PELS3

Donald Plewes -June 2016

Introduction

PELS3 is an extension of the PELS1 Arduino based Electronic Lead Screw (ELS) system which was outlined in my paper published in Digital Machinist in 2016 (An Electronic Lead screw for a D6000 Wabeco Lathe, D.B Plewes, Digital Machinist, Vol 11, No.4, 6-19, 2016). I would urge anyone who is interested in building this project to review this paper first. I will briefly summarize PELS1 and then point out the difference with PEL3. The PELS3 code is shown as the end of this README document.

PELS1

PELS1 was the first version of this software and enables one to conduct Imperial threading and conventional turning on a D6000 bench-top lathe which normally requires gear or pulley changes to provide different threading options. This hardware/software permits all common Imperial threading options to be performed, which include 10, 11, 12, 13, 16, 20, 24, 28, 32, 36, 40, 44, 48 and 52 threads/inch. In addition it provides conventional turning at three pitches/spindle-revolution of: 'normal' (0.0019 inches), 'coarse' (0.0039 inches) and 'fine' (0.00150 inches). The carriage advance is achieved by using a 1024 step/revolution quadrature optical encoder which is geared down via pulleys to a value of 635 pulses/spindle-revolution. This is used in conjunction with a 1.8^o stepper motor with a 2:1 speed reduction to drive the Wabeco D6000 lead screw (4mm pitch). With this arrangement, all Imperial threads can be achieved by dividing the encoder pulse train by an integer which is equal to the desired number of threads/inch (tpi). For example, a 24 tpi thread is achieved by dividing the pulse stream by the integer 24. The pulse stream division and pulse generation is performed by an Arduino Uno R3 using a simple interrupt routine.

PELS3

PELS3 uses a slightly different approach to threading that allows it to be applied to any lathe. Rather than using a counting operation, as in PELS1, it uses floating point arithmetic and other non-linear operations to create stepper pulses. There is no need to have a specific encoder or stepper resolution. The only constraint is that the number of pulses to drive the lead screw one revolution for the coarsest thread pitch must be less than that provided by the spindle encoder for one revolution.

The code is composed of 3 routines plus the standard parameter definitions section and setup() components required for all Arduino code. These routines are count(), thread_parameters() and loop(). I will describe each of these in turn.

Parameter Definition

This section is straightforward and is well commented. It does require two libraries as follows:

Wire.h

This can be downloaded from https://www.arduino.cc/en/reference/wire

Adafruit_RGBLCDShield.h

This can be downloaded from https://learn.adafruit.com/rgb-lcd-shield/using-the-rgb-lcd-shield

It is in this section that one can alter the parameters of PEL3 to match the requirements of a different lathe. As currently defined it is set up for the D6000 lathe. The code is shown as:

```
int spindle_encoder_resolution=1024;   // the number of pulses per revolution of the spindle encoder
float lead_screw_pitch=4.0;       // the pitch of the lathe lead screw in mm
 int motor_steps=400;               // the number of steps per revolution of the lead screw
float pitch=0.085;                  //a parameter to indicate the pitch in mm's
```

The first parameter defines an integer parameter called *spindle_encoder_resolution* which is current set to 1024. This is the number of pulses per revolution of the spindle encoder.

The second parameter defines a floating point number which measures the lead screw pitch in millimeters. It is called *lead_screw_pitch* and is currently set for a pitch of 4mm.

The third parameter defines an integer parameter *motor_steps* which is the stepper resolution of the lead screw. I am assuming a 1:2 gear reduction between a 1.8^{o} stepper motor to the lead screw which give a lead screw resolution of 400 steps/revolution.

The final parameter *pitch* sets up the initial value for normal turning when the Arduino is first powered on. It is currently set to deliver 0.085mm per spindle revolution.

These four parameters can be edited to suit the needs of different lathes.

The rest of the parameters in this section should not require any changes.

Setup()

This section creates a number of pinMode definitions, starts the LCD display and attaches the interrupt routine to digital pin 2 of the Arduino Uno which links to the routine *count()* . This section defines a parameter called *factor* which sets up the lathe for normal turning when the unit first powers on.

Count()

This is the heart of ELS3 and generates the stepper pulses. The interrupt routine detects a spindle pulse and increments a parameter called *input_counter.* We also keep track of the number of stepper pulses we have delivered through this routine through the parameter *delivered_stepper_pulses*. This parameter is defined as a long integer and is volatile so that it can be used in the interrupt routine. We then calculate the number of stepper pulses which is given from this by the line:

*calculated_stepper_pulses=round(factor*input_counter);*

Note, that we do a floating point multiplication of the parameter *factor* and *input_counter* . Then we do a *round* operation on this product to generate the number of stepper pulses which should have been delivered. The next line:

if((calculated_stepper_pulses>delivered_stepper_pulses)&&(mode_select==0))

This conditional statement compares the *calculated_ stepper_pulses* to the *delivered_stepper_pulses*. We have two conditionals linked through an AND operation (shown as *&&)*. The first part is to see if the *calculated_pulses* is greater the *delivered_stepper_pulses*. The second part is to see if *mode_select* is set to the *lathe* setting (*mode_select==0)*. If both of these are TRUE, we generate a 10 microsecond pulse on the *stepper_pin* line. Otherwise we do nothing and just keep counting spindle pulses with *input_counter* until the conditional becomes TRUE. The AND operation with *mode_select* ensures that we cannot generate pulses when the ELS is in the programming mode.

Thread_parameters()

This section defines the parameters we need for the floating point variable *factor* used in *count()*. It starts by detecting when the push button on the ELS electronics is depressed. Pressing this button will toggle a parameter called *mode_select* . This parameter determines if the ELS is set to a programming ("prog") or operational ("lathe") mode. In programming mode, we can adjust the thread or turning parameters by turning the knob. Once we have found the correct setting, we push the button to the "lathe" setting to enable the *count()* routine to generate stepper motor pulses for these new parameters. This is a safety feature that ensures that we cannot inadvertently alter the lathe parameters while the lathe is running. We need to actively toggle to the system from one mode to the other. While in programming mode, stepper pulses cannot be generated.

The next section developed a quadrature encoder routine which increments a parameter called *menu*. This parameter will increment or decrement depending on which way the knob is rotated. The parameter *menu* is an integer which is used to select a value for factor. This parameter is used in a *switch(menu)….break;* structure which permits selecting the *factor* parameter.

The first three values of menu (1,2 or 3) selects among the *factor* needed when running the lathe for turning. There are three values I have chosen for a "normal", "fine" and "coarse" turning operation. These match the settings for the stock D6000 lathe. These can be edited if different values are needed.

The menu values between 4 and 18 selects for Imperial threads of pitch from 11→52 threads per inch.

The menu values between 19 and 34 select for Metric threads of pitch from 0.4→7.0mm.

If one wanted to add a new thread option, it can be added here by adding a new case.

The next few lines do the calculation of the *factor* parameter based on either the *pitch* or *tpi* parameters found through the *switch* commands. If we choose an Imperial thread, we use *tpi* to determine the value of *factor*. Alternatively, if we choose a Metric thread, we use *pitch* to determine the vaue of *factor*. In these sections we display the parameters we have chosen on the lcd display.

I have included another factor call *depth* which is also displayed on the LCD. This number is a product of the *pitch_factor* and *pitch*. This gives the depth we need to move the angled cross-slide for a given thread. This assumes that we have the lathe cross-slide angled at 59.5 degrees and we advance the cross-slide for consecutive thread cuts. It assumes that the proper cross-slide depth for a thread of pitch *p* is 0.75*p*. This avoids having to look up the cross-slide distance in a book for each thread as it is reported on the LCD screen. I have found it a useful addition. As this number is only approximate, I use it as a guide only. However if one wanted to have accurate values for each thread, one could include accurate values in the *switch…break* structure along with *tpi* or *pitch* as a new parameter for each thread.

Loop()

This runs the thread_parameters() code to detect and change the thread parameters through the rotary switch. Note that count() is not referenced here as it runs independently as an interrupt routine.

How the Algorithm Works

It is interesting to take a particular case and observe how the stepper pulses are delivered. Assume that we needed to cut a thread and we required 3.5 spindle pulses per stepper pulse to deliver this thread. In this case we would have the value D=3.5 or k=1/D=0.2857. Let's further assume that we have a lead screw pitch of 4mm, a lead screw resolution of 400 steps and a spindle encoder of 1024 steps/revolution. Figure 1, shows a plot of stepper pulses with spindle pulses that would be generated by PELS3.

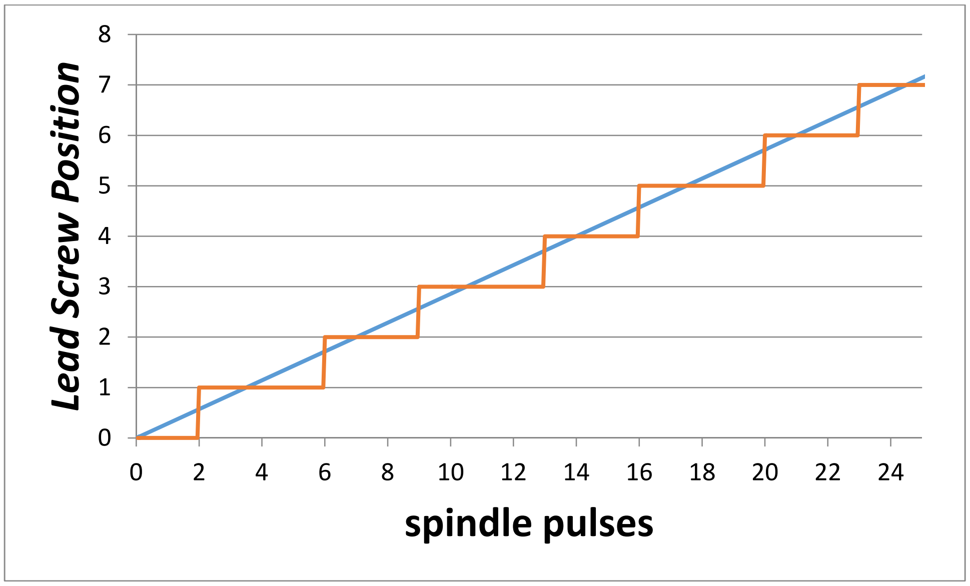

Figure 1 Lead screw pulses versus spindle pulses. The ideal curve (blue) is a linear graph with a slope of 1/3.5 .The stepper pulses approximate this curve by the stepper motor pulses (orange). The stepper motor pulses occur at the sharp transitions in the orange curve.

This plot shows the result for the first 25 spindle pulses as a plot of lead screw pulses for each spindle pulse. Ideally, we would like our lead screw to rotate in a continuous manner as a function of spindle pulses which would generate the blue line. However, the lead screw can only move in a stepped manner as it is attached to a stepper motor. Thus the actual motion will be that shown by the red line. We see that the stepper pulses occur at the transitions between the staircase edges. The 1st stepper pulse occurs after the second spindle pulses, the 2nd stepper pulse after the 6th spindle pulse, the 3rd after 9 , the 4th after 13, the 5th after 16, the 6th after 20 and the 7th after 23. We note the difference between the number of spindle pulses for consecutive stepper pulses goes as follows in the following table:

Stepper pulse #	Previous spindle pulses	Current spindle pulse	Difference in spindle pulse
1	0	2	2
2	2	6	4
3	6	9	3
4	9	13	4
5	13	16	3
6	16	20	4
7	20	23	3

If we ignore the first stepper pulse, we see that the number of spindle pulses for each consecutive stepper pulse is given by the sequence 4,3,4,3,4,3. If we were to consider a larger number of spindle pulses this sequence would continue indefinitely. So, we have a repeating sequence of divisors of 4→3→4→3…. That means that the ELS is counting first 4 spindle pulses, then 3 to give the average divisor of (3+4)/2=3.5, which is the desired divisor. One might think that this would generate a shifting error. To see the error more clearly, we plot the difference between the ideal lead screw position and that which is delivered to the lead screw by the stepper motor as shown next in Figure 2. .

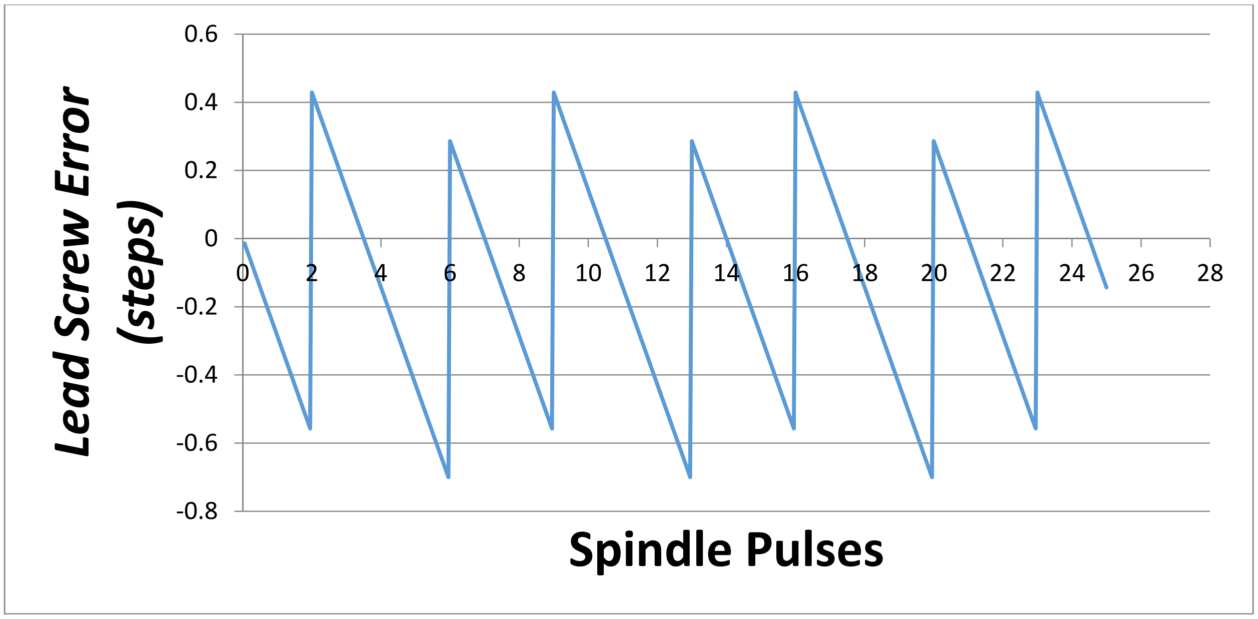

Figure 2. Lead Screw positional error versus spindle pulses

We see a sawtooth pattern which shows that the error oscillates around the correct value by less than a single lead screw step. We see again that the stepper pulse are generated at spindle pulses, 2,6,9,13,16,20 and 23.   One could argue that this is a special case as the divisor is a simple number.

What about a more arbitrary number like 3.666 which could be a possible divisor? In this case, we would get the next plot.

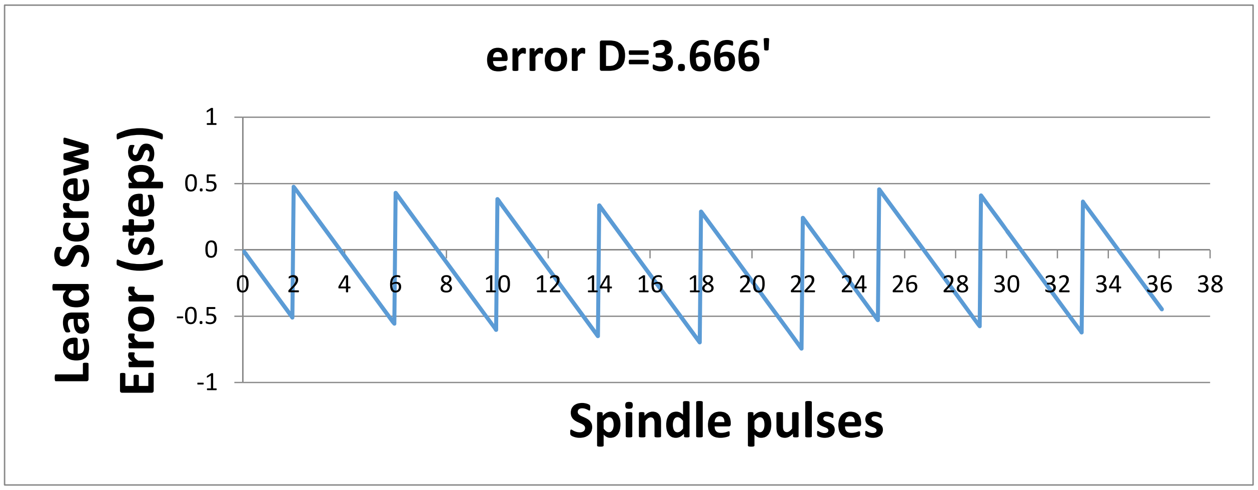

Figure 3. Lead Screw Error for a divisor of 3.66' vs spindle pulses.

The sequence for the first 10 stepper pulses would be;

Stepper pulse #	Previous spindle pulses	Current spindle pulse	Difference in spindle pulse
1	0	2	2
2	2	6	4
3	6	10	4
4	10	13	3
5	13	17	4
6	17	21	4
7	21	24	3
8	24	28	4
9	28	32	4
10	32	35	3

Now, we see the repeating sequence is 4→4→3 which gives an average divisor of (4+4+3)/2=3.666.

So, now we see that the way the system works is to create sequences of divisors on a real-time basis.

The first case (3.5) showed a simple sequence of 3→4 to give a divisor of 3.5.

The second case (3.6’) showed a sequence 4→4→3 to give a divisor of 3.6’.

Other more arbitrary numbers would show more complex sequences to give the correct divisors. Inspection of these sequences show that they might be a simple repeating sequence like 3→4→3→4 or they might be more complex and formed of sequence of sequences. The net consequence is that whatever numbers of spindle pulses are needed to keep the correct lead screw resolution are delivered on a real-time basis.

Testing PELS3 for Accuracy

Once the ELS is assembled and ready for testing, I would recommend this simple method for assessing its accuracy assuming you have a DRO attached to your lathe. It will not give the best accuracy but will indicate whether the code is working correctly without having to actually cut a thread and test it against a gauge. The idea is to rotate the spindle by hand exactly 1 revolution and measure the distance the carriage moves. This should match the pitch you chose with the ELS.

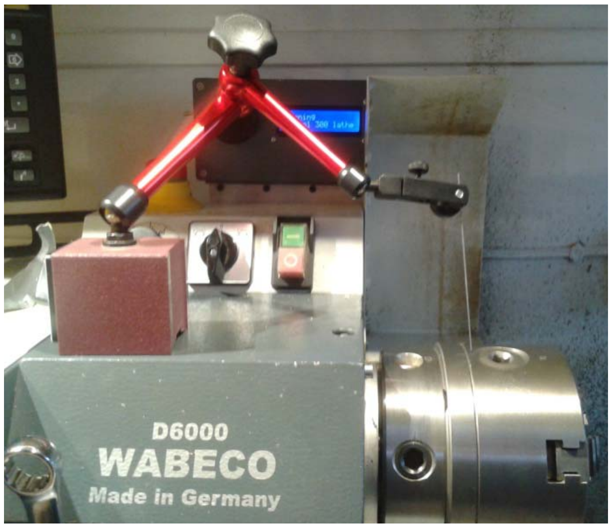

Figure 4 position a wire so that it can mark the position of the spindle.

To do this I used my magnetic indicator dial holder which held a fine wire. I then found an alignment mark on my cam-lock spindle and positioned the tip of the wire just above this mark as shown in Figure 4.

Placing the tip of the wire close to this mark allows one to reposition the lathe spindle with considerable accuracy. This allows one to rotate the spindle through to one revolution fairly accurately as shown in Figure 5.

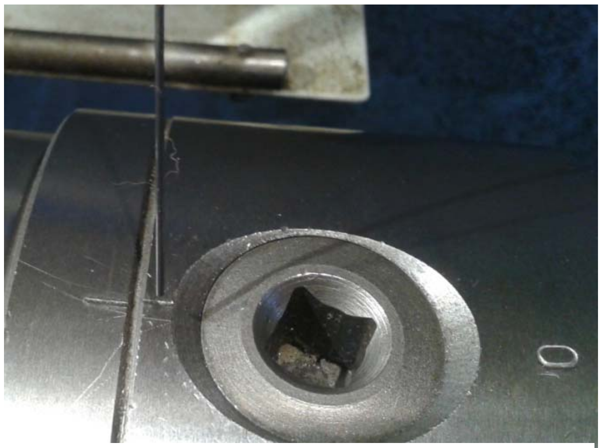
Figure 5. The wire tip pointing to an alignment mark on spindle.

Now choose a thread pitch value for testing. Here I am choosing values of 7.0 and 2.0mm. Next it is necessary to back out the backlash on the leadscrew by rotating the spindle by hand with the ELS in the “lathe” mode. You will hear the stepper motor moving the lead screw with pulses. Rotate the spindle in the cutting (forward) direction, until the spindle mark aligns with the wire tip. Next, we zero out the DRO as shown in Figure 6. Now, we carefully rotate the spindle by hand in the forward direction until the wire is once again aligned with the spindle mark. The DRO should indicate the pitch of the thread after this carriage motion. As shown in Figure 6, we get a value of 7.01mm and 2.01 which are within .01mm (0.00039”) as expected. This gives a very quick indication of whether the

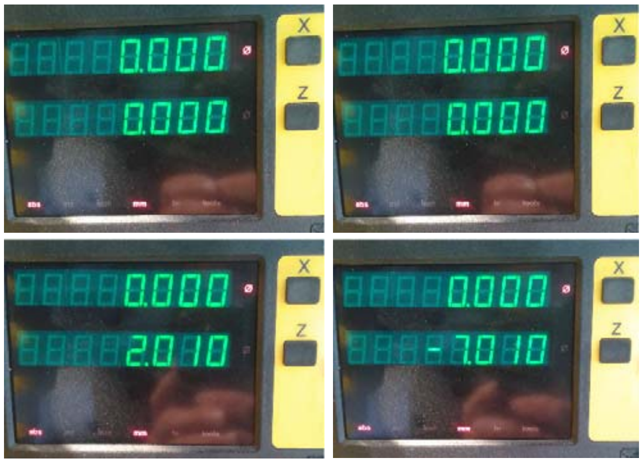

Figure 6. Zeroing the DRO with the wire aligned to the mark (top). After one rotation of the spindle by hand to show a carriage translation of 2.01 mm (bottom- left) and 7.01 mm (bottom right)

system is working without resorting to actually cutting a thread and measuring it against a thread gauge. A more accurate assessment of the errors could be achieved by doing several rotations of the spindle and then dividing the displacement by the number of rotations to get the pitch.

PELS3 Code

```
/* PELS3 - Universal Electronic Lead Screw Code. This code is to be used on and Electronic Lead Screw
outlined in: "An Electronic Lead screw for a D6000 Wabeco Lathe", by D.B Plewes,
  Digital Machinist, Vol 11, No.4, 6-19, 2016). This uses a 1024 pulse/spindle-rev encoder on the D6000
lathe and assumes a 400 steps/revolution attached to the lead screw.
  This uses floating point arithmetic by comparing the desired thread position and compares this against
the discrete position which has been delivered by the stepper motor.
  The code updates the step pulses as needed to keep the same position between the desired position at
that delivered. The selection between metric and imperial threads is
  achieved by a push-button on a rotary encoder while the value of the thread/turning parameters are
achieved by rotating the rotary switch.

 The inputs to the Arduino which are needed is as follows:
  - knob rotary encoder - Q to Arduino pin 4, I to Arduino pin 5, button to Arduino pin 6.
  - spindle rotary encoder goes to LS7184 chip.  Output from pin 6 of LS7184 goes to Arduino pin 2.
  - input to step input of Gecko stepper motor controller comes from Arduino pin 13.
  - output from Arduino to change the output_pulses/input_pulses ratio to the LM7184 chip via line
'range_select_pin'

The button is used to put the ELS into a programming mode so that the menu parameter can be
changed.  Once the parameter is found we push the button again to lock in this parameter
so that it can't change during operation. All the thread parameters are put into one big menu file which
includes: Turning, Imperial threads and Metric threads. The approximate cut depth
for the cross-slide set at 59.5 degrees are shown on the LCD display for each thread size in mm's.

 This program is free software. It can be redistributed and/or modified under the terms of the GNU
General Public License as published by the Free Software Foundation.
 This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without
even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
 PURPOSE.  See the GNU General Public License for more details. You can read a copy of the GNU
General Public License at http://www.gnu.org/licenses/.
  */

  #include <Wire.h>
  #include <Adafruit_RGBLCDShield.h>
 Adafruit_RGBLCDShield lcd = Adafruit_RGBLCDShield();

 // User parameters to be altered depending on lathe parameters....................................................
 // To use this code one must replace these parameters with correct values for your lathe. These are
'spindle_encoder_resolution','lead_screw_pitch' and 'motor_steps'.
 // For example, if you had a spindle encoder of 200 step/rev... then edit the current
'spindle_encoder_resolution=1024' to read 'spindle_encoder_resolution=200'.
 // The applies to the 'lead_screw_pitch parameter' and the 'motor_steps' parameter. I have used
numbers which apply to a Wabeco D6000 lathe. You must edit for your particular lathe.

 int spindle_encoder_resolution=1024;   // the number of pulses per revolution of the spindle encoder
 float lead_screw_pitch=4.0;           // the pitch of the lathe lead screw in mm
 int motor_steps=400;                 // the number of steps per revolution of the lead screw
 float pitch=0.085;                  // the pitch to be cut in millimeters.  It also defines the lathe pitch for
turning when first power on.

 //.............................................................................................................
 // System parameters which need no alteration.
 int stepper_pin=13;                           //the output pin for the stepper pin for the motor driver
 int encoder0PinA = 5;                          //the input pin for knob rotary encoder 'I' input
 int encoder0PinB = 4;                          //the input pin for knob rotary encoder 'Q' input
 int buttonPin=6;                             // the button for the knob rotary encoder push button line
 int range_select_pin=7;                        // the pin for the output to drive the LS7184 from mode 0 to
mode 2, permanently set to LOW for this code. Legacy from PELS1
 int menu = 1;                                // the parameter for the menu select
 int encoder0PinALast = LOW;                     //a dummy parameter for the rotary encoder quad select
algorithm
 int n = HIGH;                                //another dummy parameter for the rotary encoder quad select
algorithm
 int newButtonState = 0;                        //a parameter for the button push algorithm
 int oldButtonState=0;                          //a parameter for the button push algorithm
 int mode_select=1;                            // a parameter to define the programming versus operation
settings

 float depth;                                 // a parameter to define the thread depth in mm on the compound
slide. This is set at 75% of the pitch which seems to work
 float pitch_factor=0.75;                        // a parameter to define how deep to push the oblique cutter for
each thread pitch in mm. May differ depending on thread design. This one works
 volatile long input_counter=0;                  //a parameter for the interrupt to count input pulses
 volatile float factor;                         // the ratio of needs steps/rev for stepper to
spindle_encoder_resolution for each thread pitch we pick, this is calculated in the programme
 volatile long delivered_stepper_pulses=0;        //number of steps delivered to the lead screw stepper
motor
 volatile float calculated_stepper_pulses=0;       //number of steps we should have delivered for a given
lead screw pitch

 void setup()
 {
   pinMode (encoder0PinA,INPUT);                    //input for the Q channel of the switch rotary encoder
   pinMode (encoder0PinB,INPUT);                    //input for the I channel of the switch rotary encoder
   pinMode (buttonPin,INPUT_PULLUP);                  //input for the button of the switch rotary encoder
```

```
  pinMode (range_select_pin,OUTPUT);                // set up digital pin "range_select_pin" to digital
output to change the state of the LS7184 quad chip with HIGH giving 4x pulse number and LOW giving
1x pulse number. Legacy from PELS1
  lcd.begin(16,2);                              // initializes the LCD display
  pinMode(stepper_pin, OUTPUT);                     // sets up the stepper_pin as an output for the stepper
pulses
  attachInterrupt(0, count, RISING);                // enable the interrupt for Arduino pin 2 which is
interrupt "0"
  digitalWrite(range_select_pin,LOW);               // sets ground to pin 6 of LS7184 chip to range select
for LS7184 chip for a factor of 1.
  factor= (motor_steps*pitch)/(lead_screw_pitch*spindle_encoder_resolution); //initial factor when
turning on the ELS to deliver a turning operation of "medium" pitch


 //................This next section starts the system to Nnormal Imperial
Turning.............................................
  lcd.setCursor(0,0);
  lcd.print("Turning");
  lcd.setCursor(0,1);
  lcd.print("Normal    ");
 }



 void count()    //this is the interrupt routine for the floating point division algorithm
  {
   input_counter++;                                          // increments a counter for the number of spindle
pulses received
   calculated_stepper_pulses=round(factor*input_counter);                 // calculates the required
number of stepper pulses which should have occured based on the number spindle pulses
(input_counter number)
   if((calculated_stepper_pulses>delivered_stepper_pulses)&&(mode_select==0))  // if the calculated
number of pulses is greated than the delivered pulses, we deliver one more stepper pulse only if
mode_select is set for lathe (==0)
      {
     digitalWrite(stepper_pin,HIGH);                       // turns the stepper_pin output pin to HIGH
     delayMicroseconds(10);                            // keeps that level HIGH for 10 microseconds
     digitalWrite(stepper_pin,LOW);                        // turns the stepper_pin output pin to LOW
     delivered_stepper_pulses++;                         // increment the number of
delivered_stepper_pulses to reflect the pulse just delivered
     }
  }

 void thread_parameters()                              //this defines the parameters for the thread and
turning for both metric and imperial threads
 {
  newButtonState = digitalRead(buttonPin);                  // Get the current state of the button
```

```
     if (newButtonState == HIGH && oldButtonState == LOW)     // Has the button gone high since we last
read it?
        { mode_select=!mode_select;}

        if (mode_select == 0)                              //mode_select==0 for lathe operation which I call "lathe"
         {
        lcd.setCursor(11,1);
        lcd.print("lathe");
          }
         else
          {
           mode_select=1;
         lcd.setCursor(11,1);                            // mode_select==1 for parameter selection which I call
"prog" for programme
         lcd.print(" prog");
          }
        oldButtonState = newButtonState;

  if(mode_select==1)
         {
          n = digitalRead(encoder0PinA);                                    //Selecting the Thread and Turning
Parameters
               if ((encoder0PinALast == LOW) && (n == HIGH)) {                //true if button got pushed?
                    if (digitalRead(encoder0PinB) == LOW) {                  //this is the quadrature routine for the
rotary encoder
                    menu++;
               } else {
                    menu--;
               }
               Serial.println(menu);
                if(menu>35){                                  //the next four lines allows the rotary select to go around
the menu as a loop in either direction
                         menu=35;
                         }
                if(menu<1){
                         menu=1;
                         }
                            switch(menu) {
                                      case(1):     pitch=0.085;               break;  // Normal Turning
                                      case(2):     pitch=0.050;               break;  // Fine Turning
                                      case(3):     pitch=0.160;               break;  // Coarse Turning
                           //..........................................................................................imperial data
                                      case(4):     tpi=11;   break;
                                      case(6):     tpi=12;   break;
                                      case(7):     tpi=13;   break;
                                      case(8):     tpi=16;   break;
                                      case(9):     tpi=18;   break;
                                      case(10):    tpi=20;   break;
```

```
                                    case(11):   tpi=24;   break;
                                    case(12):   tpi=28;   break;
                                    case(13):   tpi=32;   break;
                                    case(14):   tpi=36;   break;
                                    case(15):   tpi=40;   break;
                                    case(16):   tpi=42;   break;
                                    case(17):   tpi=44;   break;
                                    case(18):   tpi=48;   break;
                                    case(19):   tpi=52;   break;
                        //.......................................................................................metric data
                                    case(20):   pitch=0.4;   break;
                                    case(21):   pitch=0.5;   break;
                                    case(22):   pitch=0.7;   break;
                                    case(23):   pitch=0.75;  break;
                                    case(24):   pitch=0.8;   break;
                                    case(25):   pitch=1.0;   break;
                                    case(26):   pitch=1.25;  break;
                                    case(27):   pitch=1.5;   break;
                                    case(28):   pitch=1.75;  break;
                                    case(29):   pitch=2.0;   break;
                                    case(30):   pitch=2.5;   break;
                                    case(31):   pitch=3.0;   break;
                                    case(32):   pitch=3.5;   break;
                                    case(33):   pitch=4.0;   break;
                                    case(34):   pitch=5.0;   break;
                                    case(35):   pitch=7.0;   break;
                                      }
                                          if(menu<4){
                                          factor=
(motor_steps*pitch)/(lead_screw_pitch*spindle_encoder_resolution);
                                          switch(menu) {
                                              case(1):    lcd.setCursor(0,0);    lcd.print("Turning        ");
lcd.setCursor(0,1);    lcd.print("Normal    ");    break;
                                              case(2):    lcd.setCursor(0,0);    lcd.print("Turning        ");
lcd.setCursor(0,1);    lcd.print("Fine      ");    break;
                                              case(3):    lcd.setCursor(0,0);    lcd.print("Turning        ");
lcd.setCursor(0,1);    lcd.print("Coarse    ");    break;

                                                    }}
                                          else
                                           {
                                           if(menu<20)
                                           {
                                             depth=pitch_factor*25.4/tpi;                    //the depth of cut in mm on the
compound slide I need for each thread pitch.  I use this during operation rather than looking it up each
time
```

```
                                        factor=
motor_steps*25.4/(tpi*lead_screw_pitch*spindle_encoder_resolution);          //the imperial factor
needed to account for details of lead screw pitch, stepper motor #pulses/rev and encoder #pulses/rev
                                        lcd.setCursor(0,0);  lcd.print("Imperial ");  lcd.print(tpi);      lcd.print(" tpi
");
                                        lcd.setCursor(0,1);  lcd.print("depth="); lcd.print(depth);     lcd.print("
mm");
                                      }
                                      else
                                      {
                                        depth=pitch_factor*pitch;                        //the depth of cut in mm on the
compound slide

factor=pitch*motor_steps/(lead_screw_pitch*spindle_encoder_resolution);        //the metric factor
needed to account for details of lead screw pitch, stepper motor #pulses/rev and encoder #pulses/rev
                                        lcd.setCursor(0,0);     lcd.print("Metric ");  lcd.print(pitch);     lcd.print("
mm");
                                        lcd.setCursor(0,1);     lcd.print("depth=");        lcd.print(depth);
lcd.print(" mm");
                                      }
                                      }

              }
            delivered_stepper_pulses=0;
            input_counter=0;
        }

              encoder0PinALast = n;

 }


  void loop()
  {
  thread_parameters();
  }
```