INTRODUCTION

The PICmicro® microcontroller makes an ideal choice for an embedded DC Servomotor application. The PICmicro family has many devices and options for the embedded designer to choose from. Furthermore, pin compatible devices are offered in the PIC16CXXX and PIC18CXXX device families, which makes it possible to use either device in the same hardware design. This gives the designer an easy migration path, depending on the features and performance required in the application. In particular, this servomotor has been implemented on both the PIC18C452 and PIC16F877 devices, and we'll look at the MCU resources required to support the servomotor application. With an understanding of the servomotor functions, you can start with the design shown here and implement your own custom DC servomotor application based on the PICmicro device that suits your needs.

The PICmicro MCU handles many functions in the servomotor application, such as:

- User control interface
- Measurement of motor position
- Computation of motion profile
- Computation of error signal and PID compensation algorithm
- Generation of motor drive signal
- Communication with non-volatile EEPROM memory

HARDWARE

A Pittman Inc. 9200 Series DC motor was used to develop the application source code. The motor was designed for a 24 Vdc bus voltage and has a no-load speed of 6000 RPM. The torque constant ($K_T$) for the motor is 5.17 oz-in/A and the back-EMF constant ($K_E$) is 3.82 V/kRPM. This motor has an internal incremental encoder providing a resolution of 500 counts-per-revolution (CPR). In practice, the design should be compatible with almost any brush-DC motor fitted with an incremental encoder.

A schematic diagram for the application is shown in Figure 1. The DC motor is driven by a SGS-Thomson L6203 H-bridge driver IC that uses DMOS output devices and can deliver up to 3 A output current at supply voltages up to 52 V. The device has an internal charge pump for driving the high-side transistors and dead-time circuitry, to prevent cross-conduction of the output devices. Each side of the bridge may be driven independently and the inputs are TTL compatible. An enable input and automatic thermal shutdown are also provided. A transient voltage suppressor is connected across the motor terminals to prevent damage to the L6203.

The PWM1 output from the MCU is connected to both sides of the H-bridge driver IC with one side of the bridge driven with an inverted PWM signal. You get more switching losses when the bridge is driven in this manner, because all four devices in the bridge are switched for each transition in the PWM signal. However, this arrangement provides an easy method of bidirectional control with a single input signal. For example, a 50% PWM duty cycle delivered to the H-bridge produces zero motor torque. A 100% duty cycle will produce maximum motor torque in the forward direction, while a 0% duty cycle will produce maximum motor torque in the opposite direction. The only other control signal is an enable input that turns the output of the H-bridge driver IC on or off.

The quadrature pulse outputs from the encoder are connected through external pull-up resistors. The outputs are then filtered and decoded into up and down pulse trains with a 74HC74 dual D flip-flop. Figure 2 shows a timing diagram, indicating the output of the decoder circuit for each direction of the motor. It's possible to decode the encoder outputs so that an output pulse is generated for every transition of the encoder output signals, which yields a 4x increase in the specified encoder resolution. However, the 1x decoding circuit is implemented here for simplicity (see Figure 1). The up and down pulse outputs from the D flip-flops are connected to the Timer0 and Timer1 clock inputs, respectively. This method of decoding the motor position is beneficial, because it requires low software overhead. The cumulative forward and reverse travel distances are maintained by the timers, while the MCU is performing other tasks.
FIGURE 1: DC SERVOMOTOR SCHEMATIC DIAGRAM

FIGURE 2: DECODER CIRCUIT TIMING DIAGRAM
The primary user interface is a RS-232 connection to a host PC. A Dallas Semiconductor DS275 transceiver is used in the design. This IC supports half-duplex communication and steals power from the host device for generating the transmit voltages required for the RS-232 standard. Four DIP switches are also included in the circuit and are connected to PortB<7:4>. These switches are optional for the design and are used by the software to activate motion profiles when no host PC is available.

The LEDs shown in the schematic diagram are used to implement a bar graph display used by the firmware to indicate the percentage of the MCU bandwidth used by the servo calculations. To measure the bandwidth, I/O pin RC5 is toggled high when the servo calculations begin, and toggled low when they are completed. The resulting output is filtered to create a DC voltage proportional to the bandwidth used and is connected to Channel 0 of the A/D converter. The LEDs and filter circuit are not essential, and can be removed from the application if desired.

SOFTWARE

The servomotor software performs the servo position calculations and provides a command interpreter to create and control motion profiles.

Servo Calculations

The entire servomotor function is implemented in the Interrupt Service Routine (ISR), which must perform the following tasks:

- Get current motor position
- Get desired motor position
- Find the position error
- Determine new PWM duty cycle

Timer2, the timebase for the CCP1 module, is used to generate interrupts that time the servo calculations. This ensures that the PWM duty cycle changes are synchronous with the PWM period.

The frequency of the PWM signal that drives the motor should be high enough so that a minimal amount of current ripple is induced in the windings of the DC motor. The amount of current ripple can be derived from the PWM frequency, motor winding resistance, and motor inductance. More importantly, the PWM frequency is chosen to be just outside the audible frequency range. Depending on how much hearing loss you’ve suffered, a PWM frequency in the 15 kHz - 20 kHz range will be fine. There's no need to set the PWM frequency any higher; this will only increase the switching losses in the motor driver IC. For this application, the MCU is operated at 20 MHz and the PWM frequency is 19.53 kHz. At this PWM frequency, a Timer2 interrupt would occur every 51 μsec. The servo calculations do not need to be performed this often, so the Timer2 postcaler is used to set the Timer2 interrupt rate. Using the postcaler, interrupts may be generated at any frequency from 1/2 to 1/16th the PWM frequency.

Position Updates

The first task to be done in the servo calculations is to determine exactly where the motor is at the present moment. The function UpdPos() is called to get the new motor position. As mentioned earlier, Timer0 and Timer1 are used to accumulate the up and down pulses that are derived from the encoder output signals. The counters are never cleared to avoid the possibility of losing count information. Instead, the values of the Timer0 and Timer1 registers saved during the previous sample period are subtracted from the present timer values, using two’s-complement signed arithmetic. This calculation provides us with the total number of up and down pulses accumulated during the servo update period. The use of two’s-complement arithmetic, also accounts for a timer overflow that may have occurred since the last read. The down pulse count, DnCount, is then subtracted from UpCount, the up pulse count, which provides a signed result indicating the total distance (and direction) traveled during the sample period. This value also represents the measured velocity of the motor in encoder counts per servo update period and is stored in the variable mvelocity.

The measured position of the motor is stored in the variable mposition. The upper 24 bits of mposition holds the position of the motor in encoder counts. The lower 8 bits of mposition represent fractional encoder counts. The value of mvelocity is added to mposition to find the new position of the motor. With 24 bits, the absolute position of the motor may be tracked through 33,554 shaft revolutions using a 500 CPR encoder. If you need to cover greater distances with the motor, the size of mposition can be increased as needed.

Trajectory Updates

Now that we know where the motor is, we need to determine where the motor is supposed to be. The commanded motor position is stored in the variable position. The size of position is 32 bits with the lower 8 bits representing fractional encoder counts. When the value of position is constant, the motor shaft will be held in a fixed position. We can also have the servomotor operate at a given velocity by adding a constant value to position at each servo update. The fractional bits in position allow the motor to be operated at very low velocities. In order for the servomotor to produce smooth motion, we need a motion profile algorithm that controls the speed and acceleration of the motor. In the context of this application, we must control the rate at which position is changed. The UpdTraj() function does this job and its purpose is to determine the next required value for position, based on the current motion profile parameters. For this application, a movement distance, velocity limit, and acceleration value are required to execute the profile. From this data, the servomotor will produce trapezoidal shaped velocity curves.
Figure 3 shows a flowchart of the UpdTraj() function. If the motion profile is running and the PWM output is not saturated, indicated by the stat.run and stat.saturated flags, the motion profile algorithm will find the next value for position. Figure 4 shows a flowchart of the motion profile operation.

**FIGURE 3: UpdTraj() FLOWCHART**
FIGURE 4: MOTION PROFILE FLOWCHART

START

IN PHASE 1 OF MOVE?

YES

HAS VELOCITY LIMIT BEEN REACHED?

NO

INCREASE FLAT COUNT

ACCELERATE

YES

DECELERATE

DECREASE FLAT COUNT

SET COMMANDED POSITION EQUAL TO CALCULATED FINAL POSITION

CLEAR MOTION FLAG

END

IN PHASE 1 OF MOVE?

YES

HAS VELOCITY LIMIT BEEN REACHED?

NO

INCREASE FLAT COUNT

ACCELERATE

YES

DECELERATE

DECREASE FLAT COUNT

SET COMMANDED POSITION EQUAL TO CALCULATED FINAL POSITION

CLEAR MOTION FLAG

END

IS FLAT COUNT 0?

YES

CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS FLAT COUNT 0?

NO

CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?

YES

IS CURRENT VELOCITY 0?

NO

IS CURRENT VELOCITY 0?
The motion profile is executed in two phases. The first half of the movement distance is traveled in the first phase and the remaining distance in the second phase. The stat.phase flag indicates the current phase of the motion segment. Half of the total distance to be traveled is stored in the variable phaseldist. The final destination position for the motor is stored in fposition.

The velocity limit for the motion profile is stored in the variable vlim. The present commanded velocity of the motor is stored in velact. The acceleration value for the profile is stored in accel. A delay time for the motion profile is stored in the variable dtime. This variable tells the motion profile how many servo update periods to wait before executing the next motion segment. Finally, the direction of motion is set by the stat.neg_move flag.

Once the variables used for the motion profile have been loaded, the stat.motion flag is set and motion begins on the next servo update. This flag is cleared when the motion profile has completed.

The motor can be run at any desired speed by adding a constant value to position at each servo update, forcing the servomotor to track the new commanded position. The value added to position at each servo update is stored in the variable velact. Furthermore, the motor will accelerate (or decelerate) at a constant rate, if we add or subtract a value to velact at each servo update. The acceleration value for the profile is stored in the variable accel. The value of accel is added to velact at each servo update. The value of velact is then added or subtracted from the commanded motor position, position, depending on the state of the stat.neg_move flag. The value of velact is also subtracted from phaseldist to keep track of the distance traveled in the first half of the move. The motor stops accelerating when velact is greater than vlim. After the velocity limit has been reached, flatcount is incremented at each servo update period to maintain the number of servo updates for which no acceleration occurred.

The first half of the move is completed when phaseldist becomes zero or negative. At this time, the stat.phase flag is set to 1. The variable flatcount is then decremented at each servo period. When flatcount = 0, the motor begins to decelerate. The move is complete when velact = 0. The motion profile then waits the number of sample periods stored in dtime. When dtime is 0, the previously calculated destination in fposition is written to the commanded motor position and the stat.motion flag is cleared to indicate the motion profile has completed.

When the motion profile is completed, the UpdTraj() function checks the present motion segment value in segnum to see if another motion segment should be executed. The first and last motion segments to be executed are stored in firstseg and lastseg, respectively. If segnum is not equal to lastseg, then segnum is incremented and the SetupMove() function is called to load the new segment parameters into the motion profile variables.

Error Calculation
The CalcError() function subtracts the measured motor position, mposition, from the commanded motor position in the variable position, to find the amount of position error. The position error result is shifted to the right and the lower 8 bits that hold fractional data are discarded. This leaves the 24-bit position error result in u0, which is then truncated to a 16-bit signed value for subsequent calculations.

Duty Cycle Calculations
The CalcPID() function implements a proportional-integral-derivative (PID) compensator algorithm and uses the 16-bit error result in u0 to determine the next required PWM duty cycle value. The PID gain constants, kp, ki, and kd, are stored as 16-bit values.

The proportional term of the PID algorithm provides a system response that is a function of the immediate position error, u0. The integral term of the PID algorithm accumulates successive position errors, calculated during each servo loop iteration and improves the low frequency open-loop gain of the servo system. The effect of the integral term is to reduce small steady-state position errors.

The differential term of the PID algorithm is a function of the measured motor velocity, mvelocity, and improves the high frequency closed-loop response of the servo system.

After the three terms of the PID algorithm are summed, the 32-bit result stored in ypid is saturated to 24 bits. The upper 16 bits of ypid are used to set the duty cycle, which effectively divides the output of the PID algorithm by 256. Since the PWM module has a 10-bit resolution, the value in the upper 16 bits of ypid is checked to see if it exceeds +511 or -512. When this condition occurs, the PWM duty cycle is set to the maximum positive or negative limit and the stat.saturated flag is set.

The commanded position will not be updated by UpdTraj() when the PWM output becomes saturated. In addition, the integral accumulation in the PID algorithm is bypassed. This allows the servomotor to smoothly resume motion when the saturation condition ends. If the integral error and the motion profile continued to update, the servomotor would produce sudden and erratic motions when recovering from a mechanical overload.
Command Interpreter

The servomotor software has a command interpreter that allows you to enter motion profile segment data, run motion profiles, and change the PID gain constants. After all peripherals and data memory have been initialized, the main program loop polls the USART interrupt flag to detect incoming ASCII data. Each incoming byte of data is stored in inpbuf[] as it is received. A comma or a <CR> is used to delimit each command sequence and the DoCommand() function is called each time either of these characters are received to determine the correct response. The DoCommand() function can tell which portion of the command is in inpbuf[] by comcount, which holds the number of commas received since the last <CR>. A flowchart of the command interpreter operation is given in Figure 5 and Figure 6.

FIGURE 5: COMMAND INTERPRETER FLOWCHART
Motion profile segment data is stored as a 2 dimensional array of integer values in data memory. The data for each motion profile segment consists of a movement distance, acceleration value, velocity limit, and delay time. The software, as written, permits data for up to 24 motion profile segments to be entered and stored.

However, the number of segments may be increased or decreased depending on the available memory resources in your application.

The ASCII string required to change motion profile segment parameter, consists of the parameter, the segment number, and the data. For example, let's assume
that you wish to change the acceleration value for segment 2 to 1000. To do this, you would send the following ASCII string to the servomotor:

\[ A,2,1000 <CR> \]

This syntax can be used to change all motion profile segment parameters.

After all motion profile data has been entered, a single motion profile segment, or range of segments, may be executed using the 'G' or 'L' command. To execute segments 1 through 4, for example, you would send the following ASCII string to the servomotor:

\[ G,1,4 <CR> \]

If you only want to run one motion segment, the desired segment number is entered twice as shown:

\[ G,1,1 <CR> \]

The 'L' command is used the same way as the 'G' command, except that the range of motion segments is executed repeatedly. This command is useful for creating repetitive motions with the servomotor. The 'S' command stops the motion profile after the presently executing motion segment has completed.

Three commands are available to change the PID gain constants. With these commands, you can manually tune the PID algorithm to obtain the best performance from the motor in your application.

The 'W' command turns the motor driver IC on or off.

A summary of all servomotor commands and their syntax is given in Table 1.

Two status flags, stat.run and stat.loop, are used to control execution of the motion profile. If a 'G' command is entered to run a series of motion segments, the stat.run flag is set. If a 'L' command is entered, the stat.run and the stat.loop flags are set. When the SetupMove() function determines that the last segment in the motion profile has executed, the stat.loop flag is checked. If stat.loop is set, the motion profile segment data for the first segment in the sequence is loaded and execution continues. If stat.loop is clear, then the stat.run flag is cleared and motion stops.

**Operation With ASCII Terminal**

You can use a PC terminal program, such as PROCOMM® or HyperTerminal®, to control the servomotor. The terminal program should be configured for 19.2 kBaud, no parity, 8 data bits, and 1 stop bit. When the servomotor is reset, you will see an introduction message and a 'READY>' prompt. You should now be able to enter any of the commands shown in Table 1.

---

**TABLE 1: SERVOMOTOR COMMAND SUMMARY**

<table>
<thead>
<tr>
<th>Command: X,seg#,data &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets the distance to be travelled for the specified motion profile segment. Data is provided in encoder counts relative to the present position.</td>
</tr>
<tr>
<td>[ 0 \leq \text{seg} # \leq 23 ]</td>
</tr>
<tr>
<td>[ -32768 \leq \text{data} \leq 32767 ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: A,seg#,data &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets the acceleration for the specified motion profile segment. Data is provided in encoder counts/( T_{\text{SERVO}} / 2 / 65536 ).</td>
</tr>
<tr>
<td>[ 0 \leq \text{seg} # \leq 23 ]</td>
</tr>
<tr>
<td>[ 1 \leq \text{data} \leq 32767 ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: V,seg#,data &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets the velocity limit for the specified motion profile segment. Data is provided in encoder counts/( T_{\text{SERVO}} / 256 ).</td>
</tr>
<tr>
<td>[ 0 \leq \text{seg} # \leq 23 ]</td>
</tr>
<tr>
<td>[ 1 \leq \text{data} \leq 32767 ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: T,seg#,data &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the amount of time to wait before executing the next motion profile segment. Data is provided in ( T_{\text{SERVO}} ) multiples.</td>
</tr>
<tr>
<td>[ 0 \leq \text{seg} # \leq 23 ]</td>
</tr>
<tr>
<td>[ 0 \leq \text{data} \leq 32767 ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: G,startseg,stopseg &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executes a range of motion profile segments.</td>
</tr>
<tr>
<td>[ 0 \leq \text{startseg} \leq 23 ]</td>
</tr>
<tr>
<td>[ 0 \leq \text{stopseg} \leq 23 ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: S &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stops execution of a motion profile.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: P,data &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changes the proportional gain for the PID algorithm.</td>
</tr>
<tr>
<td>[ -32768 \leq \text{data} \leq 32767 ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: I,data &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changes the integral gain for the PID algorithm.</td>
</tr>
<tr>
<td>[ -32768 \leq \text{data} \leq 32767 ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: D,data &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changes the differential gain for the PID algorithm.</td>
</tr>
<tr>
<td>[ -32768 \leq \text{data} \leq 32767 ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: W &lt;CR&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enables or disables the PWM driver stage.</td>
</tr>
</tbody>
</table>
Stand-alone Operation

The provided application firmware allows the servomotor to perform a few basic motions without a PC connected. Specifically, data for three different motion profiles are stored in the MCU program memory and are loaded into data memory at start-up. Each profile is selected by turning on DIP switch #2, #3, or #4, connected to PORTB and pressing the MCLR button. The software polls the DIP switches once, at start-up, to see if a profile should be executed. The selected profile will begin to execute immediately. If DIP switch #1 is turned on in combination with one of the other switches, the selected profile will execute repeatedly.

PICmicro MCU RESOURCES

There is a broad range of PICmicro devices that can be used to implement the servomotor application, depending on the level of performance that you need. To begin with, let's consider the processing time needed by the servo calculations.

A large amount of time is spent in the servo calculations executing the compensator, which requires one or more multiplications depending on the type of algorithm used. Three 16 x 16 signed multiplications are required by the PID compensator algorithm used here. Since the servo update calculations must be performed frequently, a hardware multiplier can provide a significant reduction in the MCU bandwidth. With a 8 x 8 hardware multiplier, each 16 x 16 multiplication can be performed in approximately 32 instruction cycles. Without the hardware multiplier, each multiplication can take 500 instruction cycles or more, depending on the algorithm that is used.

The servo calculation times were compared for the PIC16CXXX and PIC18CXXX architectures, using the same source code. Table 2 shows the performance results. You can easily see the increase in available bandwidth gained by the hardware multiplier. For a given servo update period, the hardware multiplier in the PIC18CXXX architecture frees a large amount of MCU bandwidth for performing other tasks. In addition, extra MCU bandwidth may be obtained from the PIC18CXXX architecture, since the devices may be operated up to 40 MHz.

Table 3 and Table 4 show a comparison of memory usage by the servomotor application, for both the 16F877 and the 18C452. Depending on the memory requirements for motion profile segment data and other application functions, the design may be adapted for other MCUs. As an example of a minimal implementation, this application could be modified to operate on a PIC16C73B. The PIC16C73B has 22 I/O pins, 4K x 14 words of program memory, and 192 bytes of data memory.

The resolution of the available timer resources must be considered when using the position sensing method described here. The maximum RPM of the servomotor is a function of the timer resolution, servo update frequency, and the resolution of the incremental encoder. Because two’s complement arithmetic is used to find the motor position, the timers used to accumulate the encoder pulses should not increment more than $2^{N-1}$ counts during each servo update interval, or position information will be lost.

When a PIC16CXXX device is used for the servomotor application, Timer0 and Timer1 are the only timers with an external clock input and Timer0 has only 8 bits of resolution. For some cases, this may limit the maximum motor RPM. A formula that can be used to calculate the maximum RPM is given in Equation 1 below:

**EQUATION 1: MAXIMUM RPM**

\[
RPM_{\text{Max}} = \frac{2^{N-1} \cdot f_s \cdot 60}{\text{CPR}}
\]

In this equation, $N$ represents the resolution of the timer in bits, $f_s$ is the servo update frequency, and CPR is the resolution of the encoder. The incremental encoder used in this application provides 500 CPR. For the moment, let’s assume that our servo update frequency is 1000 Hz. Using 1x decoding, the maximum RPM that we can permit without a timer overflow, is over 15,000 RPM. This maximum limit is of no concern for us, since the motor we are using provides a no-load speed of 6000 RPM. Now, let’s assume that a 4x decoding method was used, so that our encoder now provides 2000 CPR. Now, the maximum motor speed is 3840 RPM, which is definitely a problem! In this case, the servo update frequency would have to be increased, if possible, or the encoder resolution decreased.
Two source code listings are provided with this application note. The source code given in Appendix A was written for the MPLAB®-C18 compiler and will operate on a PIC18C452 or PIC18C442. If the LEDs are omitted from the design, the code may also be compiled to operate on the PIC18C242 or the PIC18C252, which are 28-pin devices with fewer I/O pins. The PIC18C452 application utilizes the MSSP peripheral to read and write data to a 24C01 serial EEPROM.

The source code given in Appendix B was written for the HiTech PICC compiler and will operate on the PIC16F877. In particular, the 16F877 source code was written to utilize the on-chip data EEPROM to store profile data and PID gain values. These routines may be omitted, if you wish to compile the source code for other devices in the PIC16CXXX family. Like the PIC18C452 source code, the LEDs may be removed for operation on a lower pin-count device.

A sufficient amount of MCU bandwidth is available when the servomotor application is implemented using the PIC18C452. This additional bandwidth could be used for a variety of purposes, depending on the requirements of the application. One use of the available MCU bandwidth takes advantage of the PIC18CXXX family architecture, reduces external hardware, and permits two servomotors to be controlled by the same device.

Although the hardware-based solution for decoding the quadrature encoder signals requires minimal software overhead, it does require that two timers be used for each encoder. There are not enough timer resources available on the PIC18C452 device to permit two encoders to be decoded. However, it is possible to decode the encoder outputs directly in software. Furthermore, the two priority level interrupt structure of the PIC18CXXX architecture, provides an elegant way to handle the software decoding algorithm. Using priority interrupts, the servo calculations are performed in the low priority ISR and the decoding algorithm is performed in the high priority ISR. The high priority ISR is able to override a low priority interrupt that is in progress. This is important in this case, because the encoder pulses can have short durations and must be processed quickly.

### GOING FURTHER...

**Table 2:** SERVO CALCULATION BANDWIDTH COMPARISON

<table>
<thead>
<tr>
<th>Device</th>
<th>Operating Frequency</th>
<th>Hardware Multiplier</th>
<th>Servo Update Period</th>
<th>Maximum Servo Calculation Time</th>
<th>MCU Bandwidth Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16CXXX</td>
<td>20 MHz</td>
<td>No</td>
<td>563 µsec</td>
<td>540 µsec</td>
<td>96%</td>
</tr>
<tr>
<td>PIC18CXXX</td>
<td>20 MHz</td>
<td>Yes</td>
<td>563 µsec</td>
<td>97 µsec</td>
<td>17%</td>
</tr>
</tbody>
</table>

**Table 3:** PROGRAM MEMORY RESOURCES FOR SERVOMOTOR APPLICATION

<table>
<thead>
<tr>
<th>Device</th>
<th>Available Program Memory</th>
<th>Program Memory Used by Application</th>
<th>Percentage Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16F877</td>
<td>8192 x 14</td>
<td>3002 x 14</td>
<td>37%</td>
</tr>
<tr>
<td>PIC18C452</td>
<td>32768 x 8</td>
<td>8265 x 8</td>
<td>25%</td>
</tr>
</tbody>
</table>

**Table 4:** DATA MEMORY RESOURCES FOR SERVOMOTOR APPLICATION

<table>
<thead>
<tr>
<th>Device</th>
<th>Total Data Memory Available</th>
<th>Data Memory Used by Application</th>
<th>Data Memory Used by Compiler(1)</th>
<th>Available Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16F877</td>
<td>368 bytes</td>
<td>78 bytes</td>
<td>44 bytes</td>
<td>246 bytes</td>
</tr>
<tr>
<td>PIC18C452</td>
<td>1536 bytes</td>
<td>78 bytes</td>
<td>386 bytes</td>
<td>1072 bytes</td>
</tr>
</tbody>
</table>

---

**Note 1:** The amount of data memory will depend on the compiler used. This memory is used for a software stack, temporary variable storage, etc.
Figure 7 shows a flowchart of a software algorithm that performs a 1x decode of the quadrature encoder pulses. In addition, a possible connection diagram for a two-servomotor solution is shown in Figure 8. One of the encoder output signals is connected to an external interrupt pin on the MCU and the other is connected to an unused I/O pin. The external interrupt source is configured to provide an interrupt on each rising edge of the incoming signal. Each time an interrupt occurs, the state of the other encoder output is checked. An encoder count value is maintained in software and is incremented or decremented depending on the state of the encoder signal. The resulting count value is proportional to the motor velocity and direction, and is added to the measured position of the motor each time `UpdPos()` is executed.

The software decoding algorithm could be extended to provide a 2x or 4x decoding, if desired. For 2x decoding, the interrupt edge bit should be toggled each time an interrupt occurs, so interrupts occur on both the rising and falling edges of the encoder signal. For 4x decoding, the second encoder signal is connected to a second external interrupt pin. In this case, interrupts are generated on every transition of the encoder signals. You must be careful though, since the amount of MCU bandwidth needed for higher decode resolutions can become very high. For example, a 6000 RPM motor and 500 CPR encoder will produce interrupts every 5 µsec, when a 4x decoding algorithm is implemented at full speed. Considering that the maximum device frequency is 40 MHz, the MCU will be able to perform 50 instruction cycles between each encoder interrupt. In this case, the number of instruction cycles required to implement the software decoding algorithm will become very critical.

CONCLUSION

We have seen that the PIC18CXXX and PIC16CXXX architecture families can be used to implement an effective DC servomotor application. The source code and hardware solutions presented here can be applied to a range of devices in both families, depending on the hardware resources and MCU bandwidth that your application requires.

FIGURE 7: QUADRATURE DECODE FLOWCHART
FIGURE 8: TWO SERVOMOTOR SOLUTION

- Encoder 1
- Encoder 2
- PIC18C452
- H-Bridge Driver
- Motor 1
- Motor 2
- Decoding Circuit
- $+5\text{V}_{\text{DC}}$
- RB0/INT0
- I/O
- CCP1
- TOCKI
- T1CLKI
- CCP2

Connections:
- Encoder 1 to PIC18C452
- Encoder 2 to Decoding Circuit
- Decoding Circuit to PIC18C452
- PIC18C452 to H-Bridge Drivers
- H-Bridge Drivers to Motors
Software License Agreement

The software supplied herewith by Microchip Technology Incorporated (the "Company") for its PICmicro® Microcontroller is intended and supplied to you, the Company's customer, for use solely and exclusively on Microchip PICmicro Microcontroller products.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

APPENDIX A: PIC18C452 SERVOMOTOR SOURCE CODE

//---------------------------------------------------------------------
// File:18motor.c
//
// Written By: Stephen Bowling, Microchip Technology
//
// This code implements a brush-DC servomotor using the PIC18C452 MCU.
// The code was compiled using the MPLAB-C18 compiler ver. 1.00.
// The device frequency should be 20 MHz.
//
// The following files should be included in the MPLAB project:
//
// 18motor.c -- Main source code file
// p18c452.lkr -- Linker script file
//
// The following project files are included by the linker script:
//
// c018i.o -- C startup code
// clib.lib -- Math and function libraries
// p18c452.lib -- Processor library
//
//---------------------------------------------------------------------

#include <p18c452.h> // Register definitions
#include <stdlib.h>
#include <string.h>
#include <i2c.h> // I2C library functions
#include <pwm.h> // PWM library functions
#include <adc.h> // ADC library functions
#include <portb.h> // PORTB library function
#include <timers.h> // Timer library functions

// Constant Definitions

#define DIST 0 // Array index for segment distance
#define VEL 1 // Array index for segment vel. limit
#define ACCEL 2 // Array index for segment accel.
#define TIME 3 // Array index for segment delay time
#define INDEX PORTBbits.RB0 // Input for encoder index pulse
#define NLIM PORTBbits.RB1 // Input for negative limit switch
#define PLIM PORTBbits.RB2 // Input for positive limit switch
#define GPI PORTBbits.RB3 // General purpose input
#define MODE1 !PORTBbits.RB4 // DIP switch #1
#define MODE2 !PORTBbits.RB5 // DIP switch #2
#define MODE3 !PORTBbits.RB6 // DIP switch #3
#define MODE4 !PORTBbits.RB7 // DIP switch #4
#define SPULSE PORTCbits.RC5 // Software timing pulse output
#define ADRES ADRESH // Redefine for 10-bit A/D converter
const rom char ready[] = "\n\rREADY>";
const rom char error[] = "\n\rERROR!";

char inpbuf[8]; // Input command buffer

unsigned char eeadr, // Pointer to EEPROM address
firstseg, // First segment of motion profile
lastseg, // Last segment of motion profile
segnum, // Current executing segment
parameter, // Index to profile data
i, // index to ASCII buffer
comcount, // index to input string
udata // Received character from USART;

struct {// Holds status bits for servo
    unsigned phase:1; // Current phase of motion profile
    unsigned neg_move:1; // Backwards relative move
    unsigned motion:1; // Segment execution in progress
    unsigned saturated:1; // PWM output is saturated
    unsigned bit4:1;
    unsigned bit5:1;
    unsigned run:1; // Enables execution of profile
    unsigned loop:1; // Executes profile repeatedly
} stat;

int dtime, // Motion segment delay time
integral, // Integral value for PID alg.
kp,ki,kd, // PID gain constants
vlim, // Velocity limit for segment
mvelocity, // Measured motor velocity
DnCount, // Holds accumulated 'up' pulses
UpCount // Holds accumulated 'down' pulses
;

union LNG
{
    long l;
    unsigned long ul;
    int i[2];
    unsigned int ul[2];
    char b[4];
    unsigned char ub[4];
};

union LNG temp, // Temporary storage
accel, // Segment acceleration value
u0, // PID error value
ypid, // Holds output of PID calculation
velact, // Current commanded velocity
phaseldist // Half of segment distance
;

long position, // Commanded position.
mposition, // Actual measured position.
fposition, // Originally commanded position.
flatcount; // Holds the number of sample periods for which the velocity limit was reached in the first half of the move.
#pragma udata segdata1 = 0x0100

int segment1[12][4]; // Holds motion segment values in data memory.
int segment2[12][4]; //

#pragma udata

// Function Prototypes
void servo_isr(void); // Does servo calculations
void isrhandler(void); // Located at high interrupt vector
void DoCommand(void); // Processes command input strings
void Setup(void); // Configures peripherals and variables
void UpdPos(void); // Gets new measured position for motor
void CalcError(void); // Calculates position error
void CalcPID(void); // Calculates new PWM duty cycle
void UpdTraj(void); // Calculates new commanded position
void SetupMove(void); // Gets new parameters for motion profile

// Writes a string from ROM to the USART
void putrsUSART(const rom char *data);

// ExtEEWrite and ExtEERead are used to read or write an integer value to the
// 24C01 EEPROM
void ExtEEWrite(unsigned char address, int data);
int ExtEERead(unsigned char address);

#pragma interrupt servo_isr save = PRODL,PRODH,FSR0L,FSR0H

// Designate servo_isr as an interrupt function and save key registers
#pragma interrupt servo_isr

void isrhandler(void) // This function directs execution to the
{ // actual interrupt code
    _asm
        goto servo_isr
    _endasm
}

#pragma code

// Designate servo_isr as an interrupt function and save key registers
#pragma interrupt servo_isr save = PRODL,PRODH,FSR0L,FSR0H

// Locate ISR handler code at interrupt vector
#pragma code isrcode=0x0008

void servo_isr(void)
{ // Toggle output pin for ISR code timing
    SPULSE = 1;
}

#pragma code
UpdTraj(); // Get new commanded position
UpPos(); // Get new measured position
CalcError(); // Calculate new position error
CalcPID();

PIR1bits.TMR2IF = 0; // Clear Timer2 Interrupt Flag.
SPULSE = 0; // Toggle output pin for ISR code timing
}

// UpdTraj()
// Computes the next required value for the next commanded motor
// position based on the current motion profile variables. Trapezoidal
// motion profiles are produced.
//---------------------------------------------------------------------

void UpdTraj(void)
{

if(stat.motion & !stat.saturated)
{
    if(!stat.phase) // If in the first half of the move.
    {
        if(velact.ui[1] < vlim) // If still below the velocity limit
            velact.ul += accel.ul; // Accelerate
        else // If velocity limit has been reached,
            flatcount++; // increment flatcount.

        temp.ul = velact.ul; // Put velocity value into temp
        if(velact.ui[0] == 0x8000)
        {
            if(!(velact.ub[2] & 0x01))
                temp.ui[1]++;
            else;
        }
        else
            if(velact.ui[0] > 0x8000) temp.ui[1]++;
            else;

        phase1dist.ul -= (unsigned long)temp.ui[1];
        if(stat.neg_move)
            position -= (unsigned long)temp.ui[1];
        else
            position += (unsigned long)temp.ui[1];

        if(phase1dist.l <= 0) // If phase1dist is negative
            stat.phase = 1; // first half of the move has
            // completed.
    }
    else // If in the second half of the move,
    {
        if(flatcount) flatcount--;
        else
            if(velact.ul) // If commanded velocity not 0,
            {
                velact.ul -= accel.ul; // Decelerate
                if(velact.i[1] < 0)
velact.l = 0;

else // else
{
    if(dtime) dtime--; // Decrement delay time if not 0.
    else
        { // Move is done, clear motion flag
            position = fposition;
        }
}

temp.ul = velact.ul; // Put velocity value into temp
        // and round to 16 bits
if(velact.ui[0] == 0x8000)
    {
        if(!(velact.ub[2] & 0x01))
            temp.ui[1]++; // Update commanded position
        else
            { // END if (stat.motion)
                temp.ui[1]++;
            }
    }
else
    { // End if
        if(velact.ui[0] > 0x8000) temp.ui[1]++;
    }

    else
    { // If motion stopped and profile
        if(segnum < firstseg) segnum = firstseg;
        if(segnum > lastseg)
            { // Clear run flag if loop flag not set.
                if(!stat.loop) stat.run = 0;
            }
        else
            { // Get data for next motion segment.
                SetupMove();
                segnum++; // Increment segment number.
            }
    }
}

void SetupMove(void)
{
    if(segnum < 12) // Get profile segment data from
        { // data memory.
            phase1dist.i[0] = segment1[segnum][DIST];
            vlim = segment1[segnum][VEL];
            accel.i[0] = segment1[segnum][ACCEL];
dt ime = segment1[segnum][TIME];
}
else if(segnum < 24) {
    phasel dist.i[0] = segment2[segnum - 12][DIST];
    vlim = segment2[segnum - 12][VEL];
    accel.i[0] = segment2[segnum - 12][ACCEL];
    dt ime = segment2[segnum - 12][TIME];
}

phaseldist.b[2] = phasel dist.b[1]; // Rotate phasel dist one byte
phaseldist.b[1] = phasel dist.b[0]; // to the left.
phaseldist.b[0] = 0;
if(phaseldist.b[2] & 0x80) // Sign-extend value
    phasel dist.b[3] = 0xff;
else
    phasel dist.b[3] = 0;
accel.b[3] = 0; // Rotate accel one byte to
accel.b[2] = accel.b[1]; // the left
accel.b[1] = accel.b[0];
accel.b[0] = 0;

temp.l = position;
if(temp.ub[0] > 0x7f) // A fractional value is left
    temp.ub[0] = 0x100; // over in the 8 LSbits of
    temp.ub[0] = 0; // position, so round position
position = temp.l; // before computing final move
fposition = position + phasel dist.l; // Compute final position for
// the move
if(phaseldist.b[3] & 0x80) // If the move is negative,
    { stat.neg_move = 1; // Set flag to indicate negative
        phasel dist.l = -phaseldist.l; // move.
    }
else stat.neg_move = 0; // Clear flag for positive move
phaseldist.l >> = 1; // phasel dist holds total
// move distance, so divide by 2
velact.l = 0; // Clear commanded velocity
flatcount = 0; // Clear flatcount
stat.phase = 0; // Clear flag:first half of move
if(accel.l && vlim)
    stat.motion = 1; // Enable motion
}

//-------------------------------------------------------------------------------
// UpdPos()
// Gets the new measured position of the motor based on values
// accumulated in Timer0 and Timer1
//-------------------------------------------------------------------------------

void UpdPos(void)
{
    // Old timer values are presently stored in UpCount and DnCount, so
    // add them into result now.
    mvelocity = DnCount;
mvelocity -= UpCount;
    // Write new timer values into UpCount and DnCount variables.
UpCount = ReadTimer0();
DnCount = ReadTimer1();

// Add new count values into result.

mvelocity += UpCount;
mvelocity -= DnCount;

// Add measured velocity to measured position to get new motor
// measured position.

mposition += (long)mvelocity << 8;
}

//---------------------------------------------------------------------
// CalcError()
// Calculates position error and limits to 16 bit result
//---------------------------------------------------------------------

void CalcError(void)
{
  temp.l = position; // Put commanded pos. in temp
  temp.b[0] = 0; // Mask out fractional bits
  u0.l = mposition - temp.l; // Get error
  u0.b[0] = u0.b[1]; // from desired position and
  u0.b[1] = u0.b[2]; // shift to the right to discard
  u0.b[2] = u0.b[3]; // lower 8 bits.

  if (u0.b[2] & 0x80) // If error is negative.
  {
    u0.b[3] = 0xff; // Sign-extend to 32 bits.
    if((u0.ui[1] != 0xffff) || !(u0.ub[1] & 0x80))
    {
      u0.ui[1] = 0xffff; // Limit error to 16-bits.
      u0.ui[0] = 0x8000;
    }
    else;
  }

  else // If error is positive.
  {
    u0.b[3] = 0x00;
    if((u0.ui[1] != 0x0000) || (u0.ub[1] & 0x80))
    {
      u0.ui[1] = 0x0000; // Limit error to 16-bits.
      u0.ui[0] = 0x7fff;
    }
    else;
  }

  // Put 16-bit error result into temp

  //----------------------------------------------------------------------------
  // CalcPID()
  // Calculates PID compensator algorithm and determines new value for
  // PWM duty cycle
  //----------------------------------------------------------------------------

  void CalcPID(void)
  {
    ypid.i[0] = u0.i[0]*kp; // Calculate proportional term.
    ypid.ub[2] = AARGB1; // Get upper two bytes of 16 x 16
if(!stat.saturated) // If output is not saturated,
    integral += u0.i[0]; // add present error to integral
    // value.
if(ki) // If integral value is not 0,
{
    temp.i[0] = integral*ki;// calculate integral term.
    temp.ub[2] = AARGB1; // Get upper two bytes of 16 x 16
    ypid.l += temp.l; // Add multiply result to ypid.
}

if(kd) // If differential term is not 0,
{
    temp.i[0] = mvelocity*kd;// calculate differential term.
    temp.ub[2] = AARGB1; // Get upper two bytes of 16 x 16
    ypid.l += temp.l; // Add multiply result to ypid.
}

if(ypid.ub[3] & 0x80) // If PID result is negative
{
    {
        ypid.ui[1] = 0xff80; // Limit result to 24-bit value
        ypid.ui[0] = 0x0000;
    } else;
} else // If PID result is positive
{
    if((ypid.ub[3] || (ypid.ub[2] > 0x7f))
    {
        ypid.ui[1] = 0x007f; // Limit result to 24-bit value
        ypid.ui[0] = 0xffff;
    } else;
}

ypid.b[0] = ypid.b[1]; // Shift PID result right to

stat.saturated = 0; // Clear saturation flag and see
if(ypid.i[0] > 500) // if present duty cycle output
{
    ypid.i[0] = 500;
    stat.saturated = 1;
} else;
if(ypid.i[0] < -500)
{
    ypid.i[0] = -500;
    stat.saturated = 1;
}

ypid.i[0] += 512; // Add offset to get positive
    // duty cycle value.
SetDCPWM1(ypid.i[0]); // Write the new duty cycle.
}

//---------------------------------------------------------------------
// Setup() initializes program variables and peripheral registers
//---------------------------------------------------------------------

void Setup(void)
{ 
firstseg = 0; // Initialize motion segment
lastseg = 0; // variables
segnum = 0;
parameter = 0; // Motion segment parameter#
i = 0;
comcount = 0; // Input command index
udata = 0; // Holds USART received data
stat.phase = 0; // Set flags to 0.
stat.saturated = 0;
stat.motion = 0;
stat.run = 0;
stat.loop = 0;
stat.neg_move = 0;
dtime = 0;
integral = 0;
vlim = 0;
mvelocity = 0;
DnCount = 0;
UpCount = 0;
temp.1 = 0;
accel.1 = 0;
u0.1 = 0;
ypid.1 = 0;
velact.1 = 0;
phasedist.1 = 0;
position = 0;
mposition = 0;
fposition = 0;
flatcount = 0;
udata = 0;
mempset(inpbuf,0,8); // clear the input buffer

// Setup A/D converter
OpenADC(ADC_FOSC_32 & ADC_LEFT_JUST & ADC_1ANA_0REF,
ADC_CH0 & ADC_INT_OFF);

OpenPWM1(0xff); // Setup Timer2, CCP1 to provide
// 19.53 KHz PWM @ 20MHz

OpenTimer2(T2_PS_1_1 & T2_POST_1_10 & TIMER_INT_ON);
SetDCPWM1(512); // 50% initial duty cycle

EnablePullups(); // Enable PORTB pullups
PORTC = 0; // Clear PORTC
PORTD = 0; // Clear PORTD
PORTE = 0x00; // Clear PORTD
TRISC = 0xdb;
TRISD = 0; // PORTD all outputs.
TRISE = 0; // PORTE all outputs.

// Setup the USART for 19200 baud @ 20MHz
SPBRG = 15; // 19200 baud @ 20MHz
TXSTA = 0x20; // setup USART transmit
RCSTA = 0x90; // setup USART receive
putrsUSART("\r\nPIC18C452 DC Servomotor");
putrsUSART(ready);

OpenI2C(MASTER,SLEW_OFF); // Setup MSSP for master I2C
SSPADD = 49; // 100KHz @ 20MHz
kp = ExtEEERead(122); // Get PID gain constants
ki = 0; // from data EEPROM
kd = ExtEERead(126);

TMR0L = 0; // Clear timers.
TMR0H = 0;
TMR1L = 0;
TMR1H = 0;

OpenTimer0(TIMER_INT_OFF & T0_16BIT & TIMER_INT_OFF & T0_EDGE_RISE &
T0_SOURCE_EXT & T0_PS_1_1);
OpenTimer1(TIMER_INT_OFF & T1_SOURCE_EXT & T1_16BIT_RW & TIMER_INT_OFF
& T1_PS_1_1 & T1_SYNC_EXT_ON & T1_OSC1EN_OFF);

// Load motion profile data for segments 1 through 12 from
// data EEPROM

for (segnum=0; segnum < 12; segnum++)
{
    for (parameter=0; parameter < 4; parameter++)
    {
        eeadr = (segnum << 3) + (parameter << 1);
        segment1[segnum][parameter] = ExtEERead(eeadr);
    }
}

segment2[0][DIST] = 29500; // Motion profile data for segments
segment2[0][VEL] = 4096; // 13 through 24 are loaded into RAM
segment2[0][ACCEL] = 2048; // from program memory
segment2[0][TIME] = 1200;

segment2[1][DIST] = -29500;
segment2[1][VEL] = 1024;
segment2[1][ACCEL] = 512;
segment2[1][TIME] = 1200;

segment2[2][DIST] = 737;
segment2[2][VEL] = 4096;
segment2[2][ACCEL] = 2048;
segment2[2][TIME] = 1200;

segment2[3][DIST] = 737;
segment2[3][VEL] = 4096;
segment2[3][ACCEL] = 2048;
segment2[3][TIME] = 1200;

segment2[4][DIST] = 738;
segment2[4][VEL] = 4096;
segment2[4][ACCEL] = 2048;
segment2[4][TIME] = 1200;

segment2[5][DIST] = 738;
segment2[5][VEL] = 4096;
segment2[5][ACCEL] = 2048;
segment2[5][TIME] = 1200;

segment2[6][DIST] = -2950;
segment2[6][VEL] = 1024;
segment2[6][ACCEL] = 128;
segment2[6][TIME] = 1200;

segment2[7][DIST] = 2950;
segment2[7][VEL] = 256;
segment2[7][ACCEL] = 64;
segment2[7][TIME] = 1200;

segment2[8][DIST] = -2950;
segment2[8][VEL] = 4096;
segment2[8][ACCEL] = 512;
segment2[8][TIME] = 1200;

segment2[9][DIST] = 29500;
segment2[9][VEL] = 1024;
segment2[9][ACCEL] = 512;
segment2[9][TIME] = 1200;

segment2[10][DIST] = 29500;
segment2[10][VEL] = 2048;
segment2[10][ACCEL] = 512;
segment2[10][TIME] = 1200;

segment2[11][DIST] = 29500;
segment2[11][VEL] = 4096;
segment2[11][ACCEL] = 1024;
segment2[11][TIME] = 1200;

if(MODE1) // Check DIP switches at powerup
    stat.loop = 1; // If SW1 is on, set for loop mode
if(MODE2) // If SW2 is on, execute
    { // segments 12 and 13
        firstseg = 12;
        lastseg = 13;
        segnum = 12;
        stat.run = 1;
    }
else if(MODE3) // IF SW3 is on, execute
    { // segments 14 through 18
        firstseg = 14;
        lastseg = 18;
        segnum = 14;
        stat.run = 1;
    }
else if(MODE4) // If SW4 is on, execute
    { // segments 18 and 19
        firstseg = 18;
        lastseg = 19;
        segnum = 18;
        stat.run = 1;
    }
else;

INTCONbits.PEIE = 1; // Enable peripheral interrupts
INTCONbits.GIE = 1;  // Enable all interrupts
}

//@doc:// main()
//@doc//---------------------------------------------------------------------
//@doc// main()
//@doc//---------------------------------------------------------------------

void main(void)
{
    Setup(); // Setup peripherals and software
    // variables.

    while(1) // Loop forever
    {
        ClrWdt(); // Clear the WDT

        ConvertADC(); // Start an A/D conversion
        while(BusyADC()); // Wait for the conversion to complete
        PORTD = 0; // Clear the LED bargraph display.
        PORTE&= 0x04; //
    
if(ADRES > 225)
{
    PORTE |= 0x03; // Turn on 10 LEDS
    PORTD = 0xff;
}
if(ADRES > 200)
{
    PORTE |= 0x01; // Turn on 9 LEDS
    PORTD = 0xff;
}  
else if(ADRES > 175) PORTD = 0xff; // Turn on 8 LEDS
else if(ADRES > 150) PORTD = 0x7f; // 7 LEDS
else if(ADRES > 125) PORTD = 0x3f; // 6 LEDS
else if(ADRES > 100) PORTD = 0x1f; // 5 LEDS
else if(ADRES > 75) PORTD = 0x0f; // 4 LEDS
else if(ADRES > 50) PORTD = 0x07; // 3 LEDS
else if(ADRES > 25) PORTD = 0x03; // 2 LEDS
else if(ADRES > 0) PORTD = 0x01; // 1 LED
else;

if(PIR1bits.RCIF) // Check for USART interrupt
{
    switch(udata = RCREG)
    {
    case ',' : DoCommand(); // process the string
        memset(inpbuf,0,8); // clear the input buffer
        i = 0; // clear the buffer index
        comcount++; // increment comma count
        TXREG = udata; // echo the character
        break;

    case 0x0d: DoCommand(); // process the string
        memset(inpbuf,0,8); // clear the input buffer
        i = 0; // clear the buffer index
        comcount = 0; // clear comma count
        segnum = 0; // clear segment number
        parameter = 0; // clear parameter
        putrsUSART(ready); // put prompt to USART
        break;

    default: inpbuf[i] = udata; // get received char
        i++; // increment buffer index
        if(i > 7) // If more than 8 chars
        {
            putrsUSART(ready); // a <CR>, clear input
            memset(inpbuf,0,8); // buffer
            i = 0; // the buffer index
        }
        else TXREG = udata; // echo character
        break;

    }
}
}

void DoCommand(void)
{
if(comcount == 0) // If this is the first parameter of the input
    { // command...
        switch(inpbuf[0])
        {
            case 'X': parameter = DIST; // Segment distance change
                break;
            case 'V': parameter = VEL; // Segment velocity change
                break;
            case 'A': parameter = ACCEL; // Segment acceleration change
                break;
            case 'T': parameter = TIME; // Segment delay time change
                break;
            case 'P': parameter = 'P'; // Change proportional gain
                break;
            case 'I': parameter = 'I'; // Change integral gain
                break;
            case 'D': parameter = 'D'; // Change differential gain
                break;
            case 'L': parameter = 'L'; // Loop a range of segments
                break;
            case 'S': stat.run = 0; // Stop execution of segments
                break;
            case 'G': parameter = 'G'; // Execute a range of segments
                break;
            case 'W': if(PORTEbits.RE2) // Enable or disable motor
                    { // driver IC
                        putrsUSART("\r\nPWM On");
                        PORTEbits.RE2 = 0;
                    }
                    else
                    {
                        putrsUSART("\r\nPWM Off");
                        PORTEbits.RE2 = 1;
                    }
                    break;
            default: if(inpbuf[0] != '\0')
                    {
                        putrsUSART(error);
                    }
                    break;
        }
    }
else if(comcount == 1) // If this is the second parameter of the
    { // input command.
        if(parameter < 4) segnum = atob(inpbuf);
        else
            switch(parameter)
            {
                case 'P': kp = atoi(inpbuf); // proportional gain change
                    ExtEEWrite(122, kp); // Store value in EEPROM
                    break;
                case 'I': ki = atoi(inpbuf); // integral gain change
ExtEEWrite(124, ki); // Store value in EEPROM
break;

case 'D':
    kd = atoi(inpbuf); // differential gain change
    ExtEEWrite(126, kd); // Store value in EEPROM
    break;

case 'G':
    firstseg = atob(inpbuf);
    break;
    // Get the first segment in
    // the range to be executed.

case 'L':
    firstseg = atob(inpbuf);
    break;

default: break;
}

else if(comcount == 2)
{
    if(!stat.run) // If no profile is executing
    {
        if(parameter < 4) // If this was a segment parameter
        {
            // change.
            if(segnum < 12)
            {
                // Write the segment parameter into data memory
                segment1[segnum][parameter] = atoi(inpbuf);
                // Compute EEPROM address and write data to EEPROM
                eeadr = (segnum << 3) + (parameter << 1);
                ExtEEWrite(eeadr, segment1[segnum][parameter]);
            }
            else if(segnum < 24)
            // Write segment parameter data into data memory
            segment2[segnum - 12][parameter] = atoi(inpbuf);
            }
            else switch(parameter)
            {
            case 'G':
                lastseg = atob(inpbuf); // Get value for
                segnum = firstseg; // last segment.
                stat.loop = 0;
                stat.run = 1; // Start profile.
                break;

            case 'L':
                lastseg = atob(inpbuf); // Get value for
                segnum = firstseg; // last segment.
                stat.loop = 1; // Enable looping
                stat.run = 1; // Start profile
                break;

            default: break;
            }
        }
    }

else;
}

//---------------------------------------------------------------------
// ExtEEWrite()
// Writes an integer value to an EEPROM connected to the I2C bus at
// the specified location.
//---------------------------------------------------------------------
void ExtEEWrite(unsigned char address, int data) {
union {
  char b[2];
  int i;
} temp;
char error, retry;

temp.i = data;
error = 0;
retry = 10; // Poll the EEPROM up to 10 times
  do {
    error = EEAckPolling(0xA0);
    retry--;
    } while(error && retry > 0);
retry = 10; // Attempt to write low byte of data
  do // up to 10 times
    { 
    error = EEByteWrite(0xA0, address, temp.b[0]);
    retry--;
    } while(error && retry > 0);
retry = 10; // Poll the EEPROM up to 10 times
  do
    { 
    error = EEAckPolling(0xA0);
    retry--;
    } while(error && retry > 0);
retry = 10; // Attempt to write high byte of data
  do // up to 10 times
    { 
    error = EEByteWrite(0xA0, address + 1, temp.b[1]);
    retry--;
    } while(error && retry > 0);
}

// ExtEEWrite()
// Reads an integer value from an EEPROM connected to the I2C bus at
// the specified location.
//-------------------------------------

int ExtEERead(unsigned char address) {
union {
  char b[2];
  int i;
} data;
union {
  char b[2];
  int i;
} temp;
char retry;

retry = 10; // Attempt to read low byte of data do // up to 10 times
{
    temp.i = EERandomRead(0xA0, address);
    retry--;
} while(temp.b[1] && retry > 0);

if(temp.b[1]) data.b[0] = 0;// Make read result 0 if error else data.b[0] = temp.b[0]; // Otherwise get the low byte of data

retry = 10; // Attempt to read high byte of data do // up to 10 times
{
    temp.i = EERandomRead(0xA0, address + 1);
    retry--;
} while(temp.b[1] && retry > 0);

if(temp.b[1]) data.b[1] = 0; // Make read result 0 if error else data.b[1] = temp.b[0]; // Otherwise get the high byte of data

return data.i;
}

//--- putrsUSART() // Writes a string of characters in program memory to the USART
//---
void putrsUSART(const rom char *data)
{
    do
    {
        while(!((TXSTA & 0x02)));
        TXREG = *data;
        *data++;
    } while(*data++);
}
APPENDIX B: PIC16F877 SERVOMOTOR SOURCE CODE

//---------------------------------------------------------------------
//
// File:16mot877.c
//
// Written By:Stephen Bowling, Microchip Technology
//
// This code implements a brush-DC servomotor using the PIC18F877 MCU.
// The code was compiled using the HiTech PICC compiler ver. 7.85.
//
// The following files should be included in the MPLAB project:
//
// 16mot877.c-- Main source code file
//
//---------------------------------------------------------------------

#include <pic.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

//---------------------------------------------------------------------
//Constant Definitions
//---------------------------------------------------------------------
#define DIST 0 // Array index for segment distance
#define VEL 1 // Array index for segment vel. limit
#define ACCEL 2 // Array index for segment accel.
#define TIME 3 // Array index for segment delay time
#define INDEX RB0 // Input for encoder index pulse
#define NLIM RB1 // Input for negative limit switch
#define PLIM RB2 // Input for positive limit switch
#define GPI RB3 // General purpose input
#define MODE1 !RB4 // DIP switch #1
#define MODE2 !RB5 // DIP switch #2
#define MODE3 !RB6 // DIP switch #3
#define MODE4 !RB7 // DIP switch #4
#define SPULSE RC5 // Software timing pulse output
#define ADRES ADRESH // Redefine for 10-bit A/D converter
#define KP 250
#define KI 252
#define KD 254

//---------------------------------------------------------------------
// Variable declarations
//---------------------------------------------------------------------
const char ready[] = "\n\rREADY>";
const char error[] = "\n\rERROR!";

char inpbuf[8];
char tempch,UpCount;

unsigned char
eadr,
firstseg,
lastseg,
segnum,
parameter,
i,
comcount,
udata
;

```c
const char ready[] = "\n\rREADY>";
const char error[] = "\n\rERROR!";

char inpbuf[8];
char tempch,UpCount;

unsigned char
eadr,
firstseg,
lastseg,
segnum,
parameter,
i,
comcount,
udata
;
```
struct { // holds status bits for servo
    unsigned phase:1; // first half/second half of profile
    unsigned neg_move:1; // backwards relative move
    unsigned motion:1; // motion
    unsigned saturated:1; // servo output is saturated
    unsigned bit4:1;
    unsigned bit5:1;
    unsigned run:1;
    unsigned loop:1;
} stat;

union INTVAL
{
    int i;
    char b[2];
};

union LNG
{
    long l;
    unsigned long ul;
    int i[2];
    unsigned int ui[2];
    char b[4];
    unsigned char ub[4];
};

bank1 union LNG
calc,
temp,
u0,
ypid,
velact,
phasedist
;

bank1 int
dtime, // Segment delay time
integral,
kp,ki,kd, // PID gain constants
vlim, velcom,
mvelocity,
DnCount
;

bank1 long
position, // Commanded position.
mposition, // Actual measured position.
tposition,
flatcount; // Holds # of sample periods for which the
            // velocity limit was reached in first half of the move.

bank2 int segment1[12][4]; // array in bank2 for 12 motion segments
bank3 int segment2[12][4]; // array in bank3 for 12 motion segments

// Function Prototypes--------------------------------------------------

interrupt void servo_isr(void);
void putrsUSART(const char *data);
void DoCommand(void);
void Setup(void);
void UpdPos(void);
void UpdTraj(void);
void CalcError(void);
void CalcPID(void);
void SetupMove(void);
void EEDatWrite(unsigned char address, int data);
int EEDatRead(unsigned char address);

//---------------------------------------------------------------------
// servo_isr()
// Performs the servo calculations
//---------------------------------------------------------------------

interrupt void servo_isr(void)
{
    SPULSE = 1; // Toggle output pin for ISR code timing
    UpdTraj(); // Get new commanded position
    UpdPos(); // Get new measured position
    CalcError(); // Calculate new position error
    CalcPID();

    TMR2IF = 0; // Clear Timer2 Interrupt Flag.
    SPULSE = 0; // Toggle output pin for ISR code timing
}

//---------------------------------------------------------------------
// UpdTraj()
// Computes the next required value for the next commanded motor
// position based on the current motion profile variables. Trapezoidal
// motion profiles are produced.
//---------------------------------------------------------------------

void UpdTraj(void)
{
    if(stat.motion && !stat.saturated)
    {
        if(!stat.phase) // If in the first half of the move.
        {
            if(velact.ui[1] < vlim) // If still below the velocity limit
                velact.ul += accel.ul; // Accelerate
            else // If velocity limit has been reached,
                flatcount++; // increment flatcount.
            temp.ul = velact.ul; // Put velocity value into temp
                // and round to 16 bits
            if(velact.ui[0] == 0x8000)
                { // If velocity limit has been reached,
                    if(!(velact.ub[2] & 0x01))
                        temp.ui[1]++;
                    else;
                }
            else
                if(velact.ui[0] > 0x8000) temp.ui[1]++;
            else
                if(velact.ui[0] > 0x8000) temp.ui[1]++;
        }
        else
            if(velact.ui[0] < 0x8000) temp.ui[1]++;
            else
                phaseldist.ul -= (unsigned long)temp.ui[1];
        if(stat.neg_move)
            position -= (unsigned long)temp.ui[1];
        else
            position += (unsigned long)temp.ui[1];
        if(phaseldist.l <= 0) // If phaseldist is negative
            stat.phase = 1; // first half of the move has
            // completed.
else // If in the second half of the move,
{
  if(flatcount) flatcount--;

  else if(velact.ul) // If commanded velocity not 0,
  {
    velact.ul -= accel.ul; // Decelerate
    if(velact.i[1] < 0)
      velact.l = 0;
  }

  else // else
  {
    if(dtime) dtime--; // Decrement delay time if not 0.
    else
    {
      stat.motion = 0; // Move is done, clear motion flag
      position = fposition;
    }
  }

  temp.ul = velact.ul; // Put velocity value into temp
  // and round to 16 bits
  if(velact.ui[0] == 0x8000)
  {
    if(!(velact.ub[2] & 0x01))
      temp.ui[1]++;
    else;
  }
  else
  if(velact.ui[0] > 0x8000) temp.ui[1]++;
  else;

  if(stat.neg_move) // Update commanded position
    position -= (unsigned long)temp.ui[1];
  else
    position += (unsigned long)temp.ui[1];
}

  // END if (stat.motion)

else
{
  if(stat.run & !stat.motion) // If motion stopped and profile
    // running, get next segment number
    if(segnum < firstseg) segnum = firstseg;
  if(segnum > lastseg)
  {
    segnum = firstseg; // Clear run flag if loop flag not set.
    if(!stat.loop) stat.run = 0;
  }
  else
  {
    SetupMove(); // Get data for next motion segment.
    segnum++; // Increment segment number.
  }

  else;
}
}
void SetupMove(void)
{
if(segnum < 12) // Get profile segment data from
    // data memory.
    phase1dist.i[0] = segment1[segnum][DIST];
    vlim = segment1[segnum][VEL];
    accel.i[0] = segment1[segnum][ACCEL];
    dtime = segment1[segnum][TIME];
else if(segnum < 24)
    {
    phase1dist.i[0] = segment2[segnum - 12][DIST];
    vlim = segment2[segnum - 12][VEL];
    accel.i[0] = segment2[segnum - 12][ACCEL];
    dtime = segment2[segnum - 12][TIME];
    }
    phase1dist.b[2] = phase1dist.b[1]; // Rotate phase1dist one byte
    phase1dist.b[1] = phase1dist.b[0]; // to the left.
    phase1dist.b[0] = 0;
if(phase1dist.b[2] & 0x80) // Sign-extend value
    phase1dist.b[3] = 0xff;
else
    phase1dist.b[3] = 0;
accel.b[3] = 0; // Rotate accel one byte to
accel.b[2] = accel.b[1]; // the left
accel.b[1] = accel.b[0];
accel.b[0] = 0;
temp.l = position;
if(temp.ub[0] > 0x7f) // A fractional value is left
    temp.ub[0] += 0x100; // over in the 8 LSbits of
    temp.ub[0] = 0; // position, so round position
    position = temp.l; // variable to an integer value
    // before computing final move
fp = position + phase1dist.l; // Compute final position for
    // the move
if(phase1dist.b[3] & 0x80) // If the move is negative,
    {
    stat.neg_move = 1; // Set flag to indicate negative
    phase1dist.l = -phase1dist.l; // move.
    }
else stat.neg_move = 0; // Clear flag for positive move
    phase1dist.l >>= 1; // phase1dist holds total
    // move distance, so divide by 2
    velact.l = 0; // Clear commanded velocity
    flatcount = 0; // Clear flatcount
    stat.phase = 0; // Clear flag: first half of move
    if(accel.l && vlim)
        stat.motion = 1; // Enable motion
    }
}
void UpdPos(void)
{
    mvelocity = DnCount; // Get old DnCount value
    temp.b[0] = TMR1L; // Read Timer1
    temp.b[1] = TMR1H;
    if(TMR1L < temp.b[0]) // If a rollover occurred, read
        { // Timer1 again.
            temp.b[0] = TMR1L;
            temp.b[1] = TMR1H;
        }
    DnCount = temp.i[0]; // Store timer value in DnCount
    mvelocity -= DnCount; // Subtract new value from
                          // measured velocity
    tempch = -UpCount; // Put old UpCount value in
                        // temporary variable.
    UpCount = TMRO; // Read Timer0
    tempch += UpCount;
    if(tempch > 0)
        mvelocity += (int)tempch;
    else
        mvelocity -= (int)(tempch);
    mposition += (long)(mvelocity << 8); // Update measured position
}

//---------------------------------------------------------------------
// CalcError()
// Calculates position error and limits to 16 bit result
//---------------------------------------------------------------------
void CalcError(void)
{
    u0.l = mposition - position; // Get error
    u0.b[0] = u0.b[1]; // shift to the right to discard
    u0.b[1] = u0.b[2]; // lower 8 bits.
    u0.b[2] = u0.b[3];
    if (u0.b[2] & 0x80) // If error is negative.
        { // Sign-extend to 32 bits.
            u0.b[3] = 0xff;
            if((u0.i[1] != 0xffff) || !(u0.b[1] & 0x80))
                { // Limit error to 16-bits.
                    u0.ui[1] = 0xffff;
                    u0.ui[0] = 0x8000;
                }
        }
    else // If error is positive.
    {
        u0.b[3] = 0x00;
        if((u0.i[1] != 0x0000) || (u0.b[1] & 0x80))
            { // Limit error to 16-bits.
                u0.ui[1] = 0x0000;
                u0.ui[0] = 0x7fff;
            }
    }
}
else;
);
}

//---------------------------------------------------------------------
// CalcPID()
// Calculates PID compensator algorithm and determines new value for
// PWM duty cycle
//---------------------------------------------------------------------

void CalcPID(void)
{
    ypid.l = (long)u0.i[0]*(long)kp; // If proportional gain is not 0,
        // calculate proportional term.
    if(!stat.saturated) // If output is not saturated,
        integral += u0.i[0]; // add present error to integral
        // value.
    if(ki) // If integral value is not 0,
        ypid.l += (long)integral*(long)ki; // calculate integral term.
    if(kd) // If differential term is not 0,
        ypid.l += (long)mvelocity*(long)kd; // calculate differential term.

    if((ypid.b[3] & 0x80) // If PID result is negative
    {
        if((ypid.b[3] < 0xff) || !(ypid.b[2] & 0x80))
        {
            ypid.ui[1] = 0xff80; // Limit result to 24-bit value
            ypid.ui[0] = 0x0000;
        }
    }
    else // If PID result is positive
    {
        if(ypid.b[3] || (ypid.b[2] > 0x7f))
        {
            ypid.ui[1] = 0x007f; // Limit result to 24-bit value
            ypid.ui[0] = 0xffff;
        }
    }
else;
    }

    ypid.b[0] = ypid.b[1]; // Shift PID result right to
    stat.saturated = 0; // Clear saturation flag and see
    if(ypid.i[0] > 500) // if present duty cycle output
        { // exceeds limits.
            ypid.i[0] = 500;
            stat.saturated = 1;
        }
    if(ypid.i[0] < -500)
    {
        ypid.i[0] = -500;
        stat.saturated = 1;
    }

    ypid.i[0] += 512; // Add offset to get positive
    ypid.i[0] <<= 6; // duty cycle and shift left to
        // get 8 Msb’s in upper byte.
            CCPR1L = ypid.b[1]; // Write upper byte to CCP register
            // to set duty cycle.
        // Set 2 Lsb’s of duty cycle in
        // CCP1ICON register.
if(ypid.b[0] & 0x80) CCP1X = 1;
else CCP1X = 0;
if(ypid.b[0] & 0x40) CCP1Y = 1;
else CCP1Y = 0;

TMR2IF = 0;  // Clear Timer2 Interrupt Flag.
SPULSE = 0;
}

void Setup(void)
{
    firstseg = 0;  // Initialize motion segment
    lastseg = 0;   // variables
    segnum = 0;
    parameter = 0;  // Motion segment parameter#
    i = 0;         // Receive buffer index
    comcount = 0;  // Input command index
    udata = 0;     // Holds USART received data

    stat.phase = 0;      // Initialize flags and variables
    stat.saturated = 0;  // used for servo calculations.
    stat.motion = 0;
    stat.run = 0;
    stat.loop = 0;
    stat.neg_move = 0;
    dtime = 0;
    integral = 0;
    vlim = 0;
    velcom = 0;
    velcom = 0;
    mvelocity = 0;
    DnCount = 0;
    UpCount = 0;
    temp.1 = 0;
    accel.1 = 0;
    u0.1 = 0;
    ypid.1 = 0;
    velact.1 = 0;
    phaseldist.1 = 0;
    position = 0;
    mposition = position;
    fposition = position;
    flatcount = 0;

    memset(inpbuf,0,8);   // clear the input buffer
    udata = RCREG;
    udata = RCREG;
    RCREG = 0;
    udata = 0;

    SPBRG = 15;  // 19200 baud @ 20MHz
    TXSTA = 0x20;  // setup USART transmit
    RXSTA = 0x90;  // setup USART receive
    SSPADD = 49;  // Setup MSSP for master I2C
    SSPSTAT = 0;  // 100Khz @ 20MHz
    SSPCON2 = 0;
    SSPCON = 0x28;
    PR2 = 0xff;  // Setup Timer2 and CCP1 for
    T2CON = 0x44c;  // 19.53 Khz PWM @ 20MHz
    CCP1M1 = 0x7f;  // 50% initial duty cycle
CCP1CON = 0x0f; //
TMR2IF = 0; //
TMR2IE = 1; // Enable Timer2 interrupts
TMR1H = 0; //
TMR1L = 0; //
T1CON = 0x07; // Enable Timer1, ext. sync. clock
TMR0 = 0; //
OPTION = 0x6f; // Enable Timer0
ADCON1 = 0x04; // Setup A/D converter
ADCON0 = 0x81; //
PORTC = 0; // Clear PORTC
PORTD = 0; // Clear PORTD
PORTE = 0x00; // Clear PORTE
TRISC = 0xdb; //
TRISD = 0; // PORTD all outputs
TRISE = 0; // PORTE all outputs

kp = EEDatRead(KP);// Get PID gain values from
ki = 0; // data EEPROM
dk = EEDatRead(KD);//

.putsUSART("\n\nPIC16F877 DC Servomotor");
.putsUSART(ready);

PEIE = 1; // Enable interrupts.
GIE = 1; //
}

//---------------------------------------------------------------------
// main()
//---------------------------------------------------------------------

main()
{
  Setup();

  for(segnum=0;segnum < 12;segnum++) // Reads profile data into RAM for
    // from data EEPROM.
    for(parameter=0;parameter < 4;parameter++)
      {
        eeadr = (segnum << 3) + (parameter << 1);
        segment1[segnum][parameter] = EEDatRead(eeadr);
      }

  segment2[0][DIST] = 29500; // Initialize last 12 motion segments in
  segment2[0][VEL] = 4096; // RAM to pre-determined values for test
  segment2[0][ACCEL] = 2048; // purposes. This profile data is used
  segment2[0][TIME] = 1200; // when a profile is selected via the DIP
    // switches on the motor PCB.

  segment2[1][DIST] = -29500;
  segment2[1][VEL] = 1024;
  segment2[1][ACCEL] = 512;
  segment2[1][TIME] = 1200;

  segment2[2][DIST] = 737;
  segment2[2][VEL] = 4096;
  segment2[2][ACCEL] = 2048;
  segment2[2][TIME] = 1200;

  segment2[3][DIST] = 737;
  segment2[3][VEL] = 4096;
  segment2[3][ACCEL] = 2048;
  segment2[3][TIME] = 1200;
segment2[4][DIST] = 738;
segment2[4][VEL] = 4096;
segment2[4][ACCEL] = 2048;
segment2[4][TIME] = 1200;

segment2[5][DIST] = 738;
segment2[5][VEL] = 4096;
segment2[5][ACCEL] = 2048;
segment2[5][TIME] = 1200;

segment2[6][DIST] = -2950;
segment2[6][VEL] = 1024;
segment2[6][ACCEL] = 128;
segment2[6][TIME] = 1200;

segment2[7][DIST] = 2950;
segment2[7][VEL] = 256;
segment2[7][ACCEL] = 64;
segment2[7][TIME] = 1200;

if(MODE1) // If DIP switch #1 is on, set for looping mode.
    stat.loop = 1;

if(MODE2) // If DIP switches #2, #3, or #4 turned on,
{ // execute predetermined profile from above.
    firstseg = 12;
    lastseg = 13;
    segnum = 12;
    stat.run = 1;
}
else if(MODE3)
{ // if (MODE4)
    firstseg = 14;
    lastseg = 18;
    segnum = 14;
    stat.run = 1;
}

else if(MODE4)
{ // if (MODE4)
    firstseg = 18;
    lastseg = 19;
    segnum = 18;
    stat.run = 1;
}
else;

while(1)
{
    CLRWDT();

    ADGO = 1;            // Start an A/D conversion
    while(ADGO);        // Wait for the conversion to complete
    PORTD = 0;          // Clear the LED bargraph display.
    PORTE &= 0x04;      //

    if(ADRES > 225)
    {
        PORTE |= 0x03; // Turn on 10 LEDs
        PORTD = 0xff;
    }
    if(ADRES > 200)
    {
        PORTE |= 0x01; // Turn on 9 LEDs
        PORTD = 0xff;
    }
    else if(ADRES > 175) PORTD = 0x7f; // Turn on 8 LEDs
    else if(ADRES > 150) PORTD = 0x3f; // 7 LEDs
    else if(ADRES > 125) PORTD = 0x1f; // 6 LEDs
    else if(ADRES > 100) PORTD = 0x0f; // 5 LEDs
    else if(ADRES > 75) PORTD = 0x07; // 4 LEDs
    else if(ADRES > 50) PORTD = 0x03; // 3 LEDs
    else if(ADRES > 25) PORTD = 0x01; // 2 LEDs
    else if(ADRES > 0) PORTD = 0x01; // 1 LED
    else;

    if(RCIF)
    {
        switch(udata = RCREG)
        {
        case ',', ': DoCommand(); // process the string
            memset(inpbuf,0,8); // clear the input buffer
            i = 0; // clear the buffer index
            comcount++; // increment comma count
            TXREG = udata; // echo the character
            break;

        case 0x0d: DoCommand(); // process the string
            memset(inpbuf,0,8); // clear the input buffer
            i = 0; // clear the buffer index
            comnum = 0; // clear comma count
            segnum = 0; // clear segment number
            parameter = 0; // clear parameter
            putrsUSART(ready); // put prompt to USART
            break;

        default: inpbuf[i] = udata; // get received char
            i++; // increment buffer index
            if(i > 7) // If more than 8 chars
            {
                // received before getting
                putrsUSART(ready); // a <CR>, clear input
                memset(inpbuf,0,8); // buffer
                i = 0; // the buffer index
            }
            else TXREG = udata; // echo character
            break; //
        }
    }

    //end switch(udata)
    //end if(RCIF)
void DoCommand(void)
{
    if(comcount == 0) // If this is the first parameter of the input
    { // command...
        switch(inpbuf[0])
        {
            case 'X': parameter = DIST; // Segment distance change
                break;
            case 'V': parameter = VEL; // Segment velocity change
                break;
            case 'A': parameter = ACCEL; // Segment acceleration change
                break;
            case 'T': parameter = TIME; // Segment delay time change
                break;
            case 'P': parameter = 'P'; // Change proportional gain
                break;
            case 'I': parameter = 'I'; // Change integral gain
                break;
            case 'D': parameter = 'D'; // Change differential gain
                break;
            case 'L': parameter = 'L'; // Loop a range of segments
                break;
            case 'S': stat.run = 0; // Stop execution of segments
                break;
            case 'G': parameter = 'G'; // Execute a range of segments
                break;
            case 'W': if(RE2) // Enable or disable motor
            { // driver IC
                putrsUSART("\r\nPWM On");
                RE2 = 0;
            }
            else
            { // putrsUSART("\r\nPWM Off");
                RE2 = 1;
            }
            break;
            default: if(inpbuf[0] != '\0')
            {
                putrsUSART(error);
                break;
            }
        }
    }
    else if(comcount == 1) // If this is the second parameter of the
{ // input command.
if(parameter < 4) segnum = (char)(atoi(inpbuf));
else
switch(parameter)
{
    case 'P': kp = atoi(inpbuf); // proportional gain change
        EEDatWrite(KP, kp); // Store value in EEPROM
        break;
    case 'I': ki = atoi(inpbuf); // integral gain change
        EEDatWrite(KI, ki); // Store value in EEPROM
        break;
    case 'D': kd = atoi(inpbuf); // differential gain change
        EEDatWrite(KD, kd); // Store value in EEPROM
        break;
    case 'G': firstseg = (char)(atoi(inpbuf));
        break;
        // Get the first segment in
        // the range to be executed.
    case 'L': firstseg = (char)(atoi(inpbuf));
        break;

    default: break;
}
}
else if(comcount == 2)
{
if(!stat.run) // If no profile is executing
{
if(parameter < 4) // If this was a segment parameter
{
    // change.
    if(segnum < 12)
    {
        // Write the segment parameter into data memory
        segment1[segnum][parameter] = atoi(inpbuf);
        // Compute EEPROM address and write data to EEPROM
        eeadr = (segnum << 3) + (parameter << 1);
        EEDatWrite(eeadr, segment1[segnum][parameter]);
    }

    else if(segnum < 24)
    // Write segment parameter data into data memory
    segment2[segnum - 12][parameter] = atoi(inpbuf);
}
else switch(parameter)
{
    case 'G': lastseg = (char)(atoi(inpbuf)); // Get value for
        segnum = firstseg; // last segment.
        stat.loop = 0;
        stat.run = 1; // Start profile.
        break;
    case 'L': lastseg = (char)(atoi(inpbuf)); // Get value for
        segnum = firstseg; // last segment.
        stat.loop = 1; // Enable looping
        stat.run = 1; // Start profile
        break;

    default: break;
}
}
}
else;
}

//---------------------------------------------------------------------
// putrsUSART()
//
// Puts a string of characters in program memory to the USART
//---------------------------------------------------------------------

void putrsUSART(const char *data)
{
    do
    {
        while(!(TXSTA & 0x02));
        TXREG = *data;
    } while( *data++ );
}

//---------------------------------------------------------------------
// EEDatWrite()
//
// Writes an integer value to the 16F877 data EEPROM memory at
// the specified address.
//---------------------------------------------------------------------

void EEDatWrite(unsigned char address, int data)
{
    union INTVAL temp;
    temp.i = data;

    while(WR); // If write in progress, wait until done.
    EEDAT = address; // Load address to be written.
    EEDATA = temp.b[0]; // Load data to be written.
    EEPGD = 0; // Point to data memory.
    WREN = 1; // Enable writes.
    GIE = 0; // Disable interrupts.
    EECON2 = 0x55; // Required write sequence.
    EECON2 = 0xaa; // Required write sequence.
    WR = 1; // Do the write.

    while(WR); // If write in progress, wait until done.
    EEADR++; // Increment address.
    EEDATA = temp.b[1]; // Load data to be written.
    EECON2 = 0x55; // Required write sequence.
    EECON2 = 0xaa; // Required write sequence.
    WR = 1; // Do the write.
    while(WR); // Wait for write to finish.
    GIE = 1; // Reenable interrupts.
    WREN = 0; // Disable writes.
}

//---------------------------------------------------------------------
// EEDatRead()
//
// Reads an integer value as two bytes from the 16F877 data EEPROM at
// the specified address location
//---------------------------------------------------------------------

int EEDatRead(unsigned char address)
{
    union INTVAL data;

    EEAADR = address; // Load the address.
EEPGD = 0; // Point to data memory.
RD = 1; // Do the read.
data.b[0] = EEDATA; // Get the data.
EEADR++; // Point to next address.
RD = 1; // Do the read.
data.b[1] = EEDATA; // Get the data.
return data.i;
APPENDIX C: OPERATION WITH WINDOWS SOFTWARE

For convenience, a custom Windows® application is provided to control the servomotor. The control software for the servomotor is installed by copying the following files into a new directory on your PC:

- Servo.exe
- vcl35.bpl
- zcommobj.bpl
- cp3240mt.dll
- borlndmm.dll

Start the application by running Servo.exe. The following window should appear (see Figure C-1):

![DC Servomotor Control Panel](image)

The control software uses COM2 as the default COM port. If the servomotor is connected to a different COM port, you will need to change the setting to the appropriate port. If you point to Port in the menu bar, a drop-down menu appears (see Figure C-2). If the Close option is seen, you should close the current port and reopen the menu. Otherwise, point to the Select option, which provides a list of available COM ports. Choose the appropriate port and click Open.

### Programming a Single Motion Segment

A single motion segment may be programmed and executed using the Manual Motion section of the control window. For example, enter the following values into the appropriate data windows and click Start.

- **Distance**: 2950
- **Velocity**: 128
- **Acceleration**: 1024

You should observe that the motor shaft slowly rotates, then stops. Table C-1 summarizes the data value limits for all values in the control software.

### TABLE C-1: SERVOMOTOR PARAMETER LIMITS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Min. Value</th>
<th>Max. Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>-32000</td>
<td>32000</td>
<td>1 Shaft Revolution = 2950</td>
</tr>
<tr>
<td>Velocity</td>
<td>0</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>Acceleration</td>
<td>0</td>
<td>32000</td>
<td></td>
</tr>
<tr>
<td>Delay</td>
<td>0</td>
<td>32000</td>
<td>Number of servo update periods</td>
</tr>
<tr>
<td>Kp</td>
<td>0</td>
<td>6000</td>
<td></td>
</tr>
<tr>
<td>Kd</td>
<td>-32000</td>
<td>32000</td>
<td></td>
</tr>
</tbody>
</table>
Programming Up to Four Motion Segments

The User Profile Configuration area of the software control window allows four motion segments to be programmed into the motor and executed. A delay value is specified in servo update periods and determines the amount of idle time between consecutive motion segments.

You can enter the motion profile data for each motion segment into the appropriate data windows. The following are suggested starting values.

<table>
<thead>
<tr>
<th>Data Window</th>
<th>Seg 1</th>
<th>Seg 2</th>
<th>Seg 3</th>
<th>Seg 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance:</td>
<td>2950</td>
<td>2950</td>
<td>29500</td>
<td>29500</td>
</tr>
<tr>
<td>Velocity:</td>
<td>128</td>
<td>1024</td>
<td>128</td>
<td>512</td>
</tr>
<tr>
<td>Acceleration:</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
</tbody>
</table>

Click the Download button to transfer the parameters to the servomotor. Then, click the User radio button in the Motion Profiles box and either the Single or Loop button to start the profile.
Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable".
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip's products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999. The Company’s quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, KeelLoq® code hopping devices, Serial EEPROMs and microperipheral products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001 certified.