Motor Controller on a PIC 18f4520

Author: Matthew Turpin

INTRODUCTION

This design is intended as a cost effective, readily available motor controller solution capable of handling multiple controllers from one master device in a logical, powerful and easy to use interface.

This solution is best suited to drive DC brush motors at encoder rates up to and exceeding 2 MHz. Motor positions are measured with hardware quadrature encoding to achieve this high rate. A built in PID controller with adjustable gains is used to control each motor to be flexible enough for any choice of motor.

All data is stored on the device in int32 format and all calculations are integer operations, giving the controller great speed in carrying out the tracking operations. This does however have the limitation of discrete integer quantities for rotational speed.

If a desired motion is to be carried out by multiple slaves at the same instant, data can be programmed to all the devices of what the motion will be and then the device will carry out the action on the issue of a global “Go” pulse.

HARDWARE

The master device code is intended to be run on a PIC 18F4520 with a 20 MHz Clock and the slave code is intended to be run on a PIC 18F4520 with a 40 MHz clock. These values can be changed in the code to account for whatever is available, but will decrease the speed and accuracy of the response.

Motors must have sequential quadrature encoders used in combination with a hardware quadrature decoder. The decoder must have one up count pulse line and one down count pulse line. A few example chips that can be used for decoders are LS7083 and LS7183. In tests, LS7183 chips lost counts and had noisy signal and therefore the LS7083 would be recommended over the LS7183.

The motors will need to be powered from a separate voltage level from logic 5V. To drive the motors in both directions with minimal power loss, an H-Bridge will be used to convert the digital signal to a corresponding voltage. This can be driven from either a digital to analogue converter or from pulse width modulation (PWM). The later is chosen as it is built into the PIC 18F4520.

One PWM line is fed into an inverter and then the original PWM and the inverted signal are fed into the H-Bridge as inputs. This leads to a 50% duty cycle giving no voltage to the motor, a 0% duty cycle driving the motor full forwards, and a 100% duty cycle driving the motor full reverse. The PIC handles and is set up for 10 bit PWM resolution so a duty cycle of 0 is 0%, 511 is 50%, and 1023 is 100%.

For powerful motors such as the Pittman GM8224, an H-Bridge should be chosen with a large maximum current. For this example the L298N will be used. Smaller motors with maximum current draw of 1A or less will be able to be driven with smaller H-Bridges.

COMMUNICATION

The master can be used in one of two ways. The first is on its own, taking readings from each of the slave devices and computing the desired movement and relating that information to each of the slave devices.

The other method is to use MATLAB with RS232 serial communication to control the commands the master sends.

MASTER AND SLAVE

The motor controller set up consists of one master device and up to 16 preset slave devices. The master device communicates with the slaves over I2C (pronounced “I squared C”). Up to 96 additional addresses can be used with

slight modification to the slave code. Data is requested from the master and then sent back

interfere with the current command and current PID cycle, but will be resumed after communication is completed.

The I2C data transfer protocol is to send 5 bytes. The first byte is the identifier and the remaining 4 bytes are pieced together to form one 32 bit integer. Data transfer from the slave to the master is carried out in a similar manner, with the exception that the data is requested byte by byte as is required by I2C. This leads to slightly slower read speeds than write speeds.

Average timing for I2C communication is 500 microseconds for read operations of 4 bytes and only 200 microseconds for write operations of 4 bytes.

PIC C COMMANDS

There are 2 main types of commands on the master device: send data and request data. Setting variables and commands to the slave are in the format:

set_mc_command(device, desired result);

The "set" signifies the changing of a variable on the slave device and the "mc" stands for "motor controller". An example command is:

set_mc_position(0x80,-39000);
mc_go();

This commands the slave with address 0x80 to move to absolute position negative 39000 counts. Absolute position zero is defined when the device is turned on and can be reset to zero at any time using the command:

mc_resetpos(device);
mc_go();

The master device must send the mc_go(); command before the action will be carried out. This is a non-device-specific command as all devices will carry out their last set command and does not need to immediately follow each

using I2C interrupts on the slave. This may

command. The following set commands are available:

set_mc_"____"(device,number)

position	Absolute position
velocity	Rotational velocity (in encoder counts per 250 ms)
kp	Proportional constant
ki	Integral gain constant
kd	Derivative gain constant

The read commands are used in a similar fashion. The difference in the syntax is the address is the only input. The function call returns the value being asked for as an int32 or signed int32.
The following get commands are available:

mc_get_"_____"(device)

position	Absolute position
velocity	Rotational velocity (in encoder counts per 250 ms)
TargetPosition	Absolute Position being tracked
TargetVelocity	Rotational Velocity being tracked

Additional commands are:

mc_go();	Begin actions on all slaves
mc_pos(device);	Set slave to position tracking
mc_vel(device);	Set slave to velocity tracking
mc_resetpos(device);	Sets current position to zero

MATLAB COMMANDS

MATLAB commands are nearly identical to the PIC commands. The difference is that the connection the master PIC is made through the function MotorControllerConnect.m. Choose the appropriate port and it will return the serial object it creates. That serial object is an input into the commands to control the master device.

An example communication is shown below. Note that this is the communication to get to plots in figures 1 and 2.

```
s = MotorControllerConnect(7)
mc_resetpos(8,s);
pause(2);
kp = 10000;
ki = 0;
set_mc_kp(kp,8,s)
set_mc_ki(ki,8,s)
time = 0:15;
set_mc_position(39000,8,s);
tic;
for ii = 1:(size(time,2)-1);
    pos(ii) = mc_get_position(8,s);
    time(ii+1) = toc+time(ii);
    pause(.1);
end
plot(time(1,2:16),pos,)
```

This communication can handle any number between –(2^31) and 2^31 for velocities and positions and from 0 to 2^31 for gains.

Below is a complete listing of MATLAB functions. There is no need to issue a mc_go(), mc_pos() or mc_vel() command as they are built into the MATLAB functions.

serial object = MotorControllerConnect(serial port number)
position = mc_get_position(device,serial object)
velocity = mc_get_velocity(device,serial object)
mc_resetpos(device, serial object)
set_mc_position(position,device,serial object)
set_mc_velocity(velocity,device,serial object)
set_mc_kp(kp,device,serial object)
set_mc_ki(ki,device,serial object)
set_mc_kd(kd,device,serial object)

Table 1: MATLAB Commands

CONTROL

Each slave uses its own PID control algorithm with programmable control gains to drive the motor to the desired position and velocity. The algorithm completes one cycle in under 200 microseconds with a 40 MHz clock. This allows the device to call the PID function once every 250 microseconds which totals to the device processing the PID control 4000 times per second.

The device uses anti-windup when the integral term is nonzero. The method of anti-saturation is whenever the motor is at full power, the integral term does not accumulate. This results in no windup, but gives a slight discontinuity when the slave gets below saturation and the integral term begins to accrue.

Some step response plots are shown below. A mass of 3 kg was attached to a Pittman GM8224 motor and set to spin one revolution or 39000 counts. The first plot is with Kp = 10000, Ki = 0, and Kd = 0. The second plot adds an integral tern of 10. There is a slight delay in the response as well as less overshoot and nearly zero steady state error (oscillates around zero error). Additionally, notice when the integral takes control, the response speeds up with a bit of discontinuous motion.

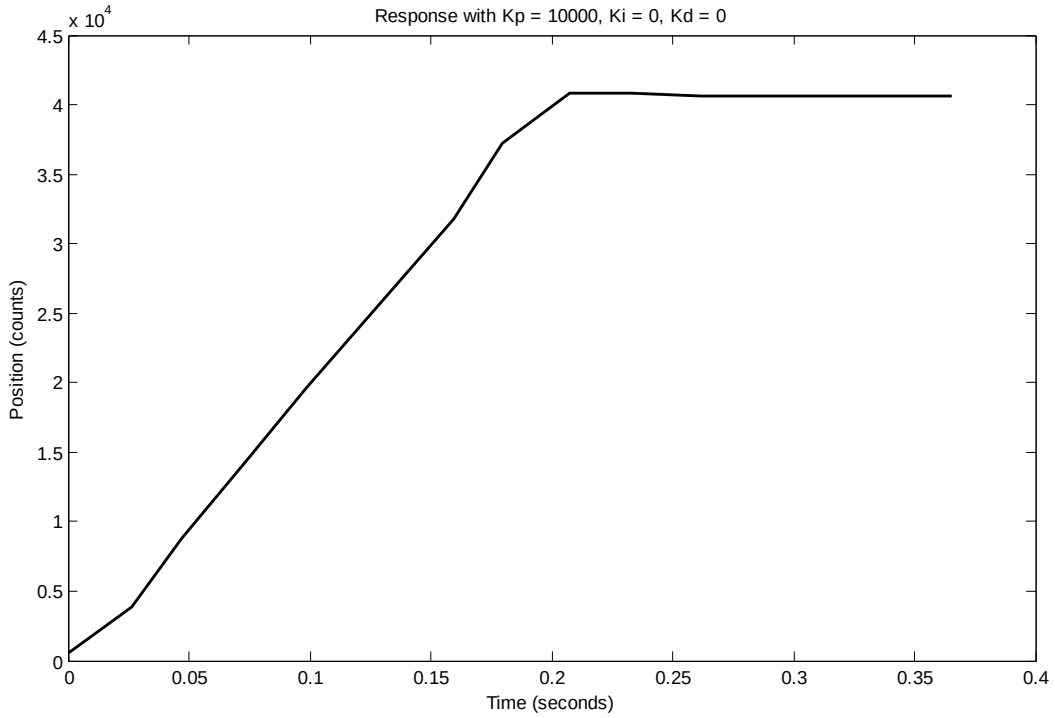

Figure 1: Plot of a step response of 39000 with Kp=10000, Ki = 10, Kd = 0: Final value = 40750

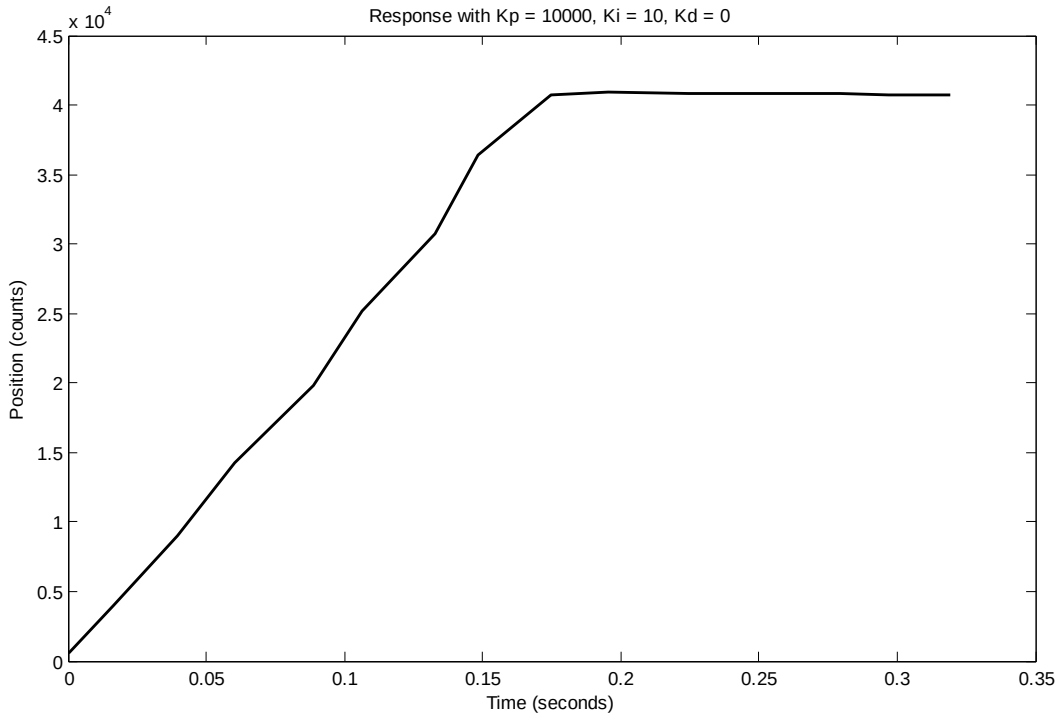

Figure 2: Plot of a step response of 39000 with Kp=10000, Ki = 10, Kd = 0: Final value = 39005

ADDRESSING

There are 16 preset addresses as outlined in the table below. Use resistors to pull up the appropriate pins on the slave. These addresses are only read once on startup, so if a change is made to the addressing, a restart of the slave is required.

Device Number	Decimal I2C Address	Hex I2C Address	Binary I2C Address	Input A Pin E1	Input B Pin E2	Input C Pin B5	Input D Pin B4
0	128	80	10000000	0	0	0	0
1	130	82	10000010	0	0	0	1
2	132	84	10000100	0	0	1	0
3	134	86	10000110	0	0	1	1
4	136	88	10001000	0	1	0	0
5	138	8A	10001010	0	1	0	1
6	140	8C	10001100	0	1	1	0
7	142	8E	10001110	0	1	1	1
8	144	90	10010000	1	0	0	0
9	146	92	10010010	1	0	0	1
10	148	94	10010100	1	0	1	0
11	150	96	10010110	1	0	1	1
12	152	98	10011000	1	1	0	0
13	154	9A	10011010	1	1	0	1
14	156	9C	10011100	1	1	1	0
15	158	9E	10011110	1	1	1	1

Table 2: Addressing Slave Device Pin Inputs

WIRING

The wiring of the PICs is straightforward. The single master controller requires only power, ground, I2C data and clock lines, a “Go” line and optional serial lines for communication with MATLAB. The SDA and SCL lines need pull up resistors with a value of about 10K.

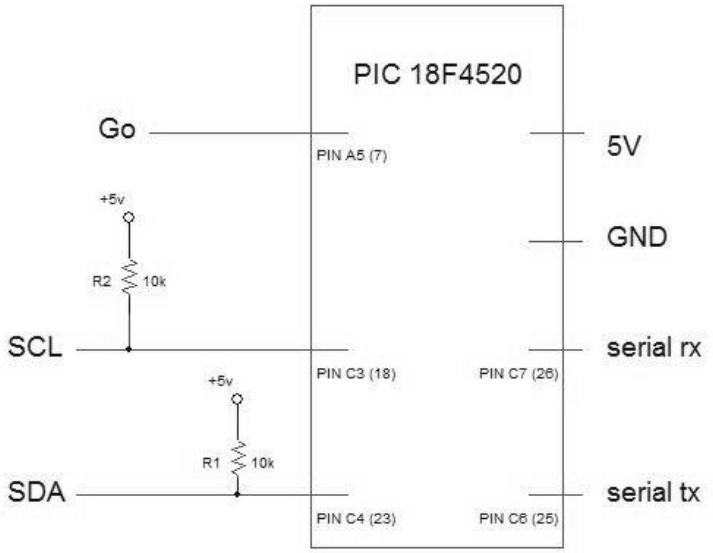

Figure 3: Master Wiring Diagram

The slave configuration is as shown in Figure 4. The LS7083 quadrature counter takes inputs A and B from the motor Encoder and outputs pulses according Figure 5. The mode can be changed from X4 mode to X1 mode by changing the mode (pin 6) to ground. This will result in one quarter the number of counts output by the device. The Rbias value of 100K results in a pulse width of approximately 500 nanoseconds. This can be increased if the controller is missing counts.

Inputs A-D on the PIC are determined based on what address the slave will take. See Table 1 for addressing information.

SCL and SDA are the same for all slaves and only one set of pull up resistors is needed for the entire motor controller. The Go line is also common between all devices and needs no resistor.

The H-Bridge takes both the inverted and non-inverted signals and outputs n appropriate voltage difference across the motor terminals. The fly back diodes are shown, but may be unnecessary for some H-Bridges.

Supply Voltage can be any voltage within the range of the motor and H-Bridge and should not be the same supply as that for logic to prevent noise from affecting the PICs.

If the motor appears to not be tracking, try reversing Encoder wires.

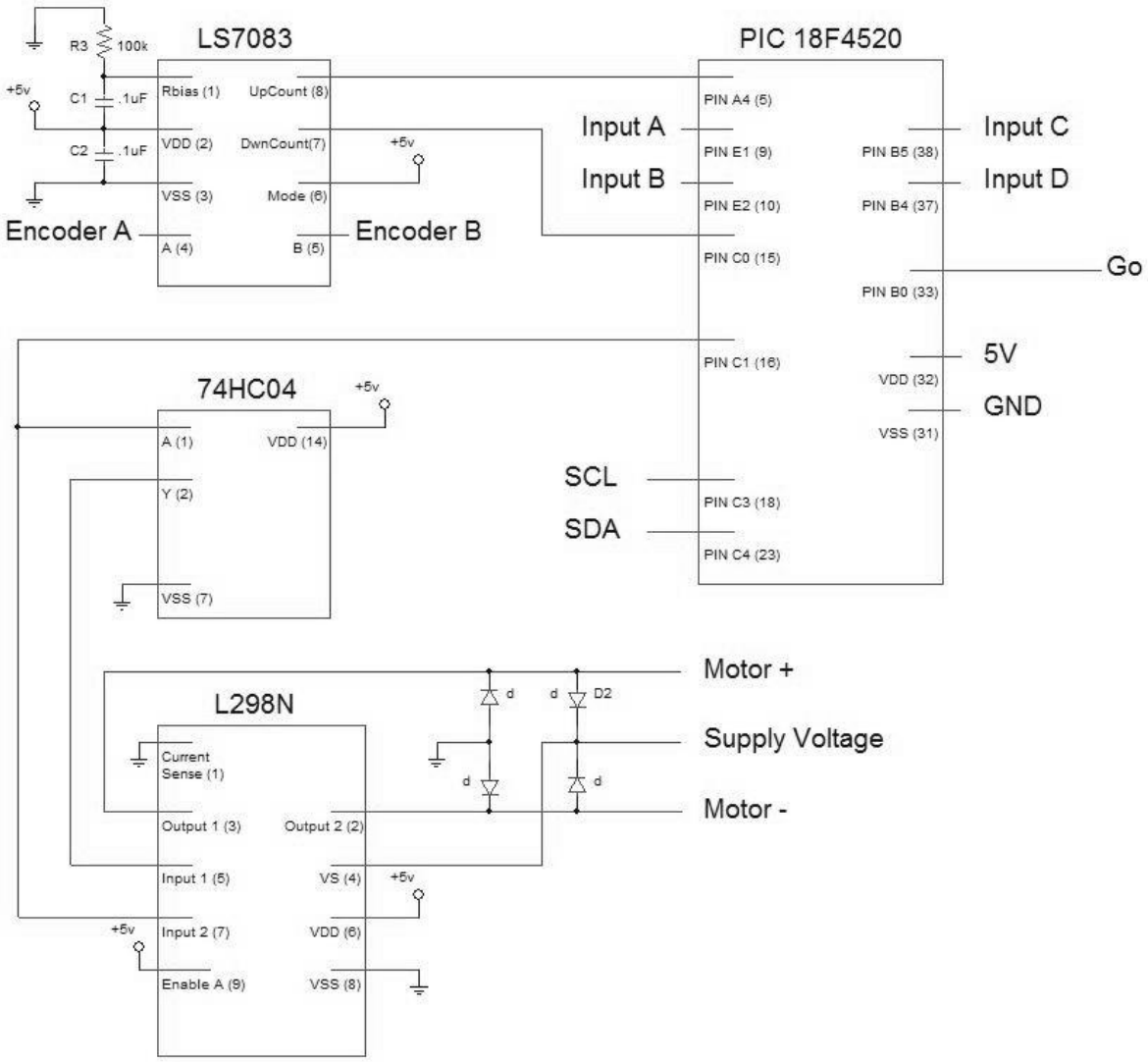

Figure 4: Slave Wiring Diagram

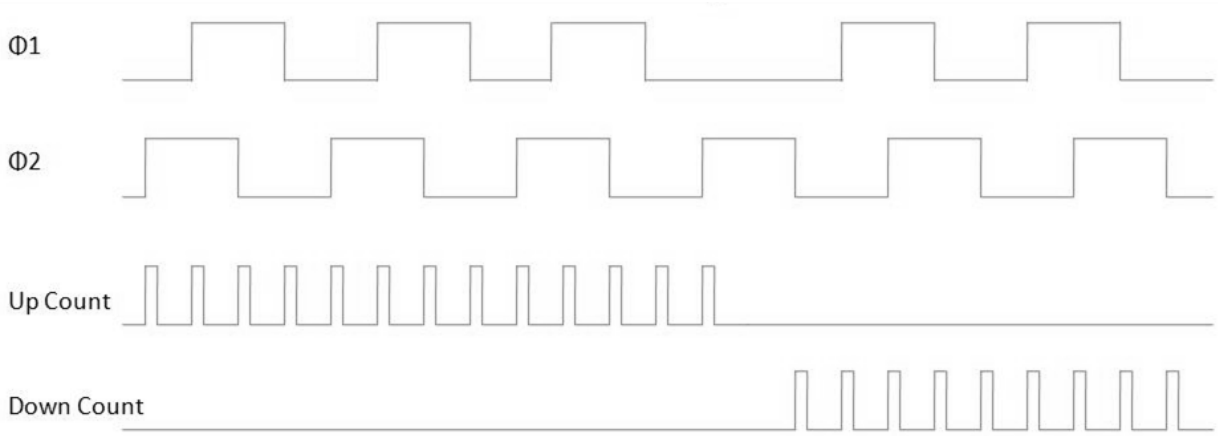

Figure 5: Quadrature Diagram in 4X mode

Slave Code

Put this code on each slave controller:

```
#include <18f4520.h>
#device high_ints=TRUE
#fuses HS,NOLVP,NOWDT,NOPROTECT
#use delay(clock=40000000)                              //Numbers are preset for 40 mHz clock
#use i2c(SLAVE, FAST, SCL=PIN_C3, SDA=PIN_C4, address=0x80, FORCE_HW)
 //Using hardware I2C, defaults to address 0x80 or device 1

//Encoder Variables
int32 TotalCount = 1<<31;
int16 Count0 = 0, Count1 = 0;
int16 Last0 = 0, Last1 = 0;

//PID vars
int32 error = 0;
signed int32 p_error = 0;
signed int32 i_error = 0;
signed int32 d_error = 0;
signed int32 error_old = 0;
signed int32 pid_error = 0;
int32 kp = 16000;
int32 ki = 0;
int32 kd = 0;
signed int16 Duty = 511;
int normalize = 16;
int i_check = 0;
int i_spacing = 10;

//Velocity Variables
signed int32 Velocity = 0;

//Tracking Variables
int AngleOrVel = 0;
signed int32 TargetVelocity = 0;
int32 TargetPosition = (1<<31);

//I2C variables
int State;                              //State of Transmission
int32 Number32Bit = 0;                  //Stores bumbers converted to 32 bit
int Number[4] = {0,0,0,0};              //Stores 4 bytes of 32 bit numbers
int DataType = 0;                       //Type of operation being preformed


void runsetup()
{
  I2C_SlaveAddr(128|((int)input(pin_B4)<<1)|((int)input(pin_B5)<<2)|((int)input(pin_E2)<<3)|((int)input(pin_E1)<<4));
//setting internal address see data sheet for truth table

  setup_timer_0(RTCC_EXT_L_TO_H | RTCC_DIV_1);      //external input from quadrature encoder 0
  setup_timer_1(T1_EXTERNAL | T1_DIV_BY_1);         //external input from quadrature encoder 1
  setup_timer_2(T2_DIV_BY_1, 255, 1);               //for PWM at 20KHz, out of audible range
  setup_timer_3(T3_INTERNAL|T3_DIV_BY_1);           //for counting time, increments every .2us and rolls over 76.3 times per sec
  enable_interrupts(INT_SSP);                       //enable I2C interrupt
  enable_interrupts(INT_EXT);                       //enable "go" interrupt on pin C7
```

```
  enable_interrupts(global);

  setup_ccp2(ccp_pwm);                    //setup PWM

  set_timer0(0);                          //clear starting point of timers
  set_timer1(0);

  set_PWM2_duty(511);                     //motor starts at 0 velocity

}
void PID()                                //Proportional-Integral-Derivative control
{
  switch(AngleOrVel)                      //0 for angle tracking, 1 for velocity tracking
  {
    case 0:
      if(TargetPosition==TotalCount)      //Exactly on target
      {
        error=0;
        Duty = 511;                       //Stop motor
      }
      else if(TargetPosition>TotalCount)
      {
        error = (TargetPosition-TotalCount);     //Current error
        p_error = ((error*kp)>>normalize);       //Calculating proportional error
        if (p_error>512)
        {
          Duty = 1023;                    //Motor saturated
        }
        else
        {
          d_error = (error-error_old)*kd;       //Calculating derivative error
          if (i_check>=i_spacing)         //Spaces out integral calculations to avoid premature integral buildup
          {
            i_error += error*ki;          //Calculating integral error
            i_check = 0;
          }
          else
          {
            i_check++;
          }
          pid_error = p_error+(i_error>>normalize)+d_error;  //PID combined error

          if (pid_error>511)              //PID error will saturate motor
          {
            if (i_check == 0)
            {
              i_error -= error*ki;        //Anti-windup action taken
            }
            Duty = 1023;                  //Motor is saturated
          }
          else if(pid_error>(-512))       //PID between -512 and 511 does not saturate motor
          {
            Duty = 511+pid_error;         //Gives non-saturated duty
          }
          else                            //PID error less than -512
          {
            Duty  = 0;                    //Motor saturated
          }
```

```
      error_old = error;                //Remember preveous error for derivative calculation
    }
  }
  else                                  //Same as above but for negative error
  {
    error = (TotalCount-TargetPosition);    //Current error
    p_error = ((error*kp)>>normalize);      //Calculating proportional error
    if (p_error>512)
    {
      Duty = 0;                         //Motor saturated
    }
    else
    {
      d_error = (error-error_old)*kd;       //Calculating derivative error

      if (i_check>=i_spacing)               //Spaces out integral calculations to avoid premature integral buildup
      {
        i_error -= error*ki;                //Calculating integral error
        i_check = 0;
      }
      else
      {
        i_check++;
      }
      pid_error = p_error+(i_error>>normalize)+d_error;  //PID combined error

      if (pid_error>511)
      {
        if (i_check == 0)                   //PID error will saturate motor
        {
          i_error += error*ki;              //Anti-windup action taken
        }
        Duty = 0;                       //Motor saturated
      }
      else if (pid_error>(-512))            //PID between -512 and 511 does not saturate motor
      {
        Duty = 512-pid_error;               //Gives non-saturated duty
      }
      else
      {
        Duty = 1023;                    //PID error less than -512
      }
      error_old = error;                //Remember preveous error for derivative calculation
    }
  }
break;
case 1:                                 //Velocity tracking
  Duty+=TargetVelocity-Velocity;            //Adjust Duty to trak the velocity
    if (Duty>1023)                          //Saturated Motor
    {
      Duty = 1023;
      Duty-=TargetVelocity-Velocity;
    }
    else if (Duty<0)                        //Saturated Motor
    {
      Duty = 0;
      Duty-=TargetVelocity-Velocity;
    }
```

```
      break;
  break;
  }
  set_PWM2_duty(Duty);                    //Adjust PWM in response to PID control
}

void cleartrackingvars()                  //clears all tracking variables on each new command
{
  Duty = 511;                             //Motors stop
  error = 0;                              //Error variables set to 0
  p_error = 0;
  i_error = 0;
  d_error = 0;
  error_old = 0;
  pid_error = 0;
}


void updatestatus()                       //Calculates position and velocity based on encoder counts
{
  Count0 = get_timer0();                  //Up counts
  Count1 = get_timer1();                  //Down counts

  Velocity = ((signed int32)(Count0 - Last0))-((signed int32)(Count1 - Last1));   //Current Velocity

  disable_interrupts(global);             //Disable interrupts temporarily to prevent inacurate position calcultaion
  TotalCount += ((int16)(Count0 - Last0));        //Increment up counts
  Last0 = Count0;

  TotalCount -= ((int16)(Count1 - Last1));        //Increment down counts
  Last1 = Count1;
  enable_interrupts(global);              //Re-enable interrupts

  PID();                                  //Track using PID
}
void updatetracking()                     //Updates variables on external signal on line B0
{
  switch(DataType)
  {
    case 1:                               //Position or Velocity select
      AngleOrVel = (int8)Number32Bit;
    break;
    case 2:                               //Target position update
      TargetPosition = (1<<31)+(signed int32)Number32Bit;
    break;
    case 3:                               //Target velocity update
      TargetVelocity = (signed int32)Number32Bit;
    break;
    case 4:                               //Proportionality constant update
      kp = (int16) Number32Bit;
    break;
    case 5:                               //Integral constant update
      ki = (int16) Number32Bit;
    break;
    case 6:                               //Derative constant update
      kd = (int16) Number32Bit;
    break;
    case 10:                              //Reset position variable
```

```
      TotalCount = 1<<31;
      TargetVelocity = 0;
      TargetPosition = 1<<31;
    break;
  }
  cleartrackingvars();                    //Clear tracking variables after new data is received
}

void recombine()                          //Turns 4 one byte numbers into one 32 bit number
{
  Number32Bit = ((int32)Number[0]|(int32)Number[1]<<8|(int32)Number[2]<<16|(int32)Number[3]<<24);
  delay_us(1);                            //Prevents overflow
}

#INT_EXT high
void motorgo()                            //Begins latest tracking information
{
  updatetracking();
}

#INT_SSP high
void ssp_interupt ()                      //I2C interrupt
{
  disable_interrupts(global);                 //Disable interrupts necessary to prevent exit during transmission
  state = i2c_isr_state();
  if(state < 0x80)                        //Master is sending data
  {
    if(state == 0)                        //First received byte is address
    {
    }
    if(state == 1)                        //Second received byte is address
    {
      DataType = i2c_read();
    }
    if(state >1)                          //Additional received bytes are data
    {
      Number[state-2] = i2c_read();
    }
    if (state==5)                         //Combine 4 one byte numbers into one 32 bit number
    {
    recombine();
    }
  }
  if(state >= 0x80)                       //Master is requesting data
  {
    switch(DataType)                      //Slices data to send to master
    {
      case 255:
        i2c_write((int8)(TotalCount));
      break;
      case 254:
        i2c_write((int8)(TotalCount>>8));
      break;
      case 253:
        i2c_write((int8)(TotalCount>>16));
      break;
      case 252:
        i2c_write((int8)(TotalCount>>24));
```

```
      break;
      case 251:
        i2c_write((int8)(Velocity));
      break;
      case 250:
        i2c_write((int8)(Velocity>>8));
      break;
      case 249:
        i2c_write((int8)(Velocity>>16));
      break;
      case 248:
        i2c_write((int8)(Velocity>>24));
      break;
      case 247:
        i2c_write((int8)(TargetPosition));
      break;
      case 246:
        i2c_write((int8)(TargetPosition>>8));
      break;
      case 245:
        i2c_write((int8)(TargetPosition>>16));
      break;
      case 244:
        i2c_write((int8)(TargetPosition>>24));
      break;
      case 243:
        i2c_write((int8)(TargetVelocity));
      break;
      case 242:
        i2c_write((int8)(TargetVelocity>>8));
      break;
      case 241:
        i2c_write((int8)(TargetVelocity>>16));
      break;
      case 240:
        i2c_write((int8)(TargetVelocity>>24));
      break;
    }
  }
  enable_interrupts(global);                    //Re-enable interrupts
}

void main()
{
  runsetup();
  while (TRUE)
  {
    while(get_timer3()>2475)                    //Enters main function every 250us or 4000 times per second
    {
      set_timer3(0);                          //Reset timer 3
      updatestatus();
    }
  }
}
```

Master Code

Put this code on the master controller as well as put the following reference file in the same directory to include it:

```
#include <18f4520.h>
#fuses HS,NOLVP,NOWDT,NOPROTECT
#use delay(clock=20000000)
#use I2C(FAST, SCL=PIN_C3, SDA=PIN_C4, FORCE_HW)        //Set up Hardware I2C
#use rs232(baud=19200, UART1)                    //Set up PIC UART on RC6 (tx) and RC7 (rx)
#include "MotorControllerFunctions.c"                //Motor Controller function calls


// Set up data_tx (transmit values), data_rx (recieve values)
int8 data_tx, data_rx[11] = {0,0,0,0,0,0,0,0,0,0,0};

//Counting Varibles
signed int ii = 0;
int jj = 0;

//Device Name
int Device = 0;

//Temporary variables
int32 Send = 0;
int32 Input=0;


void write()                                //For writing to MATLAB
{
  for(jj=0;jj<4;jj++)
  {
    data_tx = ((int8)(Send>>(8*jj)));              //Breaks variables to send to MATLAB into readable Bytes
    printf("%u\n", data_tx);                    //Sends one Byte at a time
  }
}


void readincoming()                            //Any incoming data is dealt with in this function
{
  data_rx[ii] = fgetc();                       //Read in recieved value from buffer
  if((data_rx[ii]>96)&&(ii>1))                    //Ascii characters above 96 (lowercase letters) are used to recieve values from the
Slave
  {
    switch(data_rx[ii])
    {
      case 112:                              //"p" retrieves position
        Send = mc_get_pos(Device);
        write();
      break;
      case 114:                              //"r" to reset position
        mc_resetpos(Device);
        mc_go();
      break;
      case 118:                              //"v" retrieves velocity
        Send = (mc_get_vel(Device)+(1<<31));
        write();
```

```
    break;
  }
  ii=-1;                              //Negative 1 counters ii++ later in function
}
else if ((data_rx[ii]>70)&&(ii>1))          //Capital Ascii characters are used to set vales to the Slave
{
  data_rx[2] = data_rx[ii];             //Ensures no read errors on MATLAB communication
  ii = 2;
}
else if(data_rx[ii]<65)                 //Normalizing Ascii values 1-9 to decimal values 1-9
{
  data_rx[ii] = data_rx[ii]-48;
}
else                                    //Normalizing Ascii values A-E to decimal values 10-15
{
  data_rx[ii] = data_rx[ii]-55;
}
ii++;                                   //Increment value being referenced
if(ii==2)                               //After recieving the address of the slave, convert from hex to binary
{
  Device = ((data_rx[0]<<4)|data_rx[1]);
}
if(ii>10)                               //Enter after all hex values recieved
{
  Input = 0;                            //Zero out Input variable
  for(jj=0;jj<8;jj++)                   //Convert from eight hex values to one 32 bit binary value
  {
    Input = ((Input<<4)|(data_rx[3+jj]));
  }
  switch(data_rx[2])                    //Call appropriate functions to carry out actions
  {
    case 73:                            //"I" for integral constant
      set_mc_ki(Device,(int32)(Input));
      mc_go();
    break;
    case 75:                            //"K" for proportional constant
      set_mc_kp(Device,(int32)(Input));
      mc_go();
    break;
    case 80:                            //"P" for position
      mc_pos(Device);
      mc_go();
      set_mc_abspos(Device,(signed int32)(Input-(1<<31)));
      mc_go();
    break;
    case 82:                            //"R" for derivative constant
      set_mc_kd(Device,(int32)(Input));
      mc_go();
    break;
    case 86:                            //"V" for velocity
      mc_vel(Device);
      mc_go();
      set_mc_vel(Device,(signed int32)(Input-(1<<31)));
      mc_go();
    break;
  }
  ii = 0;                               //Reset counting variable
}
```

```
}


void main()
{

  while (TRUE)                          //PIC will always watch for data
  {
    if (kbhit())                        //If PIC senses data pushed to serial buffer
    {
      readincoming();                   //Call function to handle incoming data
    }
  }
}
```

Include this C file saved as MotorControllerFunctions.c

```
void mc_transmit(int device, int* output[5])
{
  int ii;

  i2c_start();              //begin transmission
  i2c_write(device);        //select address of device to communicate with
  for(ii=0;ii<5;  ii++)
  {
    i2c_write(output[ii]);
  }
  i2c_stop();
  delay_us(10);
}

void set_mc_abspos(int device,  int32 position)
{
  int* output[5] = {0,0,0,0,0};
  int ii;

  output[0] = 2;
  for(ii=0;ii<4;ii++)
  {
    output[ii+1] = (int8)(position>>(8*(ii)));
  }
  mc_transmit(device,output);
}

void set_mc_vel(int device,  signed int32 velocity)
{
  int* output[5] = {0,0,0,0,0};
  int ii;

  output[0] = 3;
  for(ii=0;ii<4;ii++)
  {
    output[ii+1] = (int8)(velocity>>(8*(ii)));
  }
  mc_transmit(device,output);
```

```
}

void mc_pos(int device)
{
  int* output[5] = {1,0,0,0,0};
  mc_transmit(device,output);
}

void mc_vel(int device)
{
  int* output[5] = {1,1,0,0,0};
  mc_transmit(device,output);
}

void mc_go()
{
  output_high(pin_a5);
  delay_us(10);
  output_low(pin_a5);
}

void mc_resetpos(int device)
{
  int* output[5] = {10,0,0,0,0};
  mc_transmit(device,output);
}

void set_mc_kp(int device,  int32 kp)
{
  int* output[5] = {4,0,0,0,0};
  int ii;

  for(ii=0;ii<4;ii++)
  {
    output[ii+1] = (int8)(kp>>(8*(ii)));
  }
  mc_transmit(device,output);
}

void set_mc_ki(int device,  int32 ki)
{
  int* output[5] = {5,0,0,0,0};
  int ii;

  for(ii=0;ii<4;ii++)
  {
    output[ii+1] = (int8)(ki>>(8*(ii)));
  }
  mc_transmit(device,output);
}

void set_mc_kd(int device,  int32 kd)
{
  int* output[5] = {6,0,0,0,0};
  int ii;

  for(ii=0;ii<4;ii++)
  {
```

```
    output[ii+1] = (int8)(kd>>(8*(ii)));
  }
  mc_transmit(device,output);
}

signed int32 mc_request(int device, int datatype)
{
  int ii;
  int input[4] = {0,0,0,0};

  for(ii=0;ii<4;ii++)            //change this back
  {
    i2c_start ();                  //begin communication
    i2c_write (device);              //send slave address
    i2c_write (datatype-ii);
    i2c_start ();                  //send repeated start command to begin read cycle
    i2c_write (device+1);            //add 1 to the address to send a write bit
    input[ii] = i2c_read(0);         //read requested information from the slave
    i2c_stop ();                   //terminate communication
  }
  return ((int32)input[0]|((int32)input[1]<<8)|((int32)input[2]<<16)|((int32)input[3]<<24));
}

signed int32 mc_get_pos(int device)
{
  int32 position;
  position = mc_request(device,255);
  return ((signed int32)position);
}

signed int32 mc_get_vel(int device)
{
  int32 velocity;
  velocity = mc_request(device,251);
  return ((signed int32)velocity);
}

signed int32 mc_get_targetpos(int device)
{
  int32 targetposition;
  targetposition = mc_request(device,247);
  return ((signed int32)targetposition);
}

signed int32 mc_get_targetvel(int device)
{
  int32 targetvelocity;
  targetvelocity = mc_request(device,243);
  return ((signed int32)targetvelocity);
}
```

MATLAB Functions:

```
function [s] =  MotorControllerConnect(com)
%MotorController takes a COM port input and returns a serial object to be
%   used with other motorcontroller functions.
%
%Input the COM port as a double or enter it at the prompt

%Record of Revisions:
%       Date                 Programmer         Description of Change
%       ====                 ==========         =====================
%     3/14/2007            Matthew Turpin      Original Code

%clears old serial objects
delete(instrfind)

%if COM port not specified, request it
if nargin~=1
    comport = ['com' num2str(input('Enter COM port: '))];
elseif nargin==1
     comport = ['com' num2str(com)];
end

%open the serial port and return the serial object
s = serial(comport,'BAUD',19200);           % Create serial object (PORT Dependent)
fopen(s)                                    % Open the serial port for r/w

end
```

```
function [position] = mc_get_position(device, s)
%mc_get_position takes a device, and serial object and returns the absolute
%   position of that device
%
%   This function outputs numbers in the range of -(2^31) to 2^31


%Record of Revisions:
%       Date                Programmer          Description of Change
%       ====                ==========          =====================
%     3/14/2007            Matthew Turpin      Original Code


    %lookup of device hex address
    dev = devlookup(device);

    %convert to a string
    str = [dev 'p'];
    data = zeros(1,4);

    %transmit string
    for ii = 1:3
        fprintf(s, '%s', str(1,ii));
    end

    %recieve string
    for ii=1:4
        data(1,ii) = str2num(fscanf(s));
    end

    %recombine data
    position = (data(4)*2^24+data(3)*2^16+data(2)*2^8+data(1))-2^31;
 end
```

```
function [velocity] = mc_get_velocity(device,s)
%mc_get_velocity takes a device, and serial object and returns the
%   rotational velocity of that device
%
%   This function outputs numbers in the range of -(2^31) to 2^31


%Record of Revisions:
%       Date                Programmer          Description of Change
%       ====                ==========          =====================
%     3/14/2007            Matthew Turpin      Original Code


    %lookup of device hex address
    dev = devlookup(device);

    %convert to a string
    str = [dev 'v'];
    data = zeros(1,4);

    %transmit string
    for ii = 1:3
        fprintf(s, '%s', str(1,ii));
    end

    %recieve string
    for ii=1:4
        data(1,ii) = str2num(fscanf(s));
    end

    %recombine data
    velocity = (data(4)*2^24+data(3)*2^16+data(2)*2^8+data(1))-2^31;
 end
```

```
function mc_resetpos(device, s)
%mc_resetpos resets the current position of the device being addressed to zero

%Record of Revisions:
%       Date                Programmer          Description of Change
%       ====                ==========          =====================
%     3/14/2007            Matthew Turpin      Original Code


    %lookup of device hex address
    dev = devlookup(device);

    %convert to a string
    str = [dev 'r'];

    %transmit string
    for ii = 1:3
        fprintf(s, '%s', str(1,ii));
    end
 end
```

```
function set_mc_position(position, device, s)
%set_mc_position takes a position input, device, and serial object
%   this information is exported to the master of the motor controller and
%   the move to that absolute position is carried out.
%
%   This function supports inputs from -(2^31) to 2^31


%Record of Revisions:
%       Date                Programmer          Description of Change
%       ====                ==========          =====================
%     3/14/2007            Matthew Turpin      Original Code

    %lookup of device hex address
    dev = devlookup(device);

    %convert to a string
    str = [dev 'P' dec2hex(position+2^31)];

    %transmit string
    for ii = 1:11
        fprintf(s, '%s', str(1,ii));
    end
end
```

```
function set_mc_velocity(velocity, device, s)
%set_mc_velocity takes a velocity input, device, and serial object
%   This information is exported to the master of the motor controller and
%   the controller adjusts the velocity to the desired level
%
%   This function supports inputs from -(2^31) to 2^31


%Record of Revisions:
%       Date                Programmer          Description of Change
%       ====                ==========          =====================
%     3/14/2007            Matthew Turpin      Original Code

    %lookup of device hex address
    dev = devlookup(device);

    %convert to a string
    str = [dev 'V' dec2hex(velocity+2^31)];

    %transmit string
    for ii = 1:11
        fprintf(s, '%s', str(1,ii));
    end
end
```

```
function set_mc_kp(kp, device, s)
%set_mc_kp takes a gain constant input, device, and serial object
%   this information is exported to the master of the motor controller and
%   the adjusts the velocity to the desired level
%
%   This function supports inputs from 0 to 2^31

%Record of Revisions:
%       Date                Programmer         Description of Change
%       ====                ==========         =====================
%     3/14/2007            Matthew Turpin      Original Code

    dev = devlookup(device);

    %convert to a string
    str = [dev 'K' dec2hex(kp+2^31)];
    str(1,4) = '0';

    %transmit string
    for ii = 1:11
        fprintf(s, '%s', str(1,ii));
    end
end
```

```
function set_mc_ki(ki, device, s)
%set_mc_ki takes a gain constant input, device, and serial object
%   this information is exported to the master of the motor controller and
%   the adjusts the velocity to the desired level
%
%   This function supports inputs from 0 to 2^31


%Record of Revisions:
%       Date                Programmer          Description of Change
%       ====                ==========          =====================
%     3/14/2007            Matthew Turpin      Original Code


    %lookup of device hex address
    dev = devlookup(device);

    %convert to a string
    str = [dev 'I' dec2hex(ki+2^31)];
    str(1,4) = '0';

    %transmit string
    for ii = 1:11
        fprintf(s, '%s', str(1,ii));
    end
end
```

```
function set_mc_kd(kd, device, s)
%set_mc_kd takes a gain constant input, device, and serial object
%   this information is exported to the master of the motor controller and
%   the adjusts the velocity to the desired level
%
%   This function supports inputs from 0 to 2^31


%Record of Revisions:
%       Date                Programmer          Description of Change
%       ====                ==========          =====================
%     3/14/2007            Matthew Turpin      Original Code


    %lookup of device hex address
    dev = devlookup(device);

    %convert to a string
    str = [dev 'I' dec2hex(kd+2^31)];
    str(1,4) = '0';

    %transmit string
    for ii = 1:11
        fprintf(s, '%s', str(1,ii));
    end
end
```