DC motor-encoder position servo controller

1. System hardware

Position servo systems come in all shapes and sizes. Here we describe a very simple positional servo system based on a DC motor and rotary position encoder. This type of device is useful for controlling positions of a linear device after appropriate rotary-to-linear motion conversion through a rack and pinion or leadscrew arrangement. The principle application described here is associated with an objective drive system, where the arrangement is used to retract an objective to allow a microscope stage to be initialised. Once the stage is initialised, a process which would have fouled the objective, the latter is returned to its normal focus position.

The mechanical accuracy that can be achieved depends very much on these ancillary mechanical components, but two factors restrict the ultimate performance which can be achieved: the first is the resolution of the encoder and the second is the backlash inevitable in almost all forms of low-cost gearboxes. We overcome the former by ensuring a high gearing ration between the output and the encoder (which is fitted directly to the motor) and the latter by ensuring that we approach the target position form the same direction.

We have based our position servo system around a PIC microcontroller, a motor driver and an encoder pulse counter coupled to an off-the-shelf 16 mm diameter DC motor-encoder-gearbox combination available from Maxon Motors. Our design is used to position a focusing platform in an in-house developed optical microscope. We only required the ability to position this platform relatively coarsely as fine optical focus was performed with a piezo-electric objective focus system. However, a high degree of repeatability was required. This controller was used to move an objective out of the way of an XY stage while the latter was being initialised.

The drive system used is available from Maxon Motor (Maxon House, Hogwood Lane, Finchampstead, Berkshire, RG40 4QW, http://www.maxonmotor.co.uk) as part #333-118 which is a composite item consisting of a 16 mm diameter motor (1.2 W, 9V dc nominal voltage) (Maxon Motor part #110 055), a 370:1 gearbox (Maxon Motor part #110324) and a 32 counts/turn quadrature encoder (Maxon Motor part #201 935). Of course other similar motors and gearboxes can be used. The encoder is available with an index channel if required, but we do not make use of this in the current design.

Since we use a gearbox, each turn of the motor is encoded to a resolution of (32 x 370):1, i.e. 11840:1. In our microscope drive system, we further reduce the motor rotation by ~1.69:1 (i.e. overall resolution is ~20000:1) by using belt-coupled toothed gears (13 and 22 teeth on motor and microscope drive respectively, as shown in Figure 2. This rotation turns a micrometer screw with a pitch of 500 µm/turn; the 20000 encoder pulses thus correspond to a travel distance of 500 µm. The Avago HCTL-2022 quadrature counter has a 4x count mode (i.e. a one count is associated with every transition of the two phases of the quadrature encoder outputs) so the number of counts for 500µm is 80000 counts.

One count thus corresponds to ~6.25 nm of linear travel and this level of linear resolution is more than enough for a coarse focusing system. If we assume that the motor would run at ~133 turns/second (~8000 rpm), the micrometer rotation speed would be 133/(370 x 1.69) rps or ~0.21 rps, or 4.73 seconds per turn, ensuring a linear speed of ~9.5 seconds/mm of focus travel. In practice, the motor can be run at a somewhat faster speed (>10000 rpm) and the resulting linear speed is correspondingly improved. On the other hand, the maximum encoder output frequency is of the order of 8 kHz, and the above calculations assume a typical output frequency of ~4.2 kHz (133 rps x 32 pulses/rev), so the motor speed should not be increased too much. In our system the time taken to move 3 mm of focus travel is ~23 seconds.

In order to determine the motor-gearbox's output shaft position, the encoder pulses are counted in a dedicated quadrature counter, an Avago HCTL-2022 (Farnell 116-1110). This is a 20-PIN chip which includes a quadrature decoder followed by a 32 bit bidirectional counter and a 4 x 8 bit output bus. A Peripheral Interface Controller, more commonly known as a PIC processor monitors the HCTL-2022's state and calculates the appropriate motor speed and direction to drive a simple bipolar transistor H-bridge motor driver, ST Microlectronics' L293E (Farnell 146-7711). The full circuit of the servo system is shown in Figure 1. We make us of a pulse-width modulated output from the PIC to determine motor speed and approach the target position slowly so as not to 'loose' counts. The PIC can also shut down the motor and of course determine direction of rotation. The pulse-width modulated output polarity reverses when the motor direction is reversed; this is a consequence of the logic inputs of the motor driver. The target position is sent to the PIC from a host computer through an I^2C port (see application note AN10216, www.nxp.com for details of this two-wire communications protocol). However the motor can also be inched or driven continuously with a centre-off biased 1 pole 2 way manual toggle switch. Although motor stall is highly unlikely

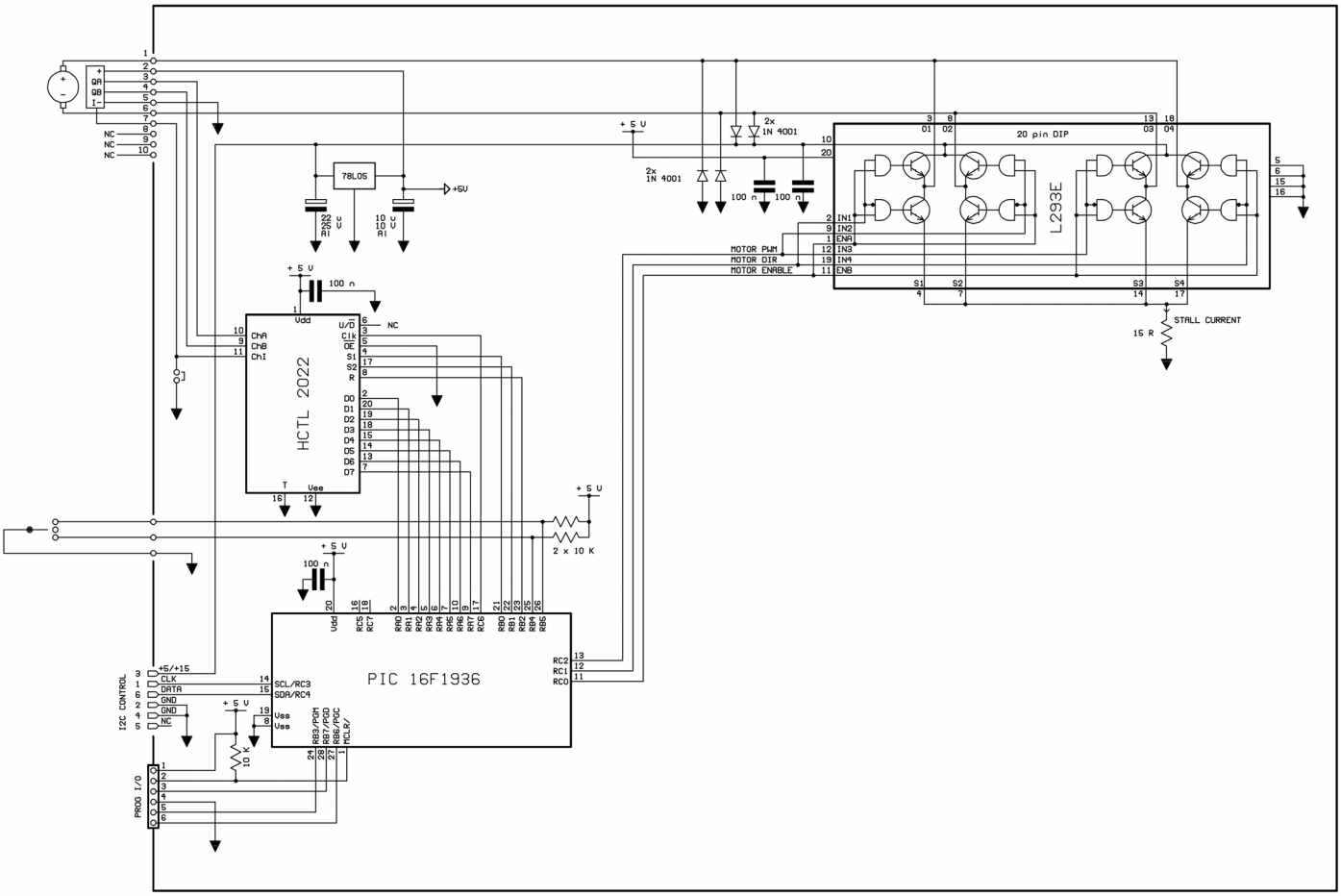


Figure 1: Complete circuit of the position servo controller. The 16F1936 PIC can be programmed through a 6 pin port (IDC style) while the motor and encoder are connected through 10 pin IDC connector. Lines RB0-2 send control data to the HCTL 2022 counter, while the counter state, i.e. position, is monitored by port RA0-7. Line RC6 provides a sampling clock (4 MHz) to the HCTL2022 quadrature decoder, while lines RC0-RC2 are used to drive the L293E H-bridge. Lines RB4 and RB5 are used to sense the position of the manual control switch.

The motor driver chip contains two identical H-bridge circuits and these are paralleled to increase available current drive before current limit, since the motor may require currents close to 1A during start-up and following an abrupt stop. Although not strictly required, four 1N4001 back-emf protection diodes are placed on the H-bridge output. We use the same printed circuit board to drive other, larger, motors where diodes are indeed required.

When the motor is running at full speed, the PIC needs to read the counter at least every 125 μs so as to minimise the possibility of target position overshoot. On the other hand, the PIC also needs to monitor the state of the external (I^2C) and local (up-down switch) communications. Furthermore there is inevitable inertia in the motor-gearbox and hence a two-speed approach algorithm is used: when the encoder-derived position is close to the target position, the motor PWM is reduced, decreasing the motor's speed and allowing a smooth stop at the target position.

2. Practical implementation

Figure 2 shows one application where the unit is used to provide coarse focus is a microscope. The left image in Figure 2 also shows a fine focus assembly using a piezo-electric focus unit; upper panels show SolidWorks models of the assembly. The servo controller electronics are constructed in a small plastic enclosure as shown in Figure 3 and the motor/gearbox/encoder is connected directly to a circuit board which contains the PIC, the counter and the motor driver chips.

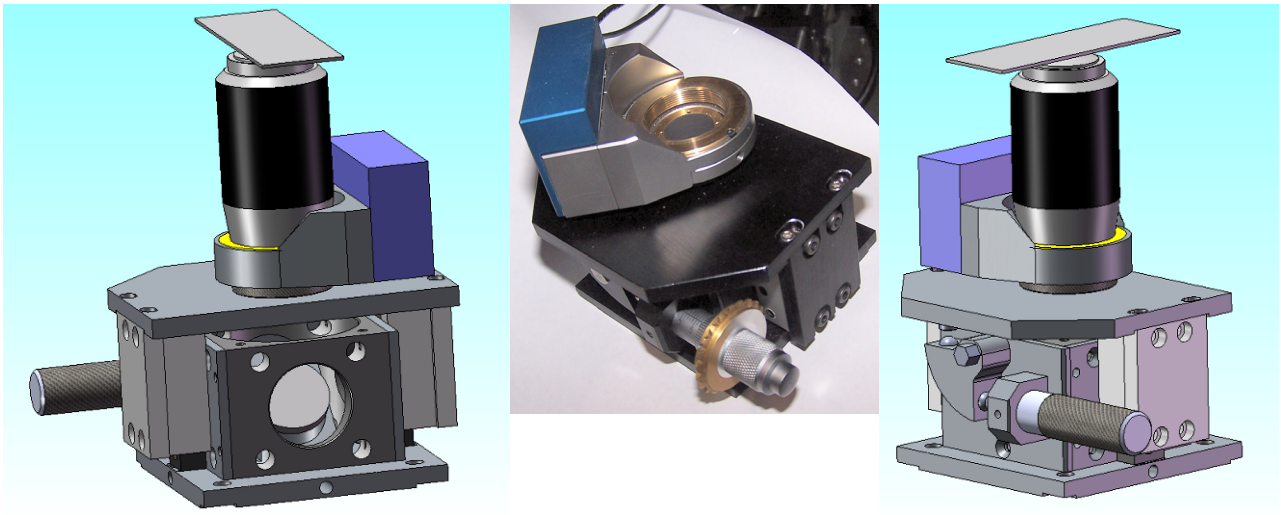


Figure 2: SolidWorks models of objective holder and rotary to linear converter.
The gears and belt drive from the motor & gearbox assembly to the objective holder are shown below. The electronics control box can just be seen attached to the support pillars.

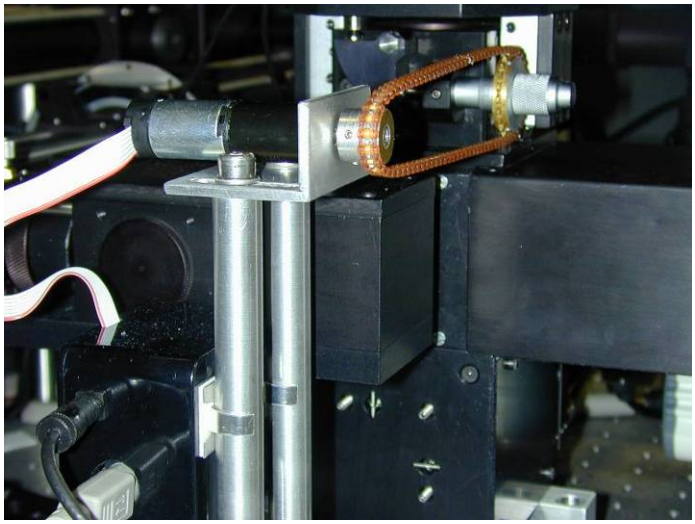
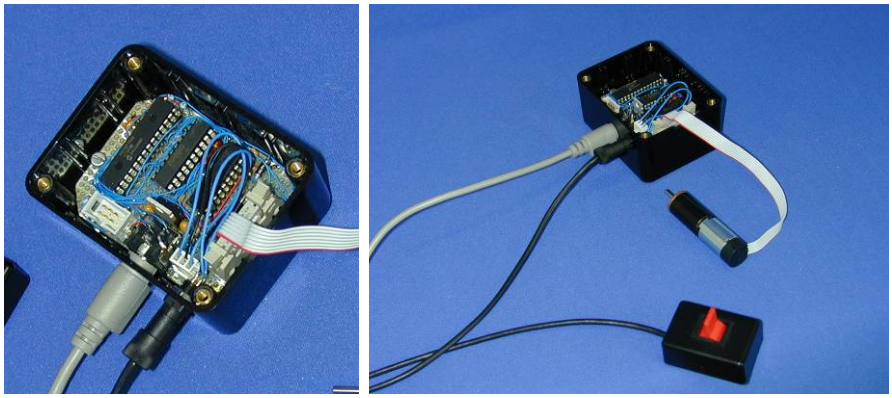


Figure 3. Construction of the position servo controller. An I^{2}C control cable and a small 3 pin cable are shown connected; the latter is used to connect an up-down jogging switch for manual operation. The 'flat' motor-encoder cable enters through a slit in the enclosure.



3. PIC firmware

The software used to drive the system is written in two sections: firmware running on the PIC microcontroller and high level host computer software. The firmware was written using a CCS C-code compiler (http://www.ccsinfo.com/) which makes generating the code considerably easier than using an assembler code language such as Microchip MPLAB. The sample code below may be found useful should future modifications be required.

The code controls the drive speed of the motor whilst reading the number of counts from the quadrature counter. A datum position is set and the distance required to 'drive down' is converted to a number of counts from the datum position. On the command to 'drive down' the motor is driven at the programmed speed until it reaches a distance (programmable) away from the required position. The speed is reduced when close to the required position to enable the motor to stop without overshoot. With the command to return to the datum position, the drive is reversed and the counter counts until the motor is near to the datum position. The motor then slows to stop at the datum. The motor is normally prevented from driving outside these positions except when the nudge up/down controls are used. These allow two types of motion. The first is small, slow movements at any time to allow fine positional control and setting of datum position. Due to inevitable backlash (slack in the drive belt, micrometer and bearings) when using these controls an anti backlash movement takes place so that the stop position is approached from the same direction. The second mode allows the motor drive continuously following an extended activation of the nudge up/down switches. These positioning commands can be operated over the software I^{2}C control or from a paddle switch which can either perform nudges or full up/down movements.

```
coarse z-drive.h file

#include <16F1936.h>
#FUSES MCLR                                     //Master Clear pin enabled
//#FUSES DEBUG                                  //Debug mode for use with ICD
//#device ICD=TRUE                              //Comment out these two lines when not debugging
#device adc=10                                  //A/D converter to 10 bit
#device *=16                                    //Extended memory
#FUSES NOWDT                                    //No Watch Dog Timer
#FUSES PLL                                      //If enabled PLL multiplies clock by 4
#FUSES INTRC_IO                                 //Internal RC Osc, no CLKOUT
#FUSES PUT                                      //Power Up Timer
#FUSES NOVCAP                                   //Pin for LDO decoupling capacitor
#FUSES NOLVP                                    //No low voltage prgming, B3(PIC16) or B5(PIC18) used for I/O
#use delay(clock=32000000)                      //Set delay functions for 32 MHz clock


//*****************************************************************************
// Compiler PCWH 4.101
// File:      coarse z-drive.c
// Date:      17/11/09
// Version: 2 added I2C interrupt functions to stop re-entry 02/09/10
// and delay when reading counter
// Code for coarse z-drive on HTS microscope using 16F1936 chip
//
//*****************************************************************************

#include "coarse z-drive.h"
```

```
#define RX_BUF_LEN  20
#define NODE_ADDR   0x84                        // Primary I2C address of the slave node

#use i2c(Slave,Slow,sda=PIN_C4,scl=PIN_C3,force_hw,address=NODE_ADDR)
#use fast_io (A)
#use fast_io (B)
#use fast_io (C)

//For 16F1936 registers
#byte PIC_CCPTMRS0=0x029E
#byte PIC_OSCCON=0x99
#define resolution 1023                         //define resolution steps of PWM
BYTE slave_buffer[RX_BUF_LEN];
BYTE state;

void ReadInput(void);
void i2c_interrupt_handler(void);
void Init(void);
void ExtUpDownCtrl(void);
void ReadPosition(void);
void GoUp(unsigned int16);
void I2CGoUp(unsigned int16);
void GoDown(unsigned int16);
void I2CGoDown(unsigned int16);
void Stop(void);
void I2CStop(void);
void ResetDatumPosition(void);
void ExtUpDownCtrl(void);
void IncStop(void);
void I2CIncStop(void);
void I2CReadPosition(void);

int1 inc_fg=0,move_fg=0,timer_fg=0,PWMset_fg=0,ExtUpDown_fg=0;
int buffer_index;
int mode=254,dir=1,stop_ind=0;

unsigned int16 timerCount=0,speed_offset=145,IncrementSpeed=200,backlash=5000;
unsigned int16 driveSpeed=400;
unsigned int32 msb_val=0,secondbyte_val=0,thirdbyte_val=0,lsb_val=0;
unsigned int32 position,addval1,lowerPosition=1999976360;
unsigned int32 IncPosition,downCounts=50000;                            //downCounts for 0.5mm
unsigned int32 offset=2147483648,setpointPosition=2147483648;
unsigned int32 incCounts=473;                                           //Counts for 10um
unsigned int32 approachCounts=10000;                                    //Counts left when slowing for position approach
unsigned int32 overshoot=0;                                             //Overshoot compensation for main move (programmable)
unsigned int32 inc_overshoot=8;                                         //Overshoot compensation for increment move

#int_RB
void RB_isr(void)    //Read external up/down control
{
   ExtUpDown_fg=1;
   #asm
   BCF    0x396.4                                                       //clear interrupt on change port B flags
   BCF    0x396.5
   #endasm
}
#int_SSP
void SSP_isr(void)
{
int16 downCountsLSB,speedLSB,IncrementSpeedLSB;
int16 backlashLSB;

   i2c_interrupt_handler();                                             //Interrupt happens on every byte received or sent

    if( state==5)                                                       //Six bytes received address discarded
    {
     mode=slave_buffer[0];
    switch(mode){
      case 0:        //Stop
      I2CIncStop();
      break;
      case 1:                                                           //Drive up/down
       dir=0;
      if((slave_buffer[1]==1) && (position < setpointPosition)){
         I2CGoUp(resolution);
         dir=1;
         move_fg=1;
         PWMset_fg=1;
      }
       if((slave_buffer[1]==0) && (position > lowerPosition)){
         I2CGoDown(resolution);
         dir=0;
         move_fg=1;
         PWMset_fg=1;
      }
      break;
      case 2:                                                           //Reset counter and datum
      output_bit (PIN_B2 , 0);
      output_bit (PIN_B2 , 1);
      I2CReadPosition();
      setpointPosition = position;
      lowerPosition = position - downCounts;
      stop_ind=2;                                                       //Stopped up indicator
      break;
      case 3:                                                           //Set increment counts
      incCounts = slave_buffer[1];
      incCounts = (incCounts<<8 | slave_buffer[2]);
      break;
      case 4:                                                           //Increment up/down
      if(slave_buffer[1]==1){    //Go up increment
        IncPosition=(position+incCounts-inc_overshoot);                 //Calculate counts with overshoot value
        dir=1;
        inc_fg=1;                                                       //Set increment flag
        I2CGoUp(IncrementSpeed) ;
      }
```

```
      if(slave_buffer[1]==0){                                          //Go down increment
          IncPosition=(position-incCounts-inc_overshoot);              //Calculate counts with overshoot value
          dir=0;
          inc_fg=1;                                                    //Set increment flag
          I2CGoDown(IncrementSpeed);
      }
      break;
      case 5:                                                          //Enable/disable  motor
      output_bit (PIN_C0 , slave_buffer[1]);
      break;
      case 6:                                                          //Down distance
      downCounts = slave_buffer[1];
      downCountsLSB = slave_buffer[2];
      downCounts = ((downCounts<<16) | (downCountsLSB<<8) |slave_buffer[3]);
      lowerPosition = 2147450000 - downCounts;                         //Sets lower position relative to datum position
      break;
      case 7:
      I2CStop();                                                       //Stop motor
      break;
      case 8:                                                          //Set slow drive speed
      speed_offset = slave_buffer[1];
      speedLSB = slave_buffer[2];
      speed_offset =  ((speed_offset <<8) |  speedLSB);
      break;
      case 9:                                                          //Set incrmental speed
      IncrementSpeed = slave_buffer[1];
      IncrementSpeedLSB = slave_buffer[2];
      IncrementSpeed  =  ((IncrementSpeed <<8) |  IncrementSpeedLSB);
      break;
      case 10:
      backlash = slave_buffer[1];
      backlashLSB = slave_buffer[2];
      backlash =  ((backlash <<8) |  backlashLSB);
      break;
      case 11:                                                         //Drive up/down
       if(slave_buffer[1]==1){                                         //Drive up
         I2CGoUp(driveSpeed);
       }
       if(slave_buffer[1]==2){                                         //Drive down
        I2CGoDown(driveSpeed);
        }
      break;
      case 12:                                                         //Set approach counts when near set point to slow down
      approachCounts = slave_buffer[1];
      approachCounts = ((approachCounts<<8) | slave_buffer[2]);
      break;
      case 13:                                                         //Set continious drive speed
      driveSpeed = slave_buffer[1];
      driveSpeed = ((driveSpeed<<8) | slave_buffer[2]);
      break;
      case 14:                                                         //Set overshoot value for maain move
      overshoot = slave_buffer[1];
      overshoot = ((overshoot<<8) | slave_buffer[2]);
      break;
      }
    }
}
//---------------------------------------------------------------------------
void ReadInput()
{
   switch(mode)
   {
     case 254:    //Read 4 bytes from counter and stop indicator
    //Return counter bytes and end position
      slave_buffer[buffer_index] = (addval1>>24);                      //Put val into buffer
      buffer_index++;
      slave_buffer[buffer_index] = (addval1>>16);                      //Put val into buffer
      buffer_index++;
      slave_buffer[buffer_index] = (addval1>>8);                       //Put val into buffer
      buffer_index++;
      slave_buffer[buffer_index] = addval1;                            //Put val into buffer
      buffer_index++;
      slave_buffer[buffer_index] = stop_ind;                           //Put val into buffer
      buffer_index++;
      break;
      case 255:
      //If needed to return other data
      break;

   }
}
//---------------------------------------------------------------------------
void i2c_interrupt_handler(void)
{
BYTE incoming;
unsigned int tx_byte;

   state = i2c_isr_state();

   if(state < 0x80)                                                    //Master is sending data
   {
      incoming = i2c_read();                                           //State 0 is address

   switch(state){
      case 1:                                                          //First data byte
         slave_buffer[(state-1)] = incoming;
         break;
      case  2:                                                         //Second data
         slave_buffer[(state-1)] = incoming;
           break;
      case  3:                                                         //Third data
         slave_buffer[(state-1)] = incoming;
           break;
      case  4:                                                         //Fourth
         slave_buffer[(state-1)] = incoming;
```

```
            break;
         case  5:                                                  //Fifth
            slave_buffer[(state-1)] = incoming;
            break;
         case  6:                                                  //Sixth
            slave_buffer[(state-1)] = incoming;
            break;
         case  7:                                                  //Seventh
            slave_buffer[(state-1)] = incoming;
            break;
      }
   }

   if(state == 0x80 )                                            //Master is requesting data
   {
      buffer_index = 0;                                          //Reset the buffer index
      ReadInput();                                               //Read bytes into buffer
      buffer_index = 0;                                          //Reset the buffer index
      tx_byte = slave_buffer[buffer_index];                      //Get byte from the buffer

#asm                                                             //Assembler for write command don't use MOVLB command in the
                                                                 //compiler use full address, compiler will bank automatically
   MOVF   tx_byte,W
   MOVWF  0xEA
   MOVF   0x211,W                                                //SSPBUFF into W
   MOVF   0xEA,W
   MOVWF  0x211
   nop                                                           //Delay before releasing clock
   nop                                                           //at 32 MHz 10 NOP's gives 1.25us
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   BSF    0x215.4                                                //Release clock (if set clock line released)
   BCF    0x11.3                                                 //Clear interrupt flag
TEST_BF:
   BTFSC  0x214.0                                                //Test BF bit (if set ready to transmit)
   GOTO  TEST_BF
   CLRF   0x278
   BTFSC  0x215.4                                                //If clock line not released (clear), skip
   INCF   0x278,F
   MOVLB  0x00
   MOVLW  0x00
 #endasm
      buffer_index++;                                            //increment the buffer index
      break;
   }
      if(state > 0x80 ){
         tx_byte = slave_buffer[buffer_index];                   // Get byte from the buffer

#asm                                                             //Assembler for write command don't use MOVLB command in the
                                                                 // compiler use full address, compiler will bank automatically
   MOVF   tx_byte,W
   MOVWF  0xEA
   MOVF   0x211,W                                                //SSPBUFF into W
   MOVF   0xEA,W
   MOVWF  0x211
   nop                                                           //Delay before releasing clock
   nop                                                           //at 32 MHz 10 NOP's gives 1.25us
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   nop
   BSF    0x215.4                                                //Release clock (if set clock line released)
   BCF    0x11.3                                                 //Clear interrupt flag
TEST_BF_1:
   BTFSC  0x214.0                                                //Test BF bit (if set ready to transmit)
   GOTO  TEST_BF_1
   CLRF   0x278
   BTFSC  0x215.4                                                //If clock line not released (clear), skip
   INCF   0x278,F
   MOVLB  0x00
   MOVLW  0x00
 #endasm
         buffer_index++;                                         //increment the buffer index

         break;
      }
}
//---------------------------------------------------------------
void I2CReadPosition()                                           //Read four bytes for position
{
 output_bit (PIN_B3 ,1);                                         //Disable outputs
 delay_us(10);                                                   //Delay to allow to latch
 output_bit (PIN_B3 ,0);                                         //Enable outputs  to latch counts
 output_bit (PIN_B0 ,0);                                         //MSB byte
 output_bit (PIN_B1 ,1);
 msb_val = input_a();

 output_bit (PIN_B0 ,1);                                         //2nd  byte
 output_bit (PIN_B1 ,1);
 secondbyte_val = input_a();

 output_bit (PIN_B0 ,0);                                         //3rd  byte
 output_bit (PIN_B1 ,0);
 thirdbyte_val = input_a();
```

```
 output_bit (PIN_B0 ,1);                                          //lsb  byte
 output_bit (PIN_B1 ,0);
 lsb_val = input_a();

 addval1 = ((msb_val<<24) | (secondbyte_val<<16) | (thirdbyte_val<<8) | (lsb_val));
 position = (addval1 + offset );

}
//-----------------------------------------------------------------------
void ReadPosition()                                               //Read four bytes for position
{
 output_bit (PIN_B3 ,1);                                          //Disable outputs
 delay_us(10);                                                    //Delay to allow to latch
 output_bit (PIN_B3 ,0);                                          //Enable outputs  to latch counts
 output_bit (PIN_B0 ,0);                                          //MSB byte
 output_bit (PIN_B1 ,1);
 msb_val = input_a();

 output_bit (PIN_B0 ,1);                                          //2nd  byte
 output_bit (PIN_B1 ,1);
 secondbyte_val = input_a();

 output_bit (PIN_B0 ,0);                                          //3rd  byte
 output_bit (PIN_B1 ,0);
 thirdbyte_val = input_a();

 output_bit (PIN_B0 ,1);                                          //lsb  byte
 output_bit (PIN_B1 ,0);
 lsb_val = input_a();

 addval1 = ((msb_val<<24) | (secondbyte_val<<16) | (thirdbyte_val<<8) | (lsb_val));
 position = (addval1 + offset );
}
//-----------------------------------------------------------------------
void GoUp(unsigned int16 up_speed)
{
   output_bit(PIN_C1 ,0);                                         //Pin C1 low direction pin
   set_pwm1_duty (up_speed);
   stop_ind=0;                                                    //Moving
}
void I2CGoUp(unsigned int16 up_speed)
{
   output_bit(PIN_C1 ,0);                                         //Pin C1 low direction pin
   set_pwm1_duty (up_speed);
   stop_ind=0;                                                    //Moving
}
void GoDown(unsigned int16 down_speed)
{
   set_pwm1_duty (resolution-down_speed);
   output_bit(PIN_C1 ,1);                                         //Pin C1 high direction pin
   stop_ind=0;    //Moving
}
void I2CGoDown(unsigned int16 down_speed)
{
   set_pwm1_duty (resolution-down_speed);
   output_bit(PIN_C1 ,1);                                         //Pin C1 high direction pin
   stop_ind=0;                                                    //Moving
}
void Stop(void)
{
unsigned int16 stop_speed=0;

   output_bit(PIN_C1 ,0);                                         //Pin C1 low direction pin
   set_pwm1_duty (stop_speed);                                    //PWM pin low
   if(dir==0){
      stop_ind=1;                                                 //Stopped down indicator
   }
   if(dir==1){
      stop_ind=2;                                                 //Stopped up indicator
   }
}
void I2CStop(void)
{
unsigned int16 stop_speed=0;

   output_bit(PIN_C1 ,0);                                         //Pin C1 low direction pin
   set_pwm1_duty (stop_speed);                                    //PWM pin low
   if(dir==0){
      stop_ind=1;                                                 //Stopped down indicator
   }
   if(dir==1){
      stop_ind=2;                                                 //Stopped up indicator
   }
}
void IncStop(void)
{
unsigned int16 stop_speed=0;

   set_pwm1_duty (stop_speed);                                    //PWM pin low
   output_bit(PIN_C1 ,0);                                         //Pin C1 low direction pin
   stop_ind=0;                                                    //Position indicators off
}
void I2CIncStop(void)
{
unsigned int16 stop_speed=0;

   set_pwm1_duty (stop_speed);                                    //PWM pin low
   output_bit(PIN_C1 ,0);                                         //Pin C1 low direction pin
   stop_ind=0;                                                    //Position indicators off
}
void incStopDown(void)
{
unsigned int16 stop_speed=resolution;

   set_pwm1_duty (stop_speed);                                    //PWM pin high
```

```
   output_bit(PIN_C1 ,1);                                          //Pin C1 low direction pin
   stop_ind=0;                                                     //Position indicators off
}
void backlashUp(void)                                              //After going down and overshooting, go up to correct position
{
     inc_fg=1;    //Set flag
     dir=1;
     GoUp(IncrementSpeed);
}
//*************************************************************************
 void ExtUpDownCtrl()    //Read up or down switch,pins RB5,RB4
 {
 int1 B4,B5;

 delay_ms(50);                                                     //Debounce
 B4 = input (pin_B4);
 B5 = input (pin_B5);

 if(B4 ==1 && B5==1 && move_fg==1){                                //Switch released and still moving
   IncStop();
   timer_fg=0;                                                     //Clear flag
   timerCount=0;     //Reset count
   move_fg=0;
 }
 if(B4 ==1 && B5==1 ){                                             //Finished moving to limit
   timer_fg=0;                                                     //Clear flag
   timerCount=0;                                                   //Reset count
 }
 if(B4 == 0){                                                      //Increment up
     IncPosition=(position+incCounts-inc_overshoot);               //Calculate counts with overshoot value
     inc_fg=1;                                                     //Set flag
     dir=1;
     GoUp(IncrementSpeed);
     timer_fg=1;
   }
 if(B5 == 0){                                                      //Increment down
     IncPosition=(position-incCounts-inc_overshoot);               //Calculate counts with overshoot value
     inc_fg=1;                                                     //Set flag
     dir=0;
     GoDown(IncrementSpeed);
     timer_fg=1;
   }
 }
//*************************************************************************
void ResetDatumPosition()                                          //Reset counter and datum
{
   output_bit (PIN_B2 , 0);
   output_bit (PIN_B2 , 1);
   ReadPosition();
   setpointPosition = position;
   lowerPosition = position - downCounts;
   stop_ind=2;                                                     //Stopped up indicator
}
//*************************************************************************
//Init() - routine
//*************************************************************************
void Init()
{
   set_tris_a (0xFF);                                              //All port A to inputs
   set_tris_b (0xF0);                                              //Set port B I/O
   set_tris_c (0x38);                                              //Set port C I/O 00111000
   PIC_CCPTMRS0 = 0x24;                                            //Set timers for CCP outputs 1,2,3

   setup_ccp1(CCP_PWM);                                            //Set for PWM1 output
   setup_ccp3(CCP_PWM);                                            //Set for PWM3 output

   set_pwm1_duty (512);
#asm //set_pwm3_duty (1);
  MOVLW  0x00
  MOVLB  0x06
  MOVWF  0x311                                                     //load MSB CCPRL3 register
  MOVF   0x313,W
  ANDLW  0xCF
  IORLW  0x10
  MOVWF  0x313                                                     //load LSB DCxB bits in CCP3CON register
#endasm                                                            //Using 32 MHz internal clock
   setup_timer_2(T2_DIV_BY_1,255,1);                               //Set period to 31.250 KHz resolution 0-1023 for motor control
   setup_timer_6(T2_DIV_BY_1,1,1);                                 //Set period to 4 MHz

   //Stop condition
   output_bit(PIN_C1 ,0);                                          //Pin C6 low direction pin
   set_pwm1_duty (0);                                              //PWM pin low

   output_bit (PIN_C0 , 1);                                        //Enable motors
   output_bit (PIN_B2 , 1);                                        //Reset pin high
   output_bit (PIN_B3 , 1);                                        //Output enable/ pin high
   port_b_pullups(TRUE);
   ResetDatumPosition();                                           //Reset position and datum
}
//*************************************************************************
//Main() - Main Routine
//*************************************************************************
void main()
{
   write_eeprom (255,0);                                           //Dummy eeprom write

   Init();                                                         //Initialize 16F1936 Microcontroller
   enable_interrupts(INT_SSP);
 #asm
 BSF    0x0B.3                                                    //enable interrupt on change port B and
 BSF    0x395.4                                                   //set bits 4,5 in register IOCBN
 BSF    0x395.5
 BSF    0x394.4                                                   //set bits 4,5 in register IOCBP
 BSF    0x394.5
 #endasm
```

```
   enable_interrupts(GLOBAL);

   while(1)                                                    //Loop Forever
   {
     if(ExtUpDown_fg==1){                                      //Check for external switch control
       ExtUpDownCtrl();
       ExtUpDown_fg=0;                                         //Clear flag
     }
      ReadPosition();

     if(move_fg==1){
      if(PWMset_fg==1){                                        //Change speed when near final position

         if((position>=(setpointPosition-approachCounts)) && (dir==1)){ //Check if close to up & slow down (approachCounts was=2000)
            set_pwm1_duty ( speed_offset);                     //Set to a lower speed
            PWMset_fg=0;      //Clear flag so only do once
         }
         if((position<=(lowerPosition+approachCounts)) && (dir==0)){ //Check if close to down and slow down (approachCounts was=2000)
             set_pwm1_duty (resolution - speed_offset);                 //Set to a lower speed
             PWMset_fg=0;       //Clear flag so only do once
         }
      }
        if((position >= (setpointPosition - overshoot)) && (dir==1)){ //Upper  limit
           Stop();
           move_fg=0;
        }
        if((position <= lowerPosition) && (dir==0)){           //Lower limit
           Stop();
           move_fg=0;
        }
     }
      ReadPosition();

      if(inc_fg==1){                                           //Increment position flag set
         timer_fg=0;
         if((position>=IncPosition) && (dir==1)){              //Check if increment up position reached
            IncStop();
            inc_fg=0;            //Reset flag
            timer_fg=1;
          }
         if(position<=(IncPosition - backlash) && dir==0){     //Check if increment down position reached minus backlash
            IncStopDown();
            inc_fg=0;           //Reset flag
            delay_ms(100);                                     //Allow to stop
            backlashUp();
           }
      }
      if(timer_fg==1){                                         //If external switch held closed, start counter
         timerCount=timerCount+1;
        if((timerCount==65000) && (input (pin_B4)==0)){
              GoUp(resolution);
              timer_fg=0;
              move_fg=1;
              PWMset_fg=1;
              timerCount=0;
         }
         if((timerCount==65000) && (input (pin_B5)==0)){
              GoDown(resolution);
              timer_fg=0;
              move_fg=1;
              PWMset_fg=1;
              timerCount=0;
              dir=0;                                           //Go down
         }
      }
   }
}
```

4. Host software

In addition to the PIC firmware, high level code running on a host is required. We first describe a stand-alone program which allows testing of the device and setting of appropriate parameters. We routinely use a National Instruments LabWindows CVI package, using C-code to write simple test programs which may also be integrated into larger programs. The interface between computer and device is by means of USB devices from FTDI Ltd. Using a combination of FTDI-USB drivers and a separate interface board described elsewhere, I^2C communications can be made to the servo unit.

The user interface for this code is shown in Figure 4. This panel allows the following operations to be performed:

On the set-up panel in Figure 4, on the right below, various adjustments can be made:

- The ‘increment’ distance is the distance of small travel in either direction.
- The ‘down distance’ is the total travel from the upper datum position to the lowered position
- The position indicator shows the objective’s actual position.
- ‘Drive speed’ is the maximum speed at which the motor runs.

- ‘Overshoot’ is the number of counts more than the required number to stop in the correct position. This number is subtracted from the calculated value to brake the motor slightly earlier. This is dependant on motor speed and loading.
- ‘nm/count is the calibration factor to convert the counts into distance in mm as described above, not set to correct value as no belt gears attached at this time.
- The ‘% approach’ speed is the speed of the motor when the distance to the desired position is within the approach distance.
- The ‘% inc’ speed is the speed of the motor when using the ‘nudge’ controls
- The ‘mm backlash’ is the distance moved beyond the small nudge movements so whether the movement is up or down, the target position is approached from the same direction eliminating backlash.
- After the working position of the objective has been found the ‘Datum’ button sets the lower position relative to this point.
- There are local Up/Down test buttons to drive to the upper and lower positions.

The user panel on the left has the basic controls.

- Objective ‘Up’ and ‘Down’ moves the objective to either the upper or lower positions.
- The ‘increment’ buttons nudge the objective up or down a small distance.
- Two indicators show when the upper or lower position has been reached.
- A position indicator shows the position of the objective.

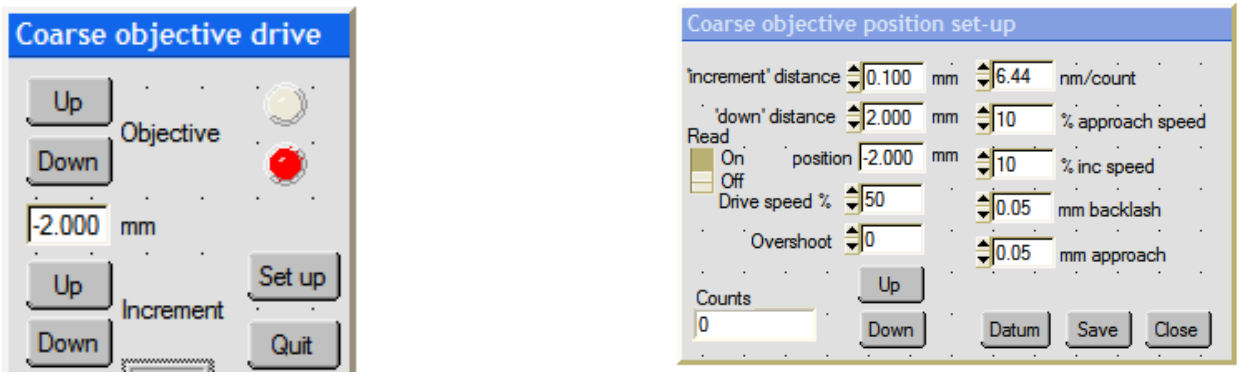


Figure 4. User interface panels used with the test code shown below.

```
#include "cvixml.h"
#include <rs232.h>
#include <ansi_c.h>
#include <cvirte.h>
#include <userint.h>
#include "utility.h"
#include "formatio.h"
#include  <analysis.h>
#include "DeviceFinder.h"
#include "focus drive_ui.h"
#include "IO_interface_v2.h"
#include "usbconverter_v2.h"
#include "DeviceManager.h"

#define bus 2

static int PORT;
static int ctrlPanel,setupPanel;
static int mode;
static double nm_count;

//int comPort(void);
//int setup(void);
static int initI2Cport(void);
static int sendPosition(int);
static int ManageDevice(char* device, int enable);
static void GCI_initFocusDrive(void);


int main (int argc, char *argv[])
{
          if (InitCVIRTE (0, argv, 0) == 0)
```

```
                    return -1;                                                          /* out of memory */
          if ((ctrlPanel = LoadPanel (0, "focus drive_ui.uir", CTRLPANEL)) < 0)
                    return -1;
          if ((setupPanel = LoadPanel (0, "focus drive_ui.uir", SETUP_PNL)) < 0)
                    return -1;

          if(initI2Cport()==-1)return 0;                                                //Set port number

          DisplayPanel (ctrlPanel);
          GCI_EnableLowLevelErrorReporting(1);
          Delay(1.0);
          GCI_initFocusDrive();
          RunUserInterface ();
          DiscardPanel (ctrlPanel);
          GCI_closeI2C();
          return 0;
}
static int getFTDIport(int *PORT)
{
char path[MAX_PATHNAME_LEN],ID[20];
int id_panel,pnl,ctrl;

                              //If we are using an FTDI gizmo Device Finder will give us the port number

          GetProjectDir (path);
          strcat(path,"\\");

          strcat(path, "focus driveID.txt");
          return selectPortForDevice(path, PORT, "Select Port for focus drive");
}
static int initI2Cport()
{
int  err,ans;
char port_string[10];

if (getFTDIport(&PORT) == 0)
                    sprintf(port_string, "COM%d",PORT);
          else {    //Device not found or not using FTDI. Use some other mechanism

                    while(getFTDIport(&PORT) != 0){
                              ans=ConfirmPopup ("Comms error", "Try plugging USB cable in or do you want to quit?");
                              if(ans==1){   //quit
                                        return -1;

                              }
                    }

          }
          sprintf(port_string, "COM%d",PORT);
          err = OpenComConfig (PORT, port_string, 9600, 0, 8, 1, 512, 512);

          SetComTime (PORT, 1.0);                                                       //Set port time-out to 1 sec
          FlushInQ (PORT);
          FlushOutQ (PORT);
          return 0;

}
static void GCI_initFocusDrive(void)
{
          RecallPanelState (setupPanel, "setupPanel.dat", 0);

          CallCtrlCallback (setupPanel, SETUP_PNL_DOWN_DISTANCE, EVENT_COMMIT, 0, 0, NULL);
          CallCtrlCallback (setupPanel, SETUP_PNL_INC_DISTANCE, EVENT_COMMIT, 0, 0, NULL);
          CallCtrlCallback (setupPanel, SETUP_PNL_DATUM, EVENT_COMMIT, 0, 0, NULL);
          CallCtrlCallback (setupPanel, SETUP_PNL_MIN_SPEED, EVENT_COMMIT, 0, 0, NULL);
          CallCtrlCallback (setupPanel, SETUP_PNL_MIN_INC_SPEED, EVENT_COMMIT, 0, 0, NULL);
          CallCtrlCallback (setupPanel, SETUP_PNL_BACKLASH, EVENT_COMMIT, 0, 0, NULL);
          CallCtrlCallback (setupPanel, SETUP_PNL_DRIVE_SPEED, EVENT_COMMIT, 0, 0, NULL);
          CallCtrlCallback (setupPanel, SETUP_PNL_OVERSHOOT, EVENT_COMMIT, 0, 0, NULL);

}

int SetIncDistance(void)
{
double incDistance;
char val1[20];
int cmd,intIncDistance,msbIncDistance,lsbIncDistance;

          GetCtrlVal(setupPanel, SETUP_PNL_INC_DISTANCE ,&incDistance);
          GetCtrlVal(setupPanel, SETUP_PNL_NM_COUNT ,&nm_count);
          intIncDistance=((incDistance*1000000)/nm_count);
          msbIncDistance= intIncDistance>>8 & 0xff;
          lsbIncDistance= intIncDistance & 0xff;

          cmd=3;                                                                        //Set increment
          val1[0] = FOCDRIVEPIC;
          val1[1] = cmd;
          val1[2] = msbIncDistance;
          val1[3] = lsbIncDistance;
          GCI_writeI2C_multiPort(PORT,6, val1, bus);
          return 0;


}

int SetDownDistance(void)
{
double downDistance;
char val1[20];
int cmd,intDownDistance,msbDownDistance,lsbDownDistance,lsb_1DownDistance;

          GetCtrlVal(setupPanel, SETUP_PNL_DOWN_DISTANCE ,&downDistance);
          GetCtrlVal(setupPanel, SETUP_PNL_NM_COUNT ,&nm_count);
          intDownDistance=((downDistance*1000000)/nm_count);
          msbDownDistance= intDownDistance>>16 & 0xff;
```

```
          lsb_1DownDistance= intDownDistance>>8 & 0xff;
          lsbDownDistance= intDownDistance & 0xff;

          cmd=6;                                                                    //Set down distance
          val1[0] = FOCDRIVEPIC;
          val1[1] = cmd;
          val1[2] = msbDownDistance;
          val1[3] = lsb_1DownDistance;
          val1[4] = lsbDownDistance;
          GCI_writeI2C_multiPort(PORT,6, val1, bus);
          return 0;

}

int CVICALLBACK cb_quit (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
char val1[20];

          switch (event)
                    {
                    case EVENT_COMMIT:

                    SetCtrlAttribute (ctrlPanel, CTRLPANEL_TIMER, ATTR_ENABLED, 0);       //Disable timer
                    SetCtrlVal(setupPanel, SETUP_PNL_READ_EN ,0);                         //Turn off read control
                    SavePanelState (setupPanel, "setupPanel.dat", 0);                     //Save panel values
                    QuitUserInterface (0);
                    break;
                    }
          return 0;
}

int CVICALLBACK cbtimer (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
char val1[20];
unsigned int cmd,msb,lsb_2,lsb_1,lsb,stop_ind;
//unsigned long int counts;
unsigned int counts;
double doublePosition;

          switch (event)
          {
                    case EVENT_TIMER_TICK:

                    cmd=254;
                    val1[0]=FOCDRIVEPIC;
                    val1[1]=cmd;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);

                    val1[0]=FOCDRIVEPIC  | 0x01;
                    if (GCI_readI2C_multiPort(PORT,5, val1, bus)) return -1;              //If problem

                    msb = val1[0] & 0xff;
                    lsb_2 = val1[1] & 0xff;
                    lsb_1 = val1[2] & 0xff;
                    lsb = val1[3] & 0xff;
                    stop_ind = val1[4] & 0xff;
                    SetCtrlVal(setupPanel, SETUP_PNL_FB_POSITION ,counts);
                    doublePosition = counts;
                    doublePosition=((doublePosition/1000000)*nm_count - (2000*nm_count)); //Offset taken off to get zero position
                    SetCtrlVal(ctrlPanel, CTRLPANEL_POSITION ,doublePosition);
                    SetCtrlVal(setupPanel, SETUP_PNL_POSITION ,doublePosition);

                    switch(stop_ind){                                                     //indicators
                    case 0:
                    SetCtrlVal(ctrlPanel, CTRLPANEL_LED_UP ,0);
                    SetCtrlVal(ctrlPanel, CTRLPANEL_LED_DOWN ,0);
                    break;
                    case 1:
                    SetCtrlVal(ctrlPanel, CTRLPANEL_LED_UP ,0);
                    SetCtrlVal(ctrlPanel, CTRLPANEL_LED_DOWN ,1);
                    break;
                    case 2:
                    SetCtrlVal(ctrlPanel, CTRLPANEL_LED_UP ,1);
                    SetCtrlVal(ctrlPanel, CTRLPANEL_LED_DOWN ,0);
                    break;
                    }
                    break;
          }
          return 0;
}

int CVICALLBACK cb_obj_up_inc (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
char val1[20];
int cmd;

          switch (event)
          {
                    case EVENT_COMMIT:
                    cmd=4;                                                            //Increment up or down
                    val1[0]=FOCDRIVEPIC;
                    val1[1]=cmd;
                    val1[2]=1;                                                        //Up
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
          }
          return 0;
}

int CVICALLBACK cb_obj_down_inc (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
```

```
char val1[20];
int cmd;

          switch (event)
          {
                    case EVENT_COMMIT:
                    cmd=4;                                                              //Increment up or down
                    val1[0]=FOCDRIVEPIC;
                    val1[1]=cmd;
                    val1[2]=0;                                                          //Down
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
          }
          return 0;
}

int CVICALLBACK cb_obj_up (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
char val1[20];
int cmd;

          switch (event)
          {
                    case EVENT_COMMIT:
                    cmd=1;                                                              //Move up or down
                    val1[0]=FOCDRIVEPIC;
                    val1[1]=cmd;
                    val1[2]=1;                                                          //Up
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;


          }
          return 0;
}

int CVICALLBACK cb_obj_down (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
char val1[20];
int cmd;

          switch (event)
          {
                    case EVENT_COMMIT:
                    cmd=1;                                                              //Move up or down
                    val1[0]=FOCDRIVEPIC;
                    val1[1]=cmd;
                    val1[2]=0;                                                          //Down
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
          }
          return 0;
}

int CVICALLBACK cb_set_up (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
          switch (event)
          {
                    case EVENT_COMMIT:
                    DisplayPanel (setupPanel);
                    break;
          }
          return 0;
}

int CVICALLBACK cb_inc_distance (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
double incDistance;
char val1[20];
int cmd,intIncDistance,msbIncDistance,lsbIncDistance;

          switch (event)
          {
                    case EVENT_COMMIT:
                    SetIncDistance();                                       //Set increment distance
                    break;
          }
          return 0;
}

int CVICALLBACK cb_down_distance (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
double downDistance;
char val1[20];
int cmd,intDownDistance,msbDownDistance,lsbDownDistance,lsb_1DownDistance;

          switch (event)
          {
                    case EVENT_COMMIT:
                    SetDownDistance();
                    break;
          }
          return 0;
}

int CVICALLBACK cb_nm_count (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
          switch (event)
          {
```

```
                    case EVENT_COMMIT:
                    GetCtrlVal(setupPanel, SETUP_PNL_NM_COUNT ,&nm_count);
                    SetIncDistance();                                       //Set increment distance
                    SetDownDistance();                                      //Set down distance
                    break;
          }
          return 0;
}

int CVICALLBACK cb_datum (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
char val1[20];
int cmd;

          switch (event)
          {
                    case EVENT_COMMIT:
                    cmd=2;                                                  //Reset datum
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
          }
          return 0;
}

int CVICALLBACK cb_save (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
          switch (event)
          {
          case EVENT_COMMIT:
          break;
          }
          return 0;
}

int CVICALLBACK cb_close (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
          switch (event)
          {
          case EVENT_COMMIT:
          HidePanel (setupPanel);
          break;
          }
          return 0;
}

int CVICALLBACK cb_set_min_speed (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
          switch (event)
          {
                    case EVENT_COMMIT:
                    {
          int value, lsb, msb;
          int cmd=8;                                                        //Set approach speed
          char val1[20];
          double d_value;

                    GetCtrlVal(panel, control ,&value);
                    d_value = value * (1023/100);                           //Percentage of 1023 max value
                    value = d_value;
                    msb= value>>8 & 0x00ff;
                    lsb= value & 0xff;
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    val1[2] = msb;
                    val1[3] = lsb;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                              break;
                    }
          }
          return 0;
}

int CVICALLBACK cb_set_min_inc_speed (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
          switch (event)
          {
                    case EVENT_COMMIT:
                    {
                    int value, lsb, msb;
                    int cmd=9;                                              //Set increment speed
                    char val1[20];
                    double d_value;
                    GetCtrlVal(panel, control ,&value);
                    d_value = value * (1023/100);                           //Percentage of 1023 max value
                    value = d_value;
                    msb= value>>8 & 0x00ff;
                    lsb= value & 0xff;
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    val1[2] = msb;
                    val1[3] = lsb;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
                    }
          }
          return 0;
}
```

```
int CVICALLBACK cb_backlash (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
int backlashcounts,msb_backlashcounts,lsb_backlashcounts;
char val1[20];
int cmd;
double backlash_val;


          switch (event)
          {
                    case EVENT_COMMIT:
                    {
                    GetCtrlVal(setupPanel, SETUP_PNL_BACKLASH ,&backlash_val);
                    backlashcounts = backlash_val/nm_count * pow (10,6);
                    msb_backlashcounts = ((backlashcounts>>8) & 0x00ff);
                    lsb_backlashcounts = (backlashcounts & 0xff);
                    cmd = 10;
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    val1[2] = msb_backlashcounts;
                    val1[3] = lsb_backlashcounts;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
                    }
          }
          return 0;
}

int CVICALLBACK cb_up (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
char val1[20];
int cmd;

          switch (event)
          {
                    case EVENT_COMMIT:
                    cmd = 0;                                                  //Stop command
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
                    case EVENT_LEFT_CLICK:
                    cmd = 11;
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    val1[2] = 1;           //Drive up
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
          }
          return 0;
}

int CVICALLBACK cb_down (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
char val1[20];
int cmd;

          switch (event)
          {
                    case EVENT_COMMIT:
                    cmd = 0;                                                  //Stop command
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
                    case EVENT_LEFT_CLICK:
                    cmd = 11;
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    val1[2] = 2;           //Drive down
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
          }
          return 0;
}

int CVICALLBACK cb_approachcounts (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
int approachcounts,msb_approachcounts,lsb_approachcounts;
char val1[20];
int cmd;
double approach_val;

          switch (event)
          {
                    case EVENT_COMMIT:
                    GetCtrlVal(setupPanel, SETUP_PNL_APPROACHCOUNTS ,&approach_val);
                    approachcounts = approach_val/nm_count * pow (10,6);
                    msb_approachcounts = ((approachcounts>>8) & 0x00ff);
                    lsb_approachcounts = (approachcounts & 0xff);
                    cmd = 12;
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    val1[2] = msb_approachcounts;
                    val1[3] = lsb_approachcounts;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
          }
          return 0;
}
```

```
int CVICALLBACK cb_drive_speed (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
          switch (event)
          {
                    case EVENT_COMMIT:
                    {
                    int value, lsb, msb;
                    int cmd=13;                                                    //Set manual drive speed
                    char val1[20];
                    double d_value;
                    GetCtrlVal(panel, control ,&value);
                    d_value = value * (1023/100);  `                               //Percentage of 1023 max value
                    value = d_value;
                    msb= value>>8 & 0x00ff;
                    lsb= value & 0xff;


                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    val1[2] = msb;
                    val1[3] = lsb;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
                    }
          }
          return 0;
}

int CVICALLBACK cb_overshoot (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
int overshoot_val,msb_overshoot_val,lsb_overshoot_val;
char val1[20];
int cmd;


          switch (event)
          {
                    case EVENT_COMMIT:
                    GetCtrlVal(setupPanel, SETUP_PNL_OVERSHOOT ,&overshoot_val);
                    msb_overshoot_val = ((overshoot_val>>8) & 0x00ff);
                    lsb_overshoot_val = (overshoot_val & 0xff);
                    cmd = 14;           //Overshoot command
                    val1[0] = FOCDRIVEPIC;
                    val1[1] = cmd;
                    val1[2] = msb_overshoot_val;
                    val1[3] = lsb_overshoot_val;
                    GCI_writeI2C_multiPort(PORT,6, val1, bus);
                    break;
          }
          return 0;
}

int CVICALLBACK cbread_en (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
int read_en;

          switch (event)
          {
                    case EVENT_COMMIT:
                    GetCtrlVal(setupPanel, SETUP_PNL_READ_EN ,&read_en);
                    SetCtrlAttribute (ctrlPanel, CTRLPANEL_TIMER, ATTR_ENABLED, read_en);     //Enable timer if 1 ,disable if 0
                    break;
          }
          return 0;
}
```

Similar user panels are used when this servo unit is part of a more complex software system, as used to control complete microscopes. These are shown in Figure 5. In this case the software has been rewritten to offer an application programming interface (API) to higher level code. Because there is no absolute position encoder, the higher level code assumes that the drive is always in the normal position when turned on. The drive is also always returned to the normal position at the end of a session. An alternative approach would be to save the position relative to the normal position in a file such that the position is known at restart, but in this case the user may be left with a long wait as the drive moves to the normal position at the start of their session (rather than at end of the last session). In either case, we cannot account for the user manually moving the drive with the toggle switch and turning off the device when the software is not running. Since the PIC code does not allow for the drive to be driven back to the normal position if it is already beyond that position, the higher level software drives to a small value inside the normal position before sending the command to drive to the normal position.

An example of this is shown overleaf:

```
static int atd_coarse_zdrive_destroy (CoarseZDrive* zd)
{
    int position=0;
    unsigned int counts;
    unsigned int stop_ind;


    // move to normal position on exit, so that next time this remains the datum
    // But make sure it gets to the datum, even if starting from above the datum
    // by driving down to some -ve value and then up again
    // (hw ignores commands to go to top if already above the top)
    coarse_zdrive_move_to_bottom(zd);
    do {
        zdrive_read_position(zd, &position, &counts, &stop_ind);
    } while (position > -100);
    coarse_zdrive_move_to_top(zd);

    #ifdef FTDI_NO_VIRTUAL_COMPORT
    ftdi_controller_close(zd->controller);
    #else
    close_comport(zd->_com_port);
    #endif

    return COARSE_Z_DRIVE_SUCCESS;
}
```

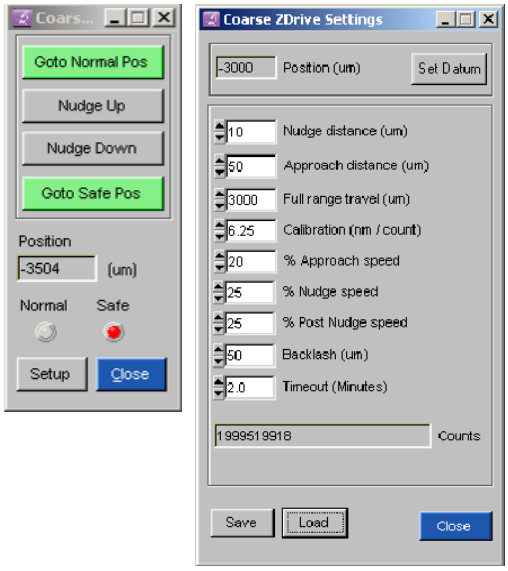


Figure 5: User interface panels of the higher level code which has the common commands on a main panel and a setup panel.

This note was prepared in March 2010 by B. Vojnovic and RG Newman, who also constructed the hardware and developed the PIC firmware. The note was updated in August 2011 by PR Barber, who also developed the host software, in collaboration with G. Pierce. Thanks go to John Prentice for machining the various mounts.

We acknowledge the financial support of Cancer Research UK, the MRC and EPSRC.