theChar=getc();                     // get next character
   while( theChar<'0' || theChar>'9')  // while not 0-9
      theChar=getc();
   while(theChar>='0' && theChar<='9') // while 0-9
   {
      theChar -= '0';                  // convert from ASCII
      i *= 10;                         // shift decimal point
      i += theChar;                    // add next digit
      theChar=getc();
   }
   return i;                           // return int value
}
//-----------------------------------------------------------------------------
void putuint(unsigned int i)
{
   char string[7];                     // string used to build output
   char *sp;                           // string pointer

   sp=string + 6;                      // build back to front
   *sp=0;                              // insert null character
   do
   {
      sp--;                            // predecrement pointer
      *sp=i%10 + 0x30;                 // take modulus add 30
      i/=10;                           // divide i by 10
   } while (i);                        // while i not zero
                                       // now output front to back
   while (*sp)                         // while not null character
   {
      putc(*sp);                       // put character in buffer
      sp++;                            // increment pointer
   }
}
//-----------------------------------------------------------------------------
unsigned char  getuchar ()             // same as getuint but returns uchar
{
   char theChar;
   unsigned char i;
   i=0;                                // build value in i
   theChar=getc();                     // get next character
   while( theChar<'0' || theChar>'9')  // while not 0-9
      theChar=getc();
   while(theChar>='0' && theChar<='9') // while 0-9
   {
      theChar -= '0';                  // convert from ASCII
      i *= 10;                         // shift decimal point
      i += theChar;                    // add next digit
      theChar=getc();
```

```
   }
   return i;                           // return uchar value
}
//-----------------------------------------------------------------------------
void newline(void)                     // normally cr and lf are used together
{
   putc('\r');                         // output carriage return
   putc('\n');                         // output linefeed
}
//-----------------------------------------------------------------------------
// getc
// Gets a character from the read buffer using readc().
// The getc() function also echos the incoming keystrokes to the display;
//
char getc(void)
{
   char theChar;
   theChar=readc();                    // get character using readc
   writec(theChar);                    // echo characters to display
   return theChar;
}
//-----------------------------------------------------------------------------
// putc
// This is a totally unnecessary layer of abstraction. It is only used to be
// consistent with the getc function which requires an additional layer of
// abstraction to handle character echo.
//
void putc(char theChar)
{
   writec(theChar);
}
//-----------------------------------------------------------------------------
void readClear(void)                   // clears read buffer
{
   unsigned char i;
   readCount=0;
   readIndex=0;
   i = READ_BUFFER_SIZE;
   do
   {
      i--;                             // predecrement
      readBuffer[i]=0;                 // zero all data
   } while (i != 0);
}
//-----------------------------------------------------------------------------
void writeClear(void)                  // clears write buffer
{
   unsigned char i;
   writeCount=0;
   writeIndex=0;
   i = WRITE_BUFFER_SIZE;
   do
   {
      i--;                             // predecrement
      writeBuffer[i]=0;                // zero all data
   } while (i != 0);
}

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

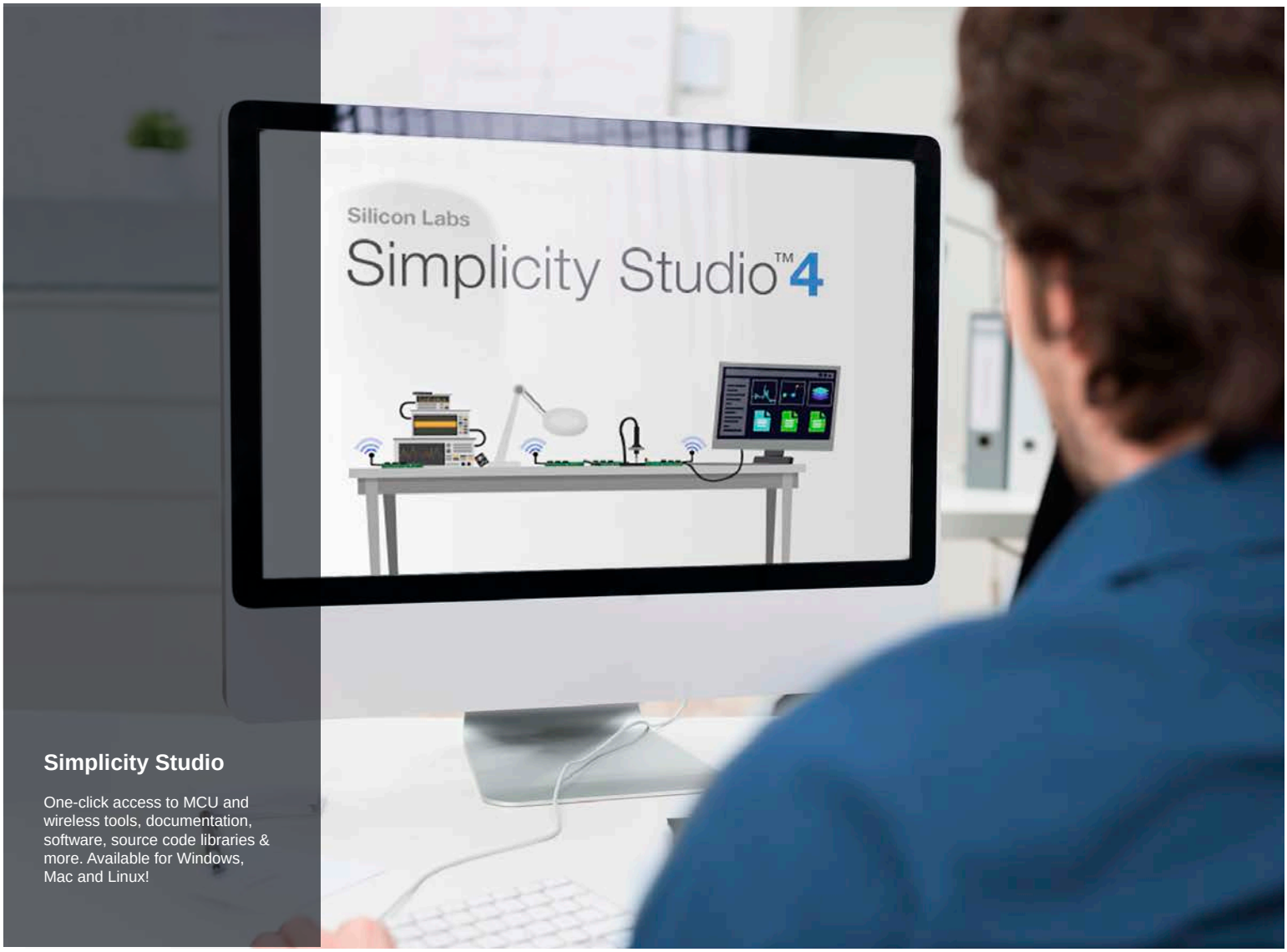
```
// readc()
//
// The readc() function is the lowest level function which provides
// direct access to the read buffer. It reads one character from the
// read buffer.
//
// Note that readc will wait if the read buffer is empty. This is usually
// desired if the program is waiting for user input.
//
// readc is a driver function and is closely integrated with the ISR.
// UART interrupts are temporarily disabled while the buffer is updated,
// This prevents the ISR from modifying the buffer and counter values
// during processing. That would be bad.

char readc(void)
{
   char theChar;                        // the character to return
   signed char i;                       // signed value to build location
   while (readCount==0);                // wait here if buffer empty [blocking]
   ES0 = 0;                             // disable UART interrupts
   i = readIndex - readCount;           // back up by readcount
   if (i < 0)                           // fix value if out of range
      i+=READ_BUFFER_SIZE;
   theChar = readBuffer[i];             // get character from read buffer
   readCount--;                         // one less character in the buffer
   ES0 = 1;                             // enable UARt interrupt
   return theChar;                      // return the character
}
//-----------------------------------------------------------------------------
//
// writec()
//
// The writec() function is the lowest level function which allows access
// to the write buffer. It writes one character to the write buffer.
//
// Note that writec will wait if the write buffer is full. This is usually
// desired to prevent the write buffer from overflowing.
//
// writec is a driver function and is closely integrated with the ISR.
// UART interrupts are temporarily disabled while the buffer is updated to
// prevent the ISR from modifying the buffer and counter values during
// processing. That would be bad.

void writec(char theChar)
{
   // wait here if full [blocking]
   while (writeCount >= WRITE_BUFFER_SIZE);
   ES0 = 0;                             // disable UART interrupts
   writeBuffer[writeIndex] = theChar;   // put character in buffer at writeIndex
   writeIndex++;                        // increment index
   if (writeIndex >= WRITE_BUFFER_SIZE)// fix if out of range
      writeIndex = 0;
   writeCount++;                        // one more character in buffer
   if(TX_Idle)                          // if transmitter idle flag set
   {
      TI0 = 1;                          // force TX interrupt
      TX_Idle = 0;                      // idle no more
   }
   ES0 = 1;                             // enable UART interrupts
```

```
}

//-----------------------------------------------------------------------------
void uartISR(void) interrupt 4         // main UART interrupt service routine
{
   char dummy;
   signed char i;
                                       // Who done it?
   if(RI0 == 1)                        // Was it the receiver?
   {
      RI0 = 0;                         // yep, clear receiver flag
      // if not full
      if (readCount != READ_BUFFER_SIZE)
      {
         readBuffer[readIndex]= SBUF0; // read char from UART
         readIndex++;                  // increment index
         if (readIndex == READ_BUFFER_SIZE)
            readIndex = 0;             // fix if out of range
         readCount++;                  // one more in the buffer
      }
      else
      {
         dummy = SBUF0;                // drop characters dummy!
      }
   }
   if(TI0 == 1)                        // Was it the transmitter?
   {
      TI0 = 0;                         // yep, clear transmitter flag
      if (writeCount>0)                // if not empty
      {
         i = writeIndex - writeCount;  // calculate where to get it
         if (i < 0)                    // fix if out of range
            i+=WRITE_BUFFER_SIZE;
         SBUF0 = writeBuffer[i];       // write character to UART
         writeCount--;                 // one less in the buffer
      }
      else
      {
      TX_Idle = 1;                     // set idle flag when empty
      }
   }
}
//-----------------------------------------------------------------------------
```



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

Disclaimer
Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

Trademark Information
Silicon Laboratories Inc.® , Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOmodem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.

SILICON LABS

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

http://www.silabs.com