Arduino Digital Gear Hobbing Code

Jan 24, 2021

Introduction

This document describes the code to drive an Arduino microcontroller for a digital gear hobbing application as outlined in my article in Digital Machinist (Winter 2020 issue, Volume 15, No. 4. “Digital Control for a Gear Hobbing Machine” is found on page 4). This is an example of a more general system which replaces gears connect one drive shaft to another with a defined gear ratio. An example would be a lead screw for a lathe or a gear hobbing machine. In these cases, changing the gear ratio, requires changing gears either manually or with a gear box. This can be expensive (gear box) or tedious (manual swapping of gears). Instead, this new approach connects an encoder to the source shaft (ie. hob spindle) to drive a stepper motor connected to the driven gear blank shaft. The encoder measures the location of the source shaft and sends pulses to the motor to properly position the drive shaft in accordance with the motion of the hob spindle position. To change the gear ratio requires only a change in parameters used in the software to connect the two.

As the controller is necessarily a digital device, the relation between input and output rotation is an approximation of the ideal case which would be achieved with a mechanical gear box. While gears smoothly deliver the torque to the drive shaft, a digital approach will deliver the motion to the output shaft in small discrete steps. Accordingly, if the input shaft was rotating very slowly, one might observe individual steps in the drive shaft rotation. However, with the correct choice of stepper motor parameters, these discrete steps can be reduced to negligible proportions

In order to measure the rotation position of the gear cutter (the “hob”) we must use some form of encoder which is linked to it. Like stepper motors, all encoders are digital devices and measure the source shaft angle in discrete increments. These can have a resolution ranging from 50 to 10,000 angular increments per revolution. In addition, these encoders can measure the rotation direction using quadrature detection[1].

[1] http://www.creative-robotics.com/quadrature-intro

A Digital Controller for Gear Hobbing

There are two elements to the gear hobbing application:

1) a “spindle” which holds the gear blank and
2) a gear hob which rotates to cut the gear.

The hob rotates faster than the spindle. For example, if we are making a gear with 10 teeth, then the hob rotates 10 times for ever rotation of the work spindle. This is ratio of speeds between the hob and the work spindle which is normally achieved with a gear set. In this application, we will replace all these gear with two things: an encoder connected to the gear hob to measure its position which delivers pulses to a controller that delivers pulses to a stepper motor to drive the work spindle.

I will assume that the encoder has a resolution of $N_{encoder}$ pulses per revolution. Gear hobbing machines typically use a worm gear to rotate the spindle. We are going to attach the stepper motor to that worm gear through some linkage. That means that stepper motor will require N_{motor} pulses to rotate the spindle once. For example, a hobbing machine might have a worm gear reduction factor of 10 and there is a 2:1 pulley connecting a 200 step/revolution motor to the worm gear. In this case, $N_{motor} = 200 \cdot 2 \cdot 10 = 4000$. That means it will take 4000 stepper pulses to rotate the spindle once. Now, consider that we want to make a gear with 10 teeth. That means that the hob needs to rotate 10 times for every spindle rotation. Thus each time the hob rotates once, the work spindle will have rotated through 1/10th of a revolution. That means that the number of steps for each hob revolution will be 4000/10 or 400 pulses. So, if we have an encoder with 1024 steps/revolution, we need to generate one spindle pulse for every (1024/400)= 2.56 encoder pulses. That means we will count 2.56 encoder pulses before sending out a stepper pulse. This is the “pulse reduction ratio” for this hypothetical hobbing machine.

More generally, the pulse reduction ratio *R* is given by:

$$R = \frac{G \cdot N_{encoder}}{N_{motor}} \qquad \text{……………………………………………………………….2}$$

Where:

R = is the pulse reduction ratio,
N_{motor} = the number of stepper pulses needed to rotate the spindle once,
$N_{encdoer}$ = the encoder resolution on the hob shaft,
G = the desired number of teeth for our gear.

Calculating the Stepper Motor pulses

From the preceding sections we can see that the pulse reduction ratios can generate arbitrary numbers. We saw that the pulse reduction ratio to cut a 10 tooth gear was 2.56. In general, the values of R will be some arbitrary decimal number. In order to deliver a reduction ratio which is arbitrary number, we use a simple algorithm which is the heart of the controller software.

Let's assume we need a pulse reduction factor R and we count pulses coming from the encoder. At any time, the number of pulses which has come from the encoder will be N_e. That means at any time, the desired position of the work spindle shaft $P_{desired}$ would be:

$$P_{desired} = \frac{N_e}{R} \quad \text{where Ne=1,2,3…} \quad \text{……………………………………….3}$$

Now, let us keep track of the number of pulses delivered to the stepper motor which we will call $N_{delivered}$. For each new value of N_e we calculate $P_{desired}$. If $P_{desired}$ becomes greater than $N_{delivered}$, we deliver a single pulse to the motor. In other words:

If ($P_{desired} > N_{delivered}$)… then increment $N_{delivered}$ by one ………………………………….4

However, if ($P_{desired} < N_{delivered}$)… do nothing

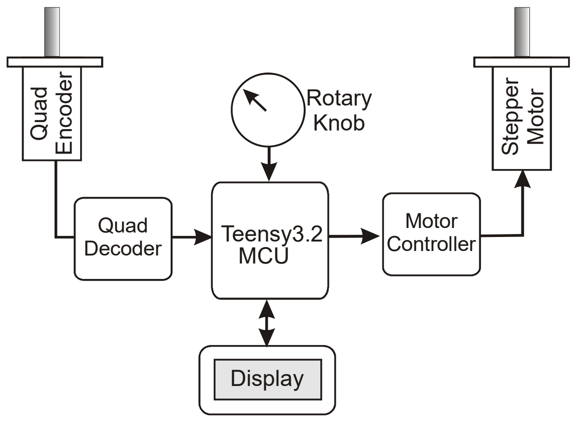
Let's take an example to see how this works as shown in the next table. The first column indicates the measured number of encoder pulses N_e starting from 1 and incrementing by one for each row. The next column shows the desired pulse reduction ratio. Let's make it some arbitrary value, say ***R=3.142***. Next, we calculate the desired number of stepper pulses $P_{desired}$ as defined by equation 3. Next, we see the column $N_{delivered}$ as defined by equation 4. We can see that for the first 3 encoder pulses, the desired position is less than 1. But we can't drive a stepper motor with less than 1 pulse. So $N_{delivered}$ remains at 0 for these first 3 pulses. Then after the 4th encoder pulse we see that $P_{desired}$ got a bit bigger than one. At that point $N_{delivered}$ gets incremented from 0 to 1 and we deliver a single pulse to the stepper motor. $N_{delivered}$ stays at 1 for the next 2 encoder pulses until N_e = 7 where the desired position is 2.228. This triggers the delivery of another pulse to the stepper motor. At this point we have now delivered 2 pulses. This process continues for each new encoder pulse. I show a few more pulses up to N_e =10 where a third pulse is delivered. The important point to note is that the actual stepper position is always within 1 step of the desired position. This means that the ***error NEVER exceeds a single motor step and is often less***.

N_e	R	$P_{desired}$	$N_{delivered}$
1	3.142	0.318	0
2	3.142	0.637	0
3	3.142	0.955	0
4	3.142	1.273	1
5	3.142	1.592	1
6	3.142	1.910	1
7	3.142	2.228	2
8	3.142	2.546	2
9	3.142	2.865	2
10	3.142	3.183	3

Let's put this in perspective by considering our hypothetical gear hobbing machine. Recall that this required 4000 pulses to rotate the work spindle once. Now the algorithm will have a maximum error of 1 step. That means that the error will be 1/4000 of a rotation or 0.09 degrees. If we have a gear of radius 1" that would correspond to a positional error of 0.00025" which will be perfectly adequate for home shop use.

The Way the Controller Electronics Works

At this point we see that the controller works by counting pulses in a special way to allow arbitrary gear ratios. How is a circuit put together to achieve this? It is basically a very simple circuit with very few components as shown here. We can see the quadrature encoder on the left which is connected to the hob. It sends signals to a quadrature decoder which generates pulses corresponding to stepper pulses and direction. This data is input to a digital channel of the Teensy3.2



MCU. This is the heart of the system and does the calculation of pulses needed for the appropriate task at hand. It does these calculations based on data input via the "Rotary Knob" which is a rotary encoder that allows the user to select the desired gear parameters. These data are loaded in the MCU. To keep track of what we are doing there is a "Display" which shows what choices we have made. Finally, on the right, we see signals coming from the Teensy to a "Motor Controller" which is directly connected to the stepper motor.

The Controller Circuit

The schematic is shown on the next page. This is a fairly simple circuit which accepts input levels from a quadrature optical encoder and delivers pulses to two open collector transistors (T1, T2) to drive an external stepper motor controller. The heart of the system is the MCU which is a Teensy3.2[2]. The optical encoder[3] for the hob was a 1024 quadrature encoder. Converting the quadrature signals from the optical encoder into step and direction pulses is done with a LS7184[4] chip which is a detected quadrature detector chip. While this could be done in software, I found that using the LS7184 chip avoided the software overhead and programming complications. It is a very reliable method for quadrature detection. I have not found a good source of this chip in the standard electronics supply houses and had to buy it directly from the manufacture (USDigital).

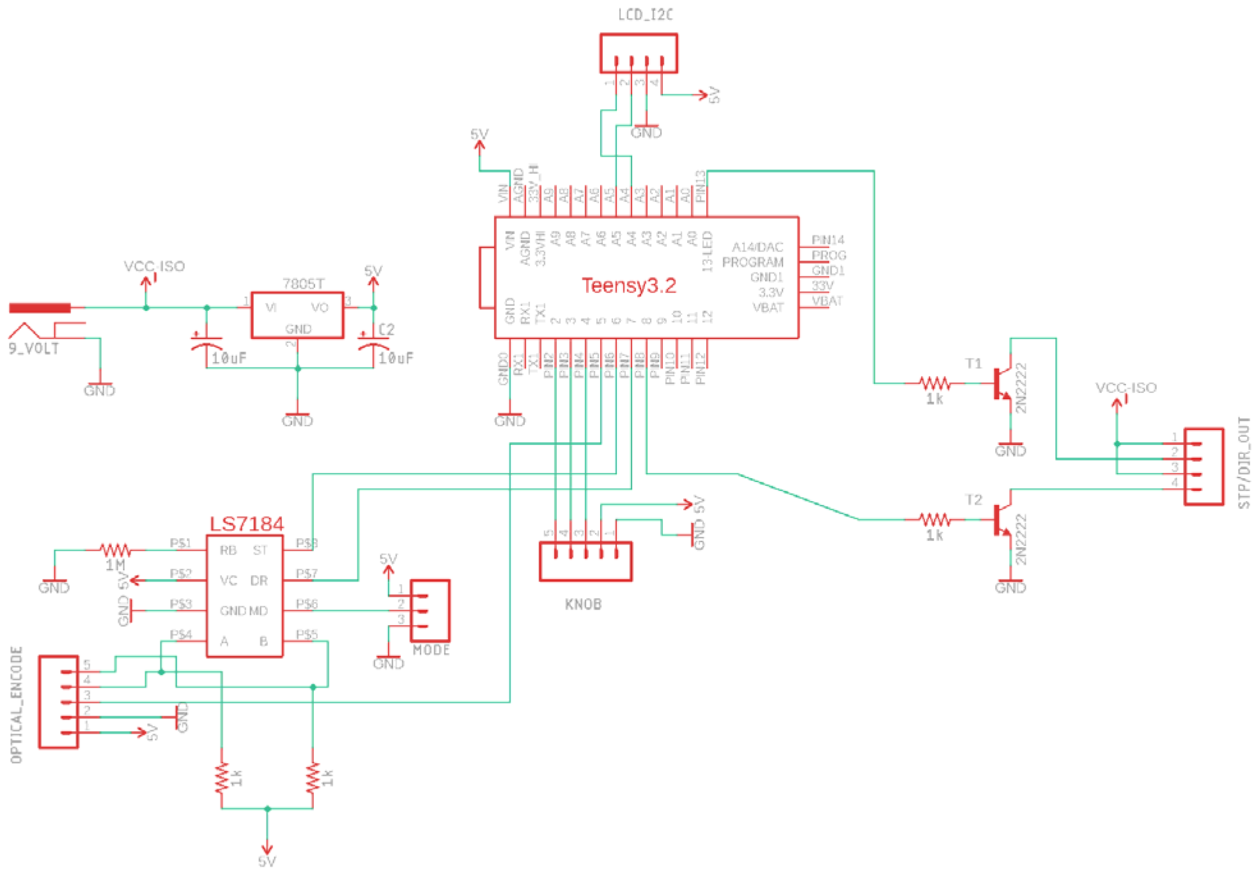
A 4x20 character LCDisplay[5] is connected to the MCU to help in data input while a standard 20 position rotary[6] encoder is used to select data from programmed menus. The LCD is connected via a standard I2C interface while the rotary encoder uses pins 2 and 3 of the MCU in an interrupt routine to detect knob rotation. I used an "I2C backpack[7]" to drive the LCD. A push button on the encoder is detected through pin 4 of the MCU.

We can see the Teensy3.2 in the middle of the schematic. The output from the optical encoder channels (A & B) are connected to pins 4&5 of the LS7184 chip. This chip is the circuit that decodes the

[2] https://www.pjrc.com/teensy/teensy31.html
[3] https://www.sparkfun.com/products/11102
[4] https://www.usdigital.com/products/accessories/interfaces/LS7184N-S
[5] https://www.adafruit.com/product/198
[6] https://www.digikey.ca/en/products/detail/bourns-inc/PEC11R-4220F-S0024/4499660?utm_adgroup=Encoders&utm_source=google&utm_medium=cpc&utm_campaign=Shopping_Product_Sensors%2C%20Transducers_NEW&utm_term=&productid=4499660&gclid=CjwKCAiA9bmABhBbEiwASb35VwkKMH4VhuJ4D-bepWEkvrJOdyCOnRSq83fmZvVoY8g-djYXPsqs8hoCYrQQAvD_BwE
[7] https://www.adafruit.com/product/292

quadrature signals into two separate signals. The first are “stepper pulses” and the second is a level which dictates which direction the motor is to turn. The step output pin from the LS7184 (ST) goes to pin 6 of the Teensy3.2 while the direction level from the LS7184 goes to Teensy pin 7. The Teensy then calculates the correct number of ***stepper pulses*** that are needed to achieve the desired result. These pulses are presented on Teensy pin #13 to the base of an open collector 2n222 transistor (T2). The ***direction level*** for the stepper comes from Teensy pin 8 and drives the base of a second open collector 2n222 transistor (T2). The outputs from the open-collector transistors are used to drive a separate stepper motor controller. Note that there is no current limiting resistor on T1 and T2 as most standard stepper motor drivers have opto-isolated inputs in the form of current limited LED’s. However, if your controller requires current limitation, insert a resistor for that purpose.

Two 10uF capacitors are used to smooth the input and output from the 5 volt regulator. The rotary encoder knob for data input is connected to the Teensy on pins 2 and 3 while the push button is connected to Teensy pin 4. The LCD display is connected through an I2C port with the SDA (DATA) channel connected to Teensy pin A4 and the SCL (CLK) channel connected to Teensy pin A5.



In the standard configuration of the LS7184, it will operate the encoder resolution by a factor 4 without any added jumpers. In this case, using an encoder that delivers 1024 pulses/revolution, the output of the LS7184 will deliver 4096 pulses (4 x 1024) from the input quadrature stream. However, the LS7184 can be adjusted by shorting pins on the "Mode" terminal by shorting pin 2 (or MD pin LS7184 pin 6) to Ground (pin3) or 5V (pin1). Shorting pin 2 to ground will deliver an output resolution equal the encoder. Shorting pin2 will increase the encoder resolution by a factor of 2. Leaving pin 2 floating (as assumed in the code) will increase the resolution by a factor of 4. This allows for varying encoder resolutions to fit the need of different hobbing machines.

Editing the Hobb Code

Depending on the details of different hobbing machines, the key parameters from this code won't function properly. For example, the hobbing encoder may use a different number of pulses per revolution, have a different worm gear ratio or use an alternative stepper motor linkage. Finally, there may be different hob pitches available. These would need to be adjusted to match these requirements.

Accounting for a Different Hobbing Machine.

This code was designed such that the hobbing tool rotated exactly 20 times for each rotation of the spindle. The code needs to know this. Looking at the code we find a parameter called

int spindle_steps_per_rev=16000;   // # motor steps needed to drive the spindle one revolution.

This states that the parameter *spindle_steps_per_rev* has a value of 16000. Now why is the value 16000? Recall that in my application we were using a stepper motor to drive the spindle. Most stepper motors require 200 pulses to rotate once. However, we are using a controller which does micro-stepping and is capable of increasing the number of steps/revolution. In our case, we set it so that it requires 4 times as many steps or 800 steps/revolution to rotate once. Now, the stepper motor is connected to a worm gear of the spindle drive with a gear reduction of 10 through a 2:1 reduction timing belt. This means that the total number of steps required to rotate the spindle once is 800 x 2 x 10 = 16,000. This is why the value *spindle_steps_per_rev* is set to 16000 in this line.

Let’s consider the case where the hobbing machine had a different size worm gear. Let’s say that it was a worm gear with a gear reduction of only 5. In that case the value of *spindle_steps_per_rev* would be 8000 (800 x 2 x 5). So, it that case we would edit this line of code to read:

int spindle_steps_per_rev=8000;   // # motor steps needed to drive the spindle one revolution.

So, depending on the hobbing machine details, this parameter might need to be altered.

hob_encoder_resolution.

What if we had a different encoder resolution? We need to take that into account. Looking at the code we find a line of code which reads:

int hob_encoder_resolution=4096;   // number of pulses per revolution of the spindle encoder after the LS7184 demodulation

In the current set up we are using a YUMO E6B2-CWZ-3E encoder which has 1024 quadrature steps per shaft revolution. This is being delivered to the LS7184N quadrature detector which defaults to deliver 4 pulses for each of the 1024 quadrature pulses. This means it delivers 1024 x 4 = 4096 pulses/shaft revolution. Now, what would we do if instead we used a more standard encoder delivering 200 pulses per revolution? That means that the proper value of the number of hob steps per revolution would be 200 x 4 =800. We would then need to edit this line of code to read:

int hob_encoder_resolution=800;   // number of pulses per revolution of the spindle encoder after the LS7184 demodulation

Hobb Pitch Values:

Let’s consider the case, where we have more hobbs or they are of different value. Looking through the code, we find a section which reads:

```
if(mode_select==2)                                        //mode_select==2 for programming the hobb pitch to calculate the gear blank diameter
      {
        if(encoderPos>10){encoderPos=10;}
        if(encoderPos<0){encoderPos=0;}

      switch(encoderPos) {
                              case(1):    DP=32.0;    break;    //set gear hobb gear pitch for common imperial and metric gears,
                              case(2):    DP=36.0;    break;
                              case(3):    DP=48.0;    break;
                              case(4):    DP=60.0;    break;
                              case(5):    DP=60.8;    break;
                              case(6):    DP=64.0;    break;
```

```
                    case(7):    DP=80.0;    break;
                    case(8):    DP=105.0;    break;
                    case(9):    DP=126.0;    break;
                    case(10):   mod=1.0;   break;

            }
              if(encoderPos<10)                  //calculates gear blank and cut depth for imperial gears
                {
              gear_diameter=(gear_tooth_number+2)/(1.0*DP);
              cut_depth=2.25/(1.0*DP);
              lcd.setCursor(0,1); lcd.print("Dia Pitch= ");lcd.print(DP);lcd.print(" ");    // print out the hobb data for imperial
              lcd.setCursor(0,2); lcd.print("Blank dia=");
                lcd.print(gear_diameter,3);lcd.print(" inch");   // print out the diameter of the blank to be machined (inches)
              lcd.setCursor(0,3); lcd.print("Cut depth=");
                lcd.print(cut_depth,3);lcd.print(" inch");      // print out the total depth of cut needed (inches)
               }
              else
              {
               gear_diameter=(gear_tooth_number+2)*mod;      //calculates gear blank and cut depth for metric gears
               cut_depth=2.25*mod;
               lcd.setCursor(0,1); lcd.print("Module= ");lcd.print(mod);lcd.print("      ");          // print out the hobb data for imperial
               lcd.setCursor(0,2); lcd.print("Blank dia=");
                 lcd.print(gear_diameter/25.4,3);lcd.print(" inch");  // print out the diameter of the blank to be machined (mm)
              lcd.setCursor(0,3); lcd.print("Cut depth=");
                 lcd.print(cut_depth/25.4,3);lcd.print(" inch");     // print out the total depth of cut needed (mm)
              }
```

This section chooses the various hobbs which were available for the machine on which this code was written and then calculates the size of the hobb size and blank diameter. We see that there are a total of 10 different choices. There are 9 hobbs with diametral pitches of: 32, 36, 48, 60, 60.8, 64, 80, 105 and 126. There is also one metric hobb with a modulus of 1.0. It goes on to print out these values on the LCD display in such as way that it reports imperial and metric hobbs appropriately.

Now let's assume we also had a two more metric hobbs; say module 0.5 and 1.5. Let's further assume that we don't have the 60.8 hobb. How would we modify the code to take this into account?

Here is the required code:

```
if(mode_select==2)                                  //mode_select==2 for programming the hobb pitch to calculate the gear blank diameter
    {
      if(encoderPos>11){encoderPos=11;}
      if(encoderPos<0){encoderPos=0;}

    switch(encoderPos) {
                    case(1):    DP=32.0;    break;    //set gear hobb gear pitch for common imperial and metric gears,
                    case(2):    DP=36.0;    break;
                    case(3):    DP=48.0;    break;
                    case(4):    DP=60.0;    break;
                    case(5):    DP=64.0;    break;
                    case(6):    DP=80.0;    break;
                    case(7):    DP=105.0;   break;
                    case(8):    DP=126.0;   break;
   case(9):     mod=0.5;    break;
```

```
            case(10):   mod=1.0;    break;
            case(11):   mod=1.5;    break;
    }
       if(encoderPos<9)                       //calculates gear blank and cut depth for imperial gears
          {
       gear_diameter=(gear_tooth_number+2)/(1.0*DP);
       cut_depth=2.25/(1.0*DP);
       lcd.setCursor(0,1); lcd.print("Dia Pitch= ");lcd.print(DP);lcd.print(" ");    // print out the hobb pitch for imperial
       lcd.setCursor(0,2); lcd.print("Blank dia=");
          lcd.print(gear_diameter,3);lcd.print(" inch");                              // print blank diameter to be machined (inches)
       lcd.setCursor(0,3); lcd.print("Cut depth=");
          lcd.print(cut_depth,3);lcd.print(" inch");                         // print out the total depth of cut needed (inches)
        }
          else
       {
         gear_diameter=(gear_tooth_number+2)*mod;               //calculates gear blank and cut depth for metric gears
         cut_depth=2.25*mod;
         lcd.setCursor(0,1); lcd.print("Module= ");lcd.print(mod);lcd.print("       ");     // print out the hobb data for imperial
         lcd.setCursor(0,2); lcd.print("Blank dia=");
           lcd.print(gear_diameter/25.4,3);lcd.print(" inch");        // print blank diameter to be machined (inches)
        lcd.setCursor(0,3); lcd.print("Cut depth=");
           lcd.print(cut_depth/25.4,3);lcd.print(" inch");                              // print out the total depth of cut needed (inches)
       }
```

Looking at this we see that case(6) was changed to 80.0 from 60.8 and case(9) was changed from 120 to 0.5. We also added a case(11) which added *case(11): mod=1.5 break;* line. Looking carefully at the code, we realize that there are now 11 hobbs instead of 10. So the lines:

```
if(encoderPos>10){encoderPos=10;}
```

was changed to :

```
if(encoderPos>11){encoderPos=11;}
```

and the line:

```
if(encoderPos<10)                //calculates gear blank and cut depth for imperial gears
```

was changed to :

```
if(encoderPos<9)                //calculates gear blank and cut depth for imperial gears
```

This was done to reflect the additional hobs and the change between the number of imperial and metric hobs. That should be all the changes needed to incorporate new hob parameters.