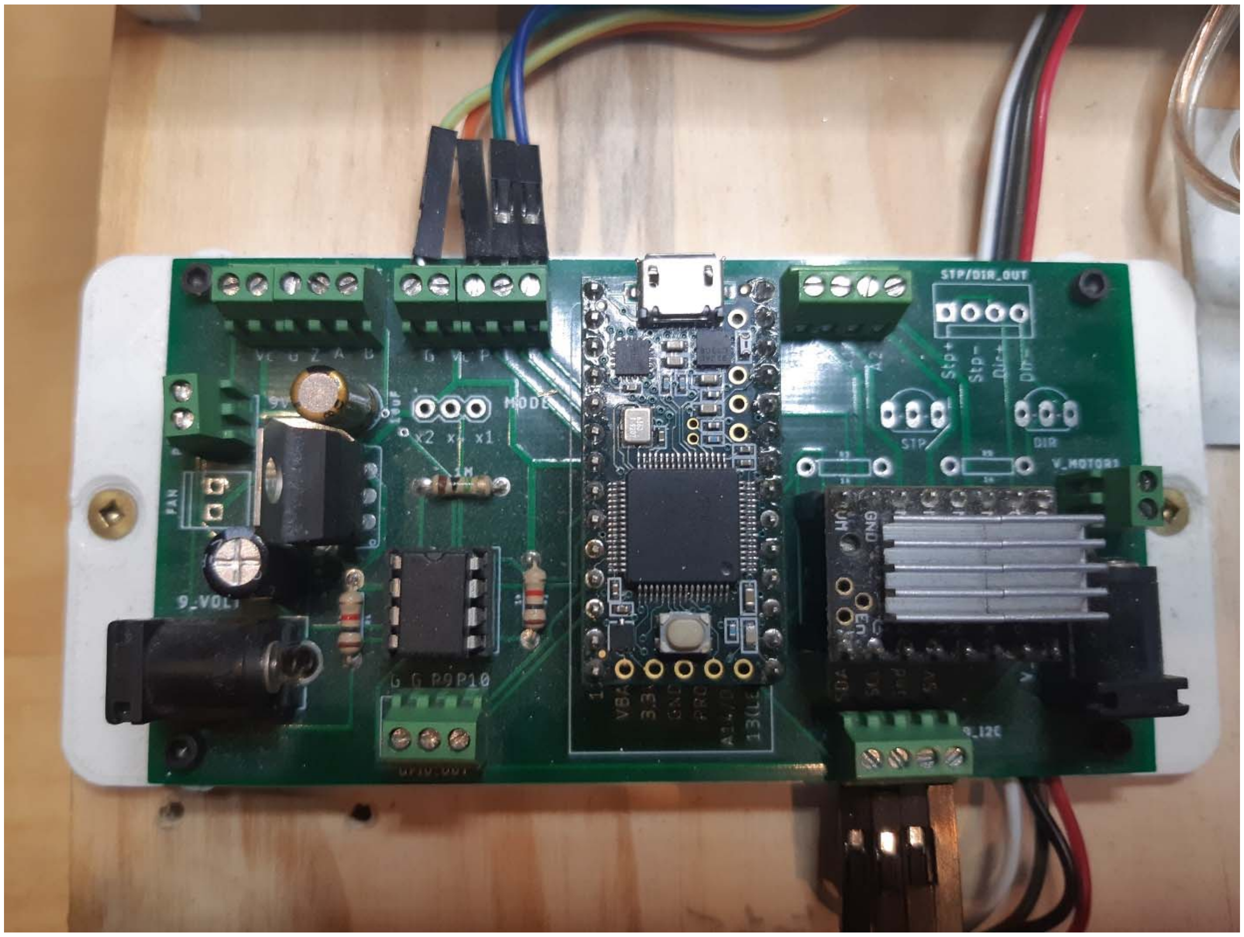
The Electronic Gear Box: A Manual



Donald Plewes, October 2019
Toronto, Ontario, Canada
don.plewes@gmail.com

Contents

Introduction

This document outlines the details of the design of an Electronic Gear Box (EGB). I will describe the underlying principles, the circuit design, the software and the method of integration with various potential applications. Applications include: an electronic lead screw for a lathe, cutting gear with a gear hobbing machine, cutting gears with a milling machine, digital indexing among other applications. The system uses relatively few components and is based on a Teensy 3.2 microcontroller unit (MCU) as its core component.

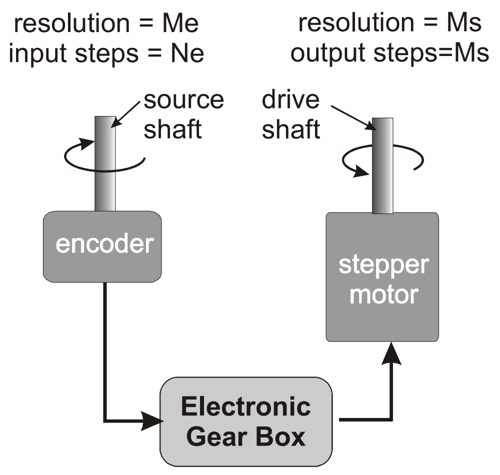
The technical details of implementing this device to various applications will require some basic arithmetic which is needed to derive numerical constants to ensure accurate operation. Furthermore, familiarity with soldering, simple wiring and electrical safety is required. Being able to read a schematic is helpful, although not strictly necessary. It is assumed that user will have loaded the Arduino programming platform[1] on a PC or Mac computer and be familiar with its use. For those unfamiliar with the Arduino platform, I would suggest reading a very simple introduction from the Arduino website[2]. In addition, some standard USB cables are needed to communicate with the MCU. Finally, you will need the circuit board which will need to be populated with components and mounted in an enclosure. It will need to be connected to the various system components with appropriate connections to a motor controller or a stepper motor, an optical encoder as well as input power.

For someone with an understanding of the Arduino programming language, this board can be used as a platform for which one can write a range of programmes for differing applications. For example, in addition to serving as a gear box, a user could easily write software to use the EGB as an indexing system that would allow precise positioning of an indexing head in a milling machine or to serve as a tachometer. However, as mentioned above the most likely application will be for an electronic lead screw for a lathe to perform single point thread cutting.

In the text, I have made recommendations of where I sourced my components. For those in another location, these sources may not be appropriate. However, I have maintained these references here, so that one can look at the components I used to help find an alternative source.

[1] https://www.arduino.cc/en/main/software
[2] https://www.arduino.cc/en/Guide/HomePage

Basic Principles



This system can be used in a situation where there are gears which connect one drive shaft to another with a defined gear ratio. The gear train driving the lead screw on a lathe comes to mind. To change 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, the DGB connects an encoder to the source shaft (lathe spindle) and a stepper motor to the driven shaft (lead screw). 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 spindle position. To change the gear ratio requires only a change in parameters used in the software to connect the two.

As the EGB 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 physical gear box. While gears smoothly deliver the torque to the drive shaft, a digital approach will deliver the motion to the output shaft in discrete small 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. Standard stepper motors are designed to deliver 200 steps per revolution. However, with appropriate drive electronics, the motor can be "micro-stepped" whereby the number of steps per revolution can greatly increased. It is possible to achieve up to 51,200 steps/revolution for some controllers. However, it is more typical to use step resolutions of 400, 800, 1600 or 3200 steps/revolution. We will take advantage of this feature in this work.

In order to measure the rotation position of the source (e.g. lathe spindle) we must use some form of encoder which must be 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 a technique called quadrature detection[3]. It is the combination of an encoder and a

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

stepper motor which are the input and output devices of the Electronic Gear Box. Let's consider a few practical applications. First, I'll consider the application to a lathe lead screw and then in application to a gear hobbing machine.

Application to Drive a Lathe Lead Screw

First let's assume we want to make an electronic lead screw. For those who have lathes which use change gears or pulley, you might be frustrated with the time required to swap out various components to give you the correct setting for a particular single point threading application. For those with gear boxes, this is less of a challenge. However, even here, the EGB could be of value in that it would allow one to create any thread pitch desired or even left-handed threads by the simple push of a button. I made my electronic lead screw about 6 years ago and it has been working beautifully ever since and saved me a great deal of time when cutting threads. To my mind any manual lathe can be greatly enhanced with the addition of the electronic lead screw.

The basic structure is to link the encoder to the spindle and a stepper motor to the lead screw. The EGB goes between these two and serves to calculate the correct number of pulses to be delivered to the lead screw for each spindle rotation. So, in this case we have two elements to the machine:

1) The spindle driving the chuck which holds the work piece and
2) The lead screw which drives the saddle to cut the thread.

For most single point threading applications, the lead screw rotates at a speed similar to that of the spindle. For example, if we are making a thread with 10 threads per inch and we have a lead screw with a pitch of 0.1 inches, means that the spindle will rotate once for every rotation of the lead screw. If we were cutting a thread of 24 threads/inch, the lead screw would rotate 2.4 times the speed as the spindle. So, how to we calculate the correct stepper motor pulse frequency for single point thread cutting?

Let us consider the problem in general. First, we note that Imperial threads are measured in threads per inch (tpi) as given by the parameter *T* which is commonly an integer (i.e. 12, 24 or 32 threads/inch). Let us assume that the lathe has a lead screw of pitch P_{lathe} (in inches) and we use a stepper motor coupled to the lead screw with some kind of gear reduction resulting in total number of pulses needed to rotate the lead screw once given by N_{motor}. The number of pulses *N*, needed to move a cutting tool a distance of *d* inches is $d \cdot N_{motor} / P_{lathe}$. The distance the cutting tool must move to cut a single turn of thread is *d=1/T*. Thus, the number of pulses which we need to deliver per spindle rotation is $N = \frac{N_{motor}}{T \cdot P_{lathe}}$.

To take an example, my Wabeco D6000 lead screw pitch is 4mm so that P_{lathe} = .1574" and my value of N_{motor} =400. This means that the number of pulses to be delivered to the lead screw per spindle rotation is given by $N = 2541.3/T$. Consider the case of cutting a thread with 24 tpi. That means that we need to deliver 2542.3/24 = 105.887 stepper pulses per spindle rotation. Now, assume, the spindle encoder has a resolution of 1024 pulses/spindle revolution. Thus, the EGB must reduce the number of pulses coming from the spindle by a factor of 1024/105.92 = 9.67.

More generally, this "pulse reduction ratio" is given by:

$$R = \frac{N_{encoder} \cdot T \cdot P_{lathe}}{N_{motor}} \text{...} 1$$

Where: N_{motor} is the number of steps to rotate the lead screw once

N_{motor} is the number of steps coming from the encoder per spindle revolution

T is the desired pitch of the thread to be cut

P_{lathe} is the pitch of the lathe lead screw.

Application to Gear Hobbing

Let's assume that we wanted to apply the Electronic Gear Box to replace the physical gear box in a gear hobbing machine. In this case, the two machine elements are:

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

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 spindle. This is ratio of speeds between the hobb and the spindle which is normally achieved with a gear set. In this application, we will replace all these gear with two things: a stepper motor on the "spindle" which supports the gear blank and an encoder to the shaft driving the gear hob to measure its position.

Once again, I will assume that the encoder will have a resolution of $N_{encoder}$ pulses per revolution. Gear hobbing machines typically have a worm gear to rotate the spindle. We are going to attach the stepper motor to that worm gear through some pulley linkage. That means that the spindle 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 assume a 2:1 pulley connecting the 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 hobbing tool needs to rotate 10 times for every spindle rotation. That means that each time the hob rotates once, the 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 pulses 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 the hobbing application. More generally, the pulse reduction ratio for the hobbing application is given by:

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

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 number of teeth for our gear.

Calculating the Stepper Motor pulses

From the preceding sections we have two equations for the operation of the EGB for two different applications. You will note that the pulse reduction ratios can generate arbitrary numbers. For example, the Wabeco lathe cutting a 24 tpi thread had a pulse reduction ratio of 9.67 while the hobb machine example (10 teeth) had a pulse reduction ratio of 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 EGB software. I will briefly explain it here, but it is not necessary to understand this to use and make the EGB.

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 $N_{encoder}$. That means at any time the desired position of the drive shaft $P_{desired}$ would be:

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

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 $N_{encoder}$, we calculate $P_{desired}$. If $P_{desired}$ becomes greater than $N_{delivered}$, we deliver a single pulse. In other words:

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

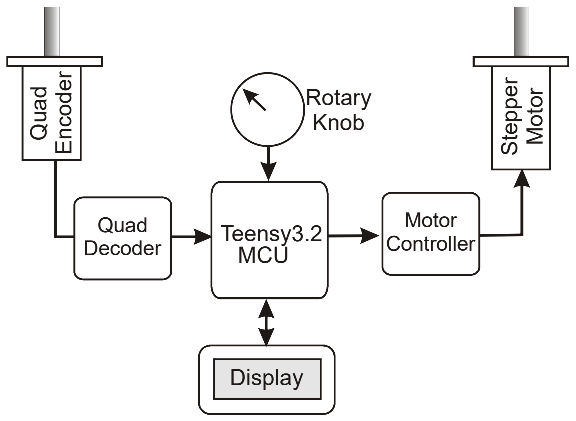
Let's take an example to illustrate how this works as shown in the next table. The first column indicates the measured number of encoder pulses $N_{encoder}$ starting from 1 and incrementing by one for each row. The next column shows the desired pulse reduction ratio. Let's make it some arbitrary value, say ***R=3.142***. Next, we calculate where the desired number of stepper pulses $P_{desired}$ as defined by equation 3. Next, we see the column $N_{delivered}$ which is calculated by taking the integer value of $P_{desired}$ as defined by equation 4. 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 $P_{desired}$ gets a bit bigger than one so $N_{delivered}$ gets incremented to 1 at which point we deliver a pulse to the stepper motor. This continues and we get our next pulse at $N_{encoder}$ = 7 where the desired position is 2.228. At this point we have now delivered 2 pulses. This continues for each new pulse. I show a few more encoder pulses up to $N_{encoder}$ =10. The key thing 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***.

$N_{encoder}$	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. Consider an example where one is using the EGB to drive a lead screw on a lathe. My lathe has a lead screw with a pitch of 4mm and my stepper motor (200 steps/revolution) is connected to the lead screw with a 2:1 timing belt reduction resulting in 400 steps per lead screw rotation. Thus the positional error in the lathe saddle is never greater than 1 step. This corresponds to 4(mm)/400 = 0.01mm = 0.0004". This is the largest error that can occur and typically is less. This error for a lathe application is perfectly adequate for home shop use. As such, this algorithm will work well for lots of applications.

The Way the Electronic Gear Box Works

Earlier, we pointed out that the EGB 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. It sends signals to a quadrature decoder which generates pulses corresponding to stepper pulses and direction. This data is input to digital channels 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 parameters input via the “Rotary Knob” which is a knob that sends pulses to the Teensy3.2. We use these pulses to pick among various menu possibilities for the desired data which are in turn incorporated into the calculations for the Teensy. 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 data coming from the Teensy to a “Motor Controller” which is directly connected to the stepper motor. As will be detailed below, the motor controller can be mounted directly onto the EGB board or it could a separate controller depending on how much torque is required for the stepper motor. That is the basic block diagram of the system. Next, I will provide a bit more detail about the EGB circuit which delivers these functions.

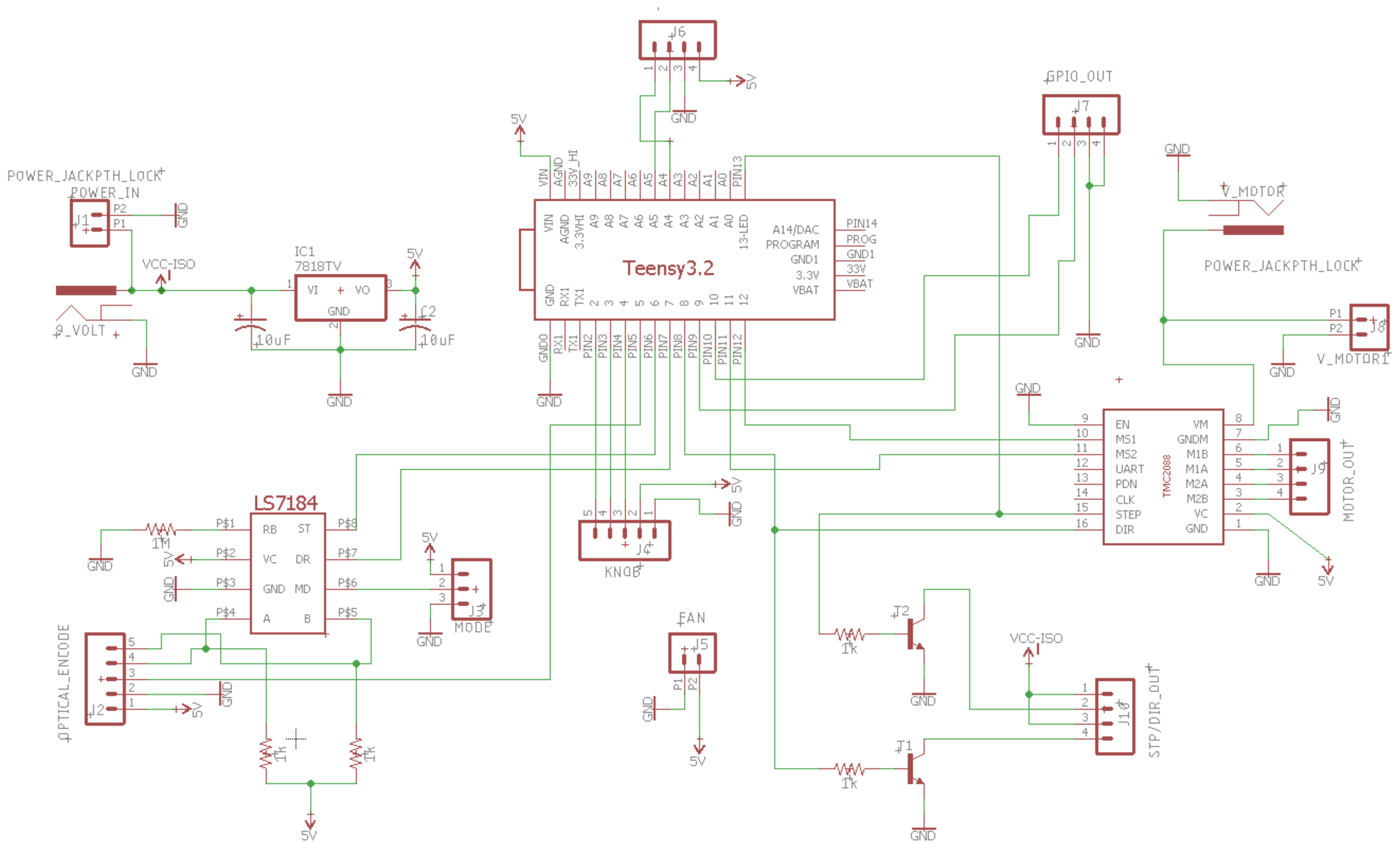
The EGB Circuit

Let’s look at the electronic circuit to see how it works. The schematic is shown on the next page. We can see the Teensy3.2 in the middle of the drawing. DC power to run the circuit is introduced on the left side through the 2.1 mm barrel connector which is connected to the voltage regulator which then generates 5 volts to power the MCU and TMC2088. This DC power comes from a 9 Volt supply “wall-wart” external plug supply. There is a screw terminal (J1-2 pins) which allows wired connections of power to the board which helps in adding an on-off switch. We can see positions for other connectors including: the optical encoder (J2- 5 pins), the mode selection (J3-3 pins), rotary knob (J4-5 pins), a fan (J5-2 pins), an LCD display (J6-4 pins), more GPIO pins (J7-4 pins), the motor power in (J8-2 pins), the motor output pins from the TMC2088 driver (J9-4 pins), and open collector outputs to transfer step/direction pulses to an independent stepper controller (J10-4 pins).

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 quadrature signals into two separate signals. The first are “stepper pulses” and the second is a level which dictates which ”direction” the encoder is turning. There is also an “index” output channel from the quadrature optical encoder, sometimes called “Z”, which is connected directly to Teensy pin 5. This gives a short pulse each full revolution. 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 and goes to pin 15 of the TMC2088 as well as to the base of an open collector transistor (T2). The ***direction level*** for the stepper comes from Teensy pin 8 and goes to pin 16 of the TMC2088 and also to a second open collector transistor for opto-isolation (T1). Pins 15 and 16 of the TMC2088 are the step and direction channels of this controller board. The outputs from the open-collector transistor can be used to drive a separate stepper motor controller. Most standard stepper motor drivers have opto-isolated inputs in the form of current limited LED's which are used to close the open collector transistor to Vcc-ISO.

This circuit has two power supplies: a 9 Volt called Vcc-ISO which runs the electronics and a second supply (V_Motor) which is used to drive only the TMC2088 controller chip. V_Motor can have a voltage ranging from 9-28 volts. So, if we want more power for the stepper using the on-board TMC2088 driver, we simply use this higher voltage power supply.

Two 10uF capacitors are used to smooth the input and output from the 5 volt regulator. GPIO pins 9 & 10 from the Teensy are also accessible as additional GPIO output levels available on connector (J7). 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 via connector J6 through an I2C port with the SDA (DATA) channel connected to Teensy pin A4 and the SCL (CLK) channel connected to Teensy pin A5. The Gnd and 5 volts needed to power the LCD are also provided on J6. A two-pin connector J5, provides 5 volts to power a small fan if desired. If the TMC2088 stepper controller is used, the connection to a bipolar stepper motor is achieved from connector J9 with appropriate connections shown on the board.



The connector labeled J3, shown as “MODE”, sets the output of the LM7184 to give varying levels of encoder resolution. Let’s assume that the encoder has a resolution of 1024 pulses/revolution. Here is a table showing the effect of adding jumpers to this connector.

Pin 2 shorted to pin…	Encoder Steps/Revolution
1 (5V)	1024
3 (Gnd)	2048
Not connected	4096

In the standard configuration which requires no wiring, the LS7184 will increase the encoder resolution by a factor 4 and nothing needs to be added. In this case, with an encoder that delivers 1024 pulses/revolution, the output of the LS7184 will deliver 4096 pulses (4 x 1024) from the input quadrature stream. Alternatively, by shorting pin 2 to either pin 1 or 3, we would get an output resolution of 1024 or 2048 respectively. Using a screw connector for this is not strictly needed but it does allow it to be changed if desired. However, for most applications, the output resolution will be known, so that a little short wire could be soldered directly into the appropriate pin pads.

As mentioned earlier, the board is designed to allow it to run in different modes which are configured through a combination of software and hardware as explained in the next section.

Various Configuration of EGB board.

The EGB board is designed to be flexible in terms of its use. The board can be configured a number of ways depending on the final application. In every case, the board requires the Teensy3.2 to be in place. However various combinations of the LS7184 chip, the optical encoder, rotary knob, LCD display, output transistors (J1 and J2) or TMC2088 stepper driver chip can be added in varying configurations depending on the application. In each case the software running the Teensy3.2 needs to be altered to meet the needs of the application. Next, I will summarize some configurations in which the DGB board can be wired.

Configuration 1

In this configuration, the board requires the LS7184 chip, the TMC2088 driver, the rotary knob and the LCD display. These are added to the board and will permit the board to accept quadrature pulses from the quad encoder and deliver step pulses directly to a stepper motor from the on-board stepper controller. It will permit programming the board to deliver varying behavior depending on input parameter choices. Two power supplies are used for the board in this instance: a 9 volt for the logic and the second (9-28 volts) for the TMC2088 stepper driver.

Configuration 2

In this configuration, the board requires the LS7184, the rotary knob, the LCD display and output transistors (T1 and T2). This will allow it to accept quad pulses from the encoder and deliver pulses to an external opto-isolated stepper motor controller. It will permit programming the board to deliver varying behavior depending on input parameter choices. In this instance, only the 9V power supply is needed.

Configuration 3

In this configuration, the board requires only the TMC2088 chip, the rotary knob and the LCD display. This will allow the board to be used to directly drive the stepper motor and doesn't require any quadrature encoder input. In this appliocation, the software generates pulses to the stepper depending on the application. Two power supplies are needed for the board in this instance: one for the logic and the second for the TMC2088 stepper driver.

Configuration 4

In this configuration, the board requires the rotary knob and the LCD display and output transistors (J1) and (J2). This will allow the board to be used to drive an external stepper motor and doesn't require any quadrature encoder input. Like configuration 3, the software generates pulses to the stepper depending on the application. Only the 9 volt power supply is needed.

Configuration 5

In this configuration, the board requires only the TMC2088 chip. This will allow the board to be used to directly drive the stepper motor without any display or data input. Two power supplies are needed for the board in this instance: one for the logic and the second for the TMC2088 stepper driver. Alternatively, the TMC2088 can be eliminated and only use the output transistors to drive an external stepper controller in which case, the second power supply is not needed.

Configuration 6

In this configuration, the board requires the LS7184, optical encoder and the LCD display. In this application, the board is used to report the rotational or translational position of some device that is controlled by some other source. For example, it could be used as a tachometer, a phase angle measurement or a linear DRO detector and display.

Components needed for Different Applications of the EGB

What are the applications for which these various configurations could be used? Here are a possible list of applications and the corresponding configurations that would be required.

Lathe Electronic Lead Screw :

This could be performed with configurations 1 and 2 for the low torque and high torque applications respectively. Some code for this application is written is shown in APPENDIX II. This particular code assumes we are using an external motor controller. It needs to be edited to suit the lathe mechanics and the stepper motor configuration.

Electronic Hobbing Machine:

This could be performed with configurations 1 (low torque stepper) and 2 (high torque stepper). Some code for this application is written is shown in APPENDIX I. This code assumes we are using an external motor controller. It needs to be edited to suit the hobbing machine mechanics and the stepper motor configuration.

Digital Indexer for a Rotary Table:

This could be performed with configurations 3 (low torque stepper) and 4 (high torque stepper).

Programmable Tachometer, Angle Measurement or DRO:

This could be performed with configuration 5 so that the sensitivity or value of the readout could be altered. In each case the quadrature encoder would be coupled to some mechanical system to report angle or translation. The rotary knob would allow a change in the readout characteristics depending on the application.

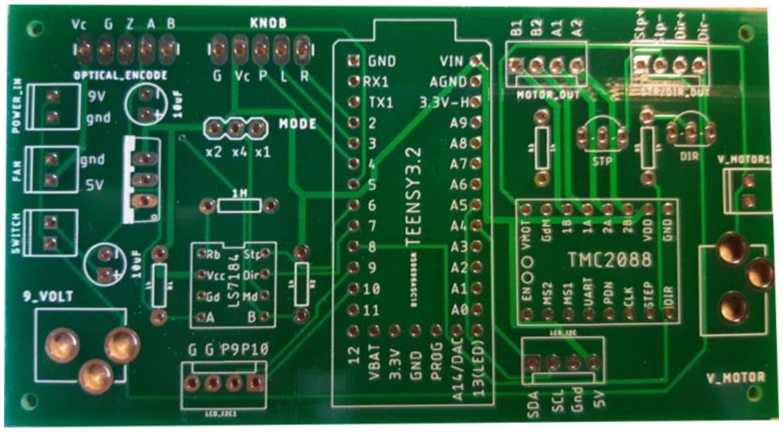
Fixed Tachometer, Angle Measurement or DRO:

This could be performed with configuration 6. In each case, the quadrature encoder would be coupled to some mechanical system to report angle or translation. The operation of the device would be fixed.

Generalized Stepper Motor Driver:

This could be performed with configuration 5. In this case, the EGB would drive a stepper motor for some programmed movement cycle. For example, this could drive a rotary tumbler for cleaning metal parts. The author has built this configuration to clear and deburr brass and steel components with a stepper motor driving the rotary drum. The stepper motor was programmed to accelerate to a specific speed, run for a fixed number of rotations and decelerate to zero speed, with reversing directions.

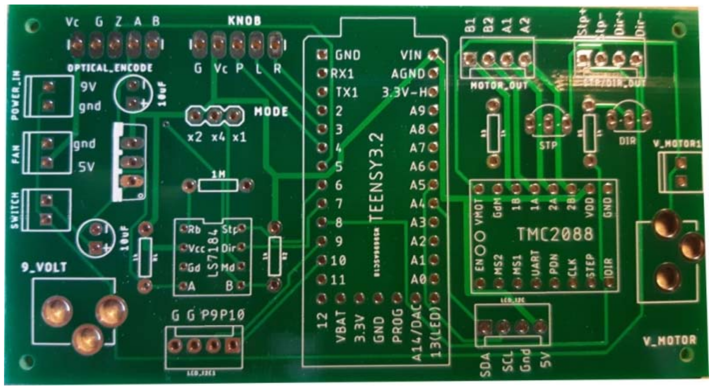
Digital Gear Box Components



In this section, I'll describe the components that are used to run the EGB. It contains only a few critical components which are easily obtained online at various electronics supply houses[4],[5] [6]. Being based in Canada I have references sources in Canada as well as the US. Next, I will describe the main components of the EGB and then follow with a summary of how to put these parts together.

EGB Printed Circuit Board

The EGB is configured around a printed circuit board which I have designed and get manufactured from a Chinese PCB house in small numbers. I have designed the board to match the various components that I will describe below and they only need to be soldered in place. The empty circuit board looks like this. A mechanical drawing of the board from the Gerber files used to make it is shown in Appendix IV.



You can see the locations of specific components, most notably the "Teensy3.2" and "TMC2088" and the "LS7184". These are all integrated circuits mounted on "breakout" boards. These get mounted into place through male headers or an IC socket which are soldered into the board. This means that we can remove the breakout boards and the LS7184 chip if needed and re-insert them easily. In addition, there are a number of items like a 5 Volt regulator, a few electrolytic capacitors, a few resistors (4 of 1kohm, 1 of 1Mohm), 2 NPN transistors and an assortment of connectors (two 2.1mm barrel connectors and various multi-pin screw terminals). These components need to be

[4] https://www.digikey.ca/
[5] https://www.creatroninc.com/
[6] https://www.robotshop.com/

soldered in place, taking care to get their polarities correct. Next, I will describe each set of components in detail.

MCU (Teensy3.2)

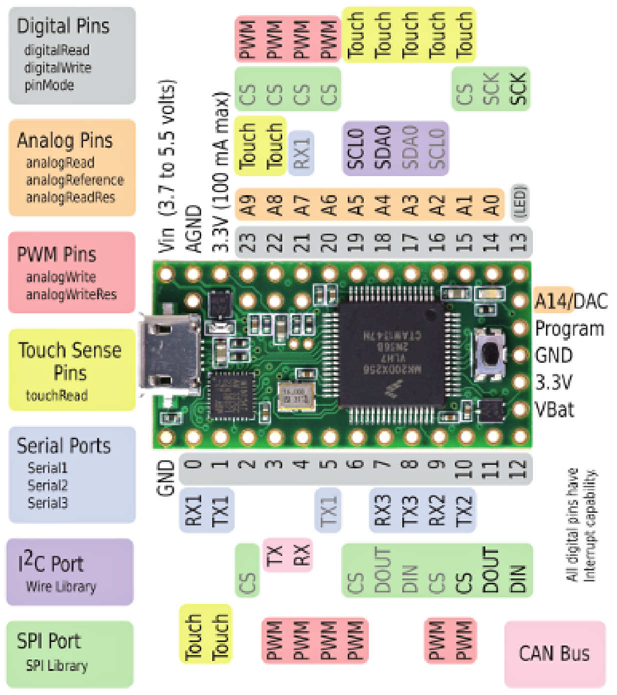
The heart of the EGB is a small microcontroller (MCU) called the Teensy3.2 which is made by PJRC[7]. This is a very powerful, yet inexpensive and physically small MCU. Here is a photograph of the Teensy3.2 and its pin out diagram.

The Teensy measures 38x18mm and has multiple I/O lines. It is run on a Cortex M4 microprocessor[8] which includes a dedicated floating point arithmetic module. There are a total of 34 GPIO lines (24 available via the breadboard) and operates at 3.3 volts. However, one of the advantages of this MCU is that the GPIO lines are tolerant to 5 volt logic levels. The CPU runs at 76 MHz and can be over-clocked to 96 MHz It has 256 kB of flash memory and 64 kB of core memory. All GPIO pins can be used for interrupt routines. It has two sets of I2C serial communication ports and a built in USB connector for programming. It can be powered from the USB connector from a 5 volt supply (or from your computer through the USB connector) and has its own voltage regulation to allow powering from an independent supply ranging from 3.7-5.5 volts. Each GPIO line can deliver a maximum current of 10mA. It is inexpensive and is an amazing little chip which is ideal for this application.

[7] https://www.pjrc.com/

[8] https://www.pjrc.com/teensy/teensy31.html

Quadrature Optical Encoder (YUMO E6B2-CWZ-3E)

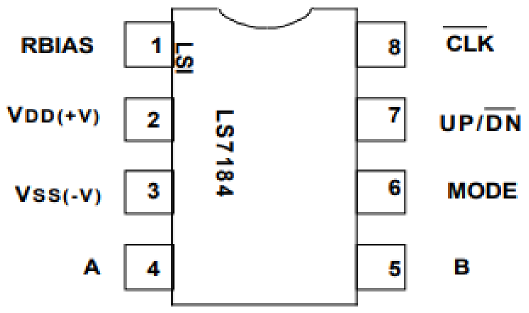


The encoder which I have used with success is a relatively inexpensive optical encoder. It is a quadrature encoder with two signal lines (A and B) as well as an index pulse (Z). It delivers a total of 1024 pulses/revolution. It can be powered by a 5-12 volt DC supply.

These encoders are small, measuring only 40mm diameter, 40 tall with a 6mm shaft and can be mounted with M3 screws. It has a bandwidth capable of supporting encoding shafts speeds up to 6000 RPM. It can be purchased from a number of sources[9] ,[10]. Other optical encoders could be used[11] but I have no experience with these.

Quadrature Decoder Chip (LS7184)

The optical encoder outputs are quadrature signals (A and B) which are used to determine source shaft position and direction of rotation. While these quad signals could be decoded in software, the timing overhead is a bit cumbersome and is more easily achieved with a dedicated quadrature detector. I have found an excellent choice made by US Digital[12]. It is an 8 pin integrated circuit that takes the A and B lines of the encoder as inputs (attached to IC pins 4 and 5 respectively). From this, the chip generates two outputs: a pulse stream reflecting encoder position



(from pin 8 shown as CLK) and a level to indicate which direction the shaft if rotating (from pin 7 shown as UP/DN). A 1 M-ohm resistor which grounds pin1 (RBIAS) is used to set the width of the output

[9] https://www.sparkfun.com/products/11102
[10] https://www.robotshop.com/ca/en/6mm-rotary-encoder-1024-p-r.html
[11] https://www.omc-stepperonline.com/optical-encoder
[12] https://www.usdigital.com/products/accessories/interfaces/LS7184N

pulses to 1 microsecond. This chip has a significant advantage in that it can be used to increase the resolution of the encoder by factors of 1, 2 or 4 by setting pin 6 (mode) to ground, 5 volts or leaving pin 6 floating. It has a very high bandwidth which is very suitable to our problem. This chip is a little hard to get. I couldn't find it at any of the standard electronics source such as Mouser or Digikey and resorted to getting it directly from the manufacture[13]. This chip is essential and I have not found a better alternative.

Low Power Stepper Motor Controller (TMC2088) < 1.5 Amps

I have designed the board to power a small stepper motor directly with an onboard controller. The controller I prefer is made by Trinamic (TMC2088)[14] and is a very efficient solution to drive small stepper motors that pull currents up to 1.5Amps/phase. The chip does efficient microstepping and generates stepper motions that are virtually silent. As such, it uses energy very efficiently. By setting two pins (MS1 and MS2) to various levels, it is possible to get microstepping values of 2, 4, 8 or 16 microsteps/step. It can be obtained from Digikey for $11.00 each. A small heat sink can be added to manage temperature if needed. This board should be used in the case where we want to drive a small stepper motor with low currents. However, many applications would be satisfied with currents of 1.5 amp/phase. However, if a larger stepper motor is needed, a separate controller can be used as explained in the next section.

Separate Stepper Motor Controller for Current > 1.5 Amps

If the application requires a more torque than can be delivered by 1.5 Amps, we don't need to use the TMC2088 chip at all. Instead, I have supplied a set of dedicated outputs to trigger an external optically isolated stepper motor controller. I have found a great source of these controllers which are quite powerful and inexpensive. They are

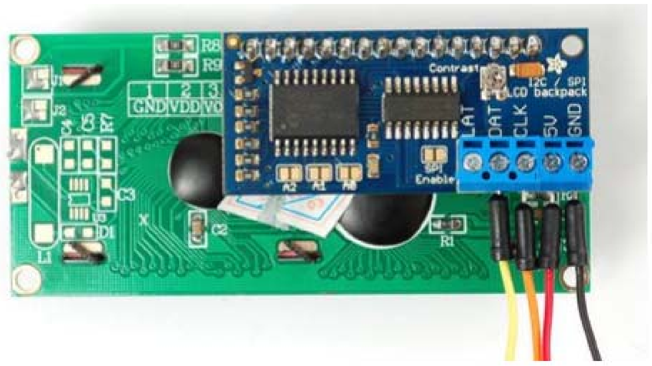
[13] https://www.usdigital.com/products/interfaces/ics/LS7184N
[14] https://www.digikey.ca/product-detail/en/trinamic-motion-control-gmbh/TMC2208-SILENTSTEPSTICK/1460-1201-ND/6873626

inexpensive and obtained online[15] and they look something like this. These often require open collector drivers for the step and direction inputs which I have provided on the EGB board.

LCD monitor

In order to adjust the parameters of the EGB, we need to be able to alter the parameters of the software. To make a selection, a small LCD screen is used to display the parameters. In fact, any I2C display could be used; however, I have found that a simple 4x20 character LCD display offers sufficient space to view the needed data. I have used this monitor which can be purchased online[16]. It needs 5 volts to power the device and simple Arduino libraries can be used to program its use. If you don't get either of these displays, make sure that it is compatible with the I2C backpack and it must run at 5 volts. There are lots than require 3.3 volts. This kind won't work in the PCB which delivers 5 Volts to the PCB.

I2C Backpack



In normal use, the LCD requires 9 interface lines for operation and as such is very wasteful of the GPIO pins on the Teensy board. However, a separate component can be added to make it suitable for I2C communication which requires only 2 GPIO lines. This an "I2C backpack" made by Adafruit which can also be purchased from Digikey[17] or Adafruit[18]. This is soldered to the display with little headers and makes it easier to interface. When the backpack is mounted on the LCD display it looks like this. A

[15] https://www.omc-stepperonline.com/digital-stepper-driver/digital-stepper-driver-rms-current-max-4a-24-60vdc.html
[16] https://www.digikey.ca/product-detail/en/adafruit-industries-llc/198/1528-1503-ND/5774229
[17] https://www.digikey.ca/product-detail/en/adafruit-industries-llc/292/1528-1446-ND/5761214?utm_adgroup=&mkwid=sOO9MMNUE&pcrid=311488457695&pkw=&pmt=&pdv=c&productid=5761214&slid=&gclid=Cj0KCQjw1MXpBRDjARIsAHtdN-3wGisP5TtKbsZMqpSAhPg5Muv-oDOWgQLoP_solIEEdacTKVOTRxMaAlxfEALw_wcB
[18] https://learn.adafruit.com/i2c-spi-lcd-backpack/arduino-i2c-use?gclid=Cj0KCQjw1MXpBRDjARIsAHtdN-1qUh6MRSKi0vLJPBc91Ei6bZb5TNAcdFvsO4As2lVM6mkt-BS3-LQaAsPbEALw_wcB

connector on the backpack provides the four connections needed for an I2C interface which are Gnd, 5V, SDA (DAT in picture) and SCL (CLK in picture) .

Rotary Knob and Push Button

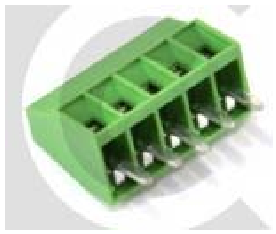
A small rotary encoder is used to allow choices generate from the software to set the operating parameters. I recommend using a rotary encoder. Typically they have 20 positions per rotation and the shaft has a push button. The standard version like this is manufactured by Bourns for about $2.00. With these you need to be careful about their wiring. If it is the Bourns encoder, it is not mounted on a breakout board and there are no labels. In the section on wiring, I'll review the wiring methods for different encoders.

Voltage Regulator (LM7805) and Smoothing Capacitors

A voltage regulator LM7805 is needed for the EGB board. These are polarized and need to be inserted into the board with attention to their polarity. Similarly, the required polarity for the capacitors is indicated on the board and needs to be aligned with markings on the capacitor.

Screw Connectors

The board needs a number of small screw terminals to connect the LCD display, quadrature encoder, motor output, various digital IO pins and rotary encoder to the board. I recommend using these 0.1" pitch screw terminals. They come in various sizes ranging from 2 to 10 terminals in a block. These are soldered into the board and make connections to the board stable and convenient. These can be purchased from Creatron for $3-$5 Cdn each, depending on the number of terminals. We need a maximum of 10 of these as detailed below but probably fewer depending on which configuration of board you choose.



Pin headers - 0.1”

It is helpful to use these little headers as connectors. For example, the fan can be easily plugged in using these. They come in long lengths and can be cut to the number of pins needed. These are also needed for the TMC2088 and Teensy3.2 board to allow insertion into their corresponding headers if your boards don’t already come with them cut to size. You can easily cut them to size with a set of wire cutters.

Transistors

Two NPN transistors are needed. They can be any small NPN switching transistor but either 2N2222 or 2N3904 would be sufficient. These can be purchased for about $0.30 each from Creatron[19].

Power Requirements and Power Switch

The board can be powered with a wall mounted power supply like this[20]. It is plugged into a standard wall outlet and has a 2.1mm barrel plug (centre pin +Ve). There are options for two power supplies on the board. A 9 Volt supply is required to power the logic and MCU (~ 1 Amp). There is also a provision for a second power supply which can have a voltage range from 9-24 volts (~2 Amps) which is used to power the on-board motor controller. If the TMC2088 is not used, this second power supply is not needed.

Adding a switch is helpful to turn the whole unit on-off with a simple SPDT toggle switch like this[21] or something similar. It is wired in series with the power input to the board.

Various Connectors

You will need connectors to get the various signals from the quadrature encoder into the board, for DC power and the motor or the step/direction output out of the unit. These don’t need to handle much current (except for the motor ~1.5 amps). The DC power can be

[19] https://www.creatroninc.com/product/2n3904-npn-bjt-40v-02a/?search_query=npn&results=68
[20] https://www.creatroninc.com/product/9v-2a-switching-power-supply/?search_query=power+supply&results=126
[21] https://www.creatroninc.com/product/spst-toggle-switch-on-off/?search_query=switch&results=306

connected with this panel mounted 2.1mm barrel connector as shown here.

The encoder which has 5 wires, so I would recommend this connector[22]. There is a male and female end. The male end is panel mounted. For the motor or output connection to the opto-isolated lines you need only 4 wires. A similar connector to the 5 pin connector would work[23] which has only 4 pins.

A 2.1mm barrel connector for power is also helpful, though not strictly needed. It looks like this.

Standoff-Headers and IC Sockets

Looking at the EGB board we see the outline for the Teensy3.2 board location with a specific orientation to must match the pin numbers. Likewise, the locations for the LS7184 and TMC2088 chips are shown. It is possible to solder all these board right into the printed circuit board but I would **not** recommend doing this. Rather, having the capacity to add or remove the boards makes the circuit more flexible in terms of its use and troubleshooting. To do this, I would suggest the use headers and an IC socket . Headers (0.1" pitch) which range in size from 2 to 16 pins. The Teeny3.2 requires two rows of 14 pins and the TMC2088 requires two rows of 8 pins.

[22] https://www.creatroninc.com/product/5-pin-circular-connector-set-panel-mount/?search_query=plug&results=163
[23] https://www.creatroninc.com/product/4-pin-circular-connector-set-panel-mount/?search_query=plug&results=163

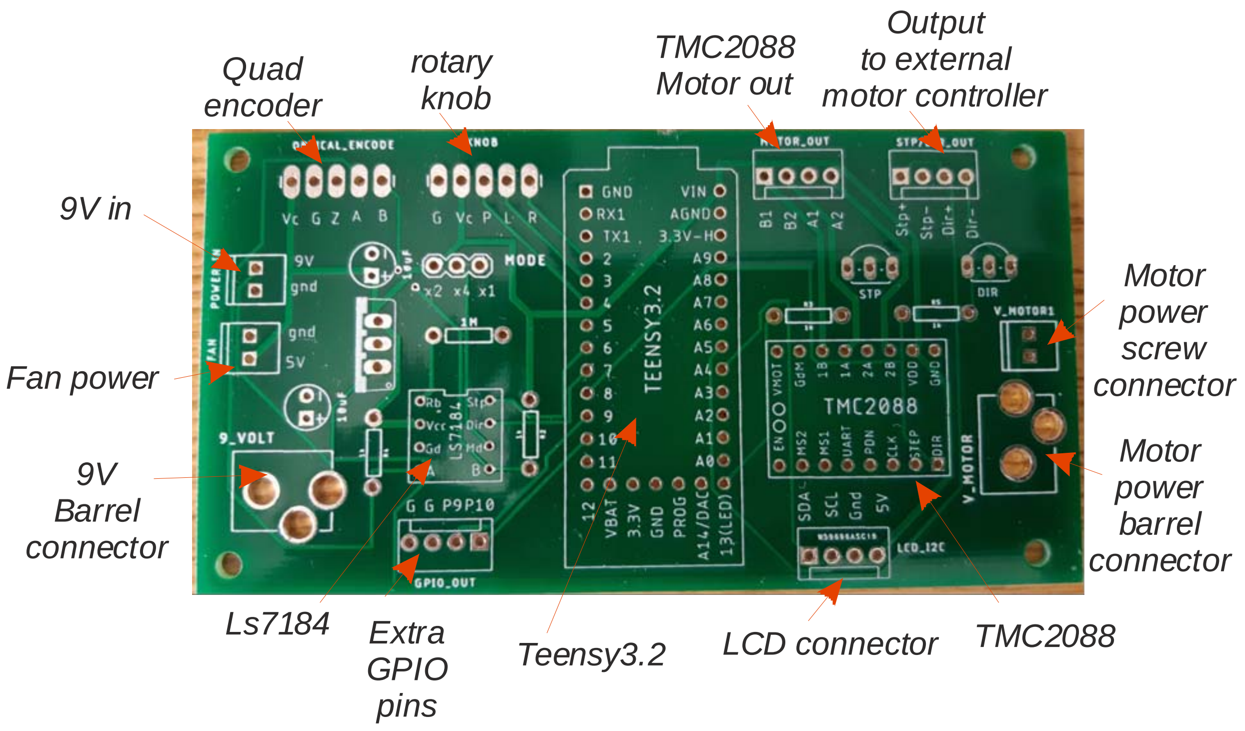
8 pin IC socket

I would recommend using an IC socket to mount the LS7184 quadrature encoder chip onto the board. It is a simple way of mounting and allows for troubleshooting should it be needed. It can be purchased from Creatron[24] for $0.40. It looks like this.

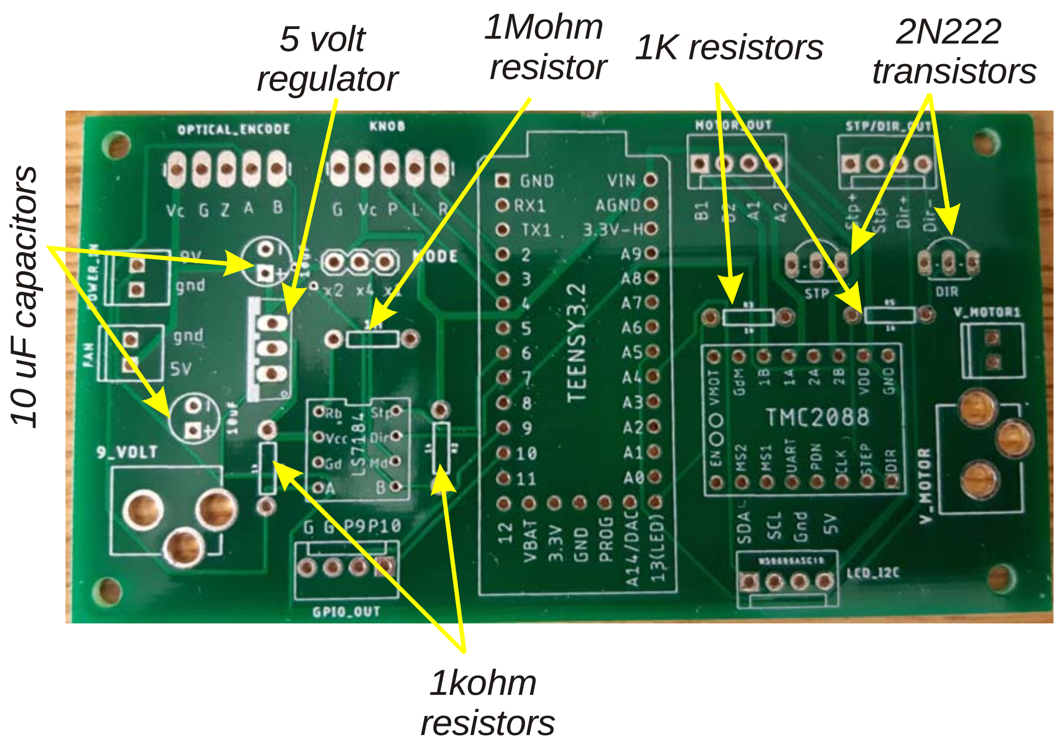
Assembling the Digital Gear Box PCB

The name printed on the printed circuit board is *Digital Gear Box, DB Plewes – 2019 Rev 2.1*. You will need to solder the parts into this board. In order to populate the board, I would suggest mounting all the needed screw terminal connectors, the headers for the Teensy and the TMC2088 as well as the IC socket for the LS7184 chip. Some encoders have open collector configurations as their output. In this case, 1 Kohm resistors are needed to be added to the input of the LS7184 lines to detect the encoder pulses. This is the case for the YUMO E6B2-CWZ-3E quadrature encoder mentioned above. We also need to mount two 1K resistors for the 2N222 transistors as well as a single 1Mohm resistor for the LS7184 timing. Finally, the two 2N2222 transistors needed be mounted in their location. The orientation of the transistor needs to match the outline shown on the board. I have provided provisions for this and their location is shown on the board.

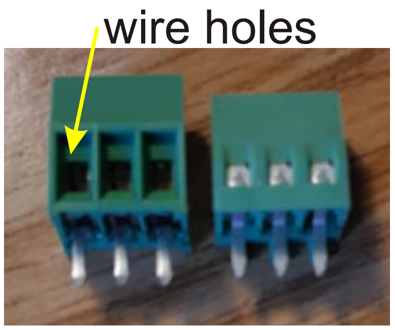
[24] https://www.creatroninc.com/product/8-pin-machine-pin-ic-socket/?search_query=ic+socket&results=114

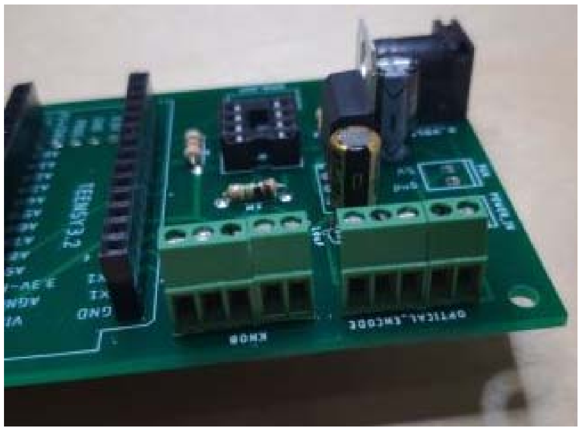


The board requires a few additional components including resistors, transistors and capacitors as shown here.

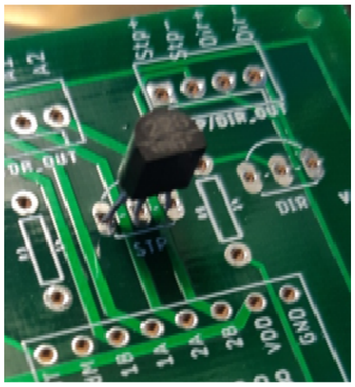


When mounting the connectors, it is important to note the side of the connector which provides the holes into which wires are inserted for connection. This is shown here. One the left side, we see the connector with the wire holes. The connector on the right side shows that back of the connector. I found that it was best to mount the connectors with the holes facing the outside of the board**.** This will allow you to connect the wires more conveniently than if the holes are facing the inside of the board. This is true for all the screw connectors which are mounted on the board. It should look something like this when completed. However, depending on the enclosure used this may not apply.

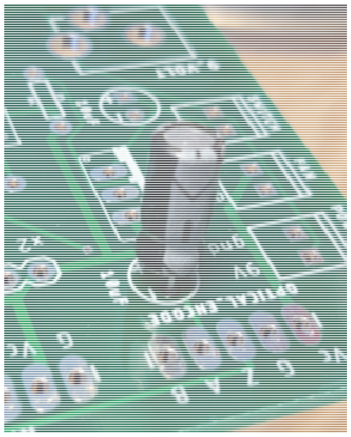




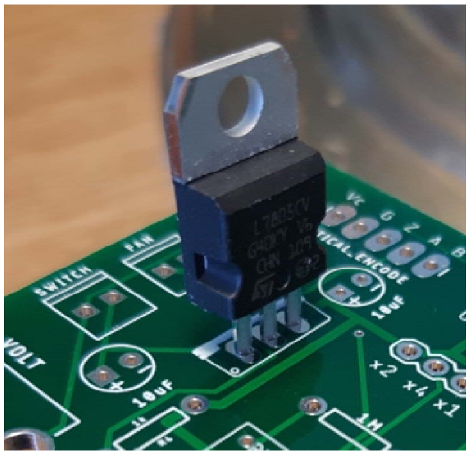
When mounting the transistors we need to take care so that the transistor has the right polarity. Note that the transistor has a flat side on it. When make sure that the outline of the transistor matches that little diagram shown on the PCB. It should look like this when it is soldered in place.



Similarly, when mounting the electrolytic capacitor (10uF) you need to pay attention to their polarity. Usually, the capacitors indicate which terminal is negative and this should be inserted into the hole with the negative sign. When the capacitor is properly mounted it should look like this with the negative terminal of the capacitor inserted into the hole marked with a "-" sign on the printed circuit board.

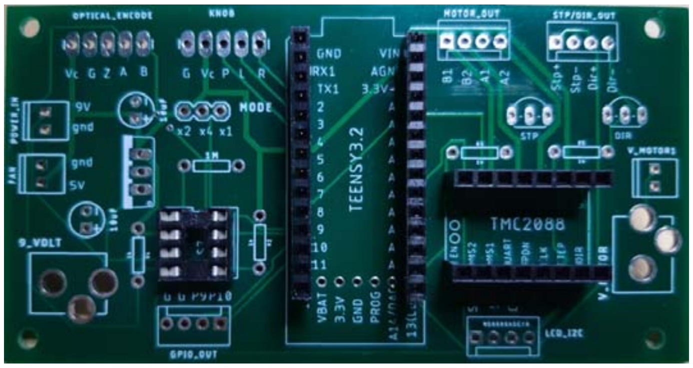


When mounting the voltage regulator, it too is polarized and must be inserted correctly. The board shows the location of the voltage regulator with a thick line on one side. This is the side which corresponds to the heat sink on the regulator. It should look like this when properly mounted.

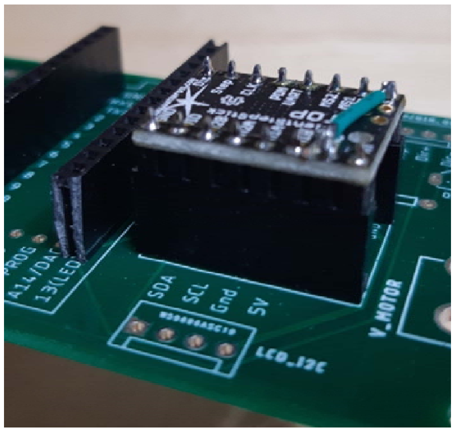


Soldering in Headers for the Breakout Boards

I have designed this board so that one can insert the Teensy3.2 and TMC2088 breakout boards without soldering them to the board. This is done by soldering headers into the board. The headers look like this.

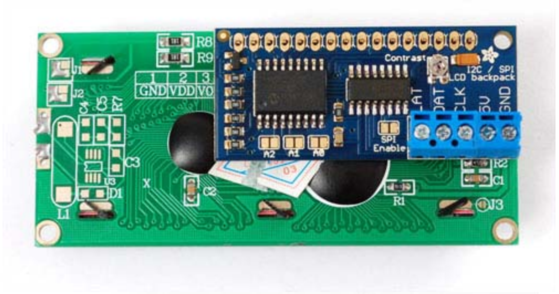


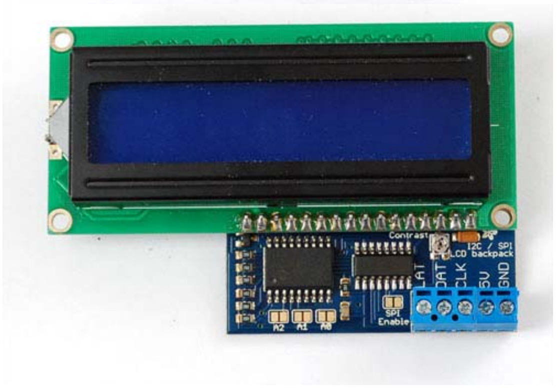
The best way to solder these in place is to put the headers onto the breakout board for which they are intended and then insert it into the PCB board like this. You can then flip it over and confidently solder the pins to the board in the knowledge that the headers are parallel to the pins on their intended board.



Mounting the I2C Backpack

In order to wire up the LCD display for an I2C interface, we must mount the I2C "Backpack" onto the LCD display. This requires a little soldering. I suggest you first read and understand the AdaFruit webpage on this installation found here (https://learn.adafruit.com/i2c-spi-lcd-backpack/assembly). Follow these instructions and you can't go wrong. They point out that there are two ways to mount the backpack, so you need to think a bit about how this will be mounted into the final project box which might tell you which way it should go. These are only mechanical changes. The electrical connections for both mounting methods are identical. When finished it should look this either of these two configurations.

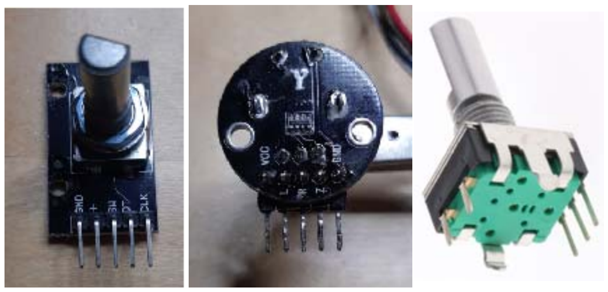




You can see that the left configuration takes up less surface area but is a bit deeper. The option on the right takes up a bit more real estate. In either case, the connections to the blue connector need to be visible as shown here. In these photographs, a 2x20 LCD is shown and not the desired 4x20 LCD display which is a bit bigger. So, don't worry if the photos don't completely agree with what you have.

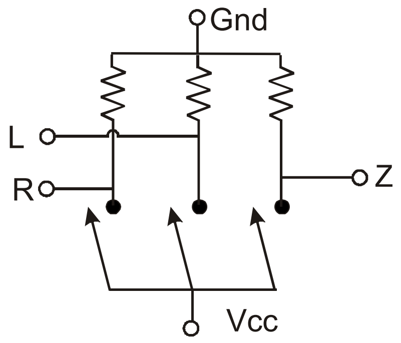
Wiring up the Rotary Encoder

There are a lot of little rotary encoders which you can buy as shown here. They are all mechanical and differ only slightly. However, it is best to understand their differences. I have indicated that you could get any of these from Creatron all of which have 5 leads. These look like this. The left hand two are already wired with pull-up resistors mounted on the breakout board, while the encoder on the right is not on a break-out board and has no pull-up resistors. The left hand encoder (with the rectangular board) has labels (Gnd, +, SW, D, Clk) while the middle one (with the circular board) has labels (Vcc, L, R, Z, Gnd). The labeling is rather curious but here is what is going on inside these encoders.



EC-11 Encoder-Round Break-Out Board

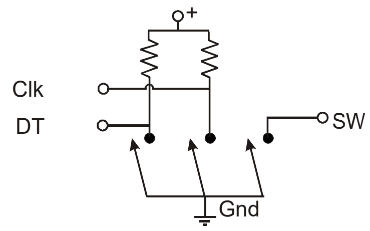
This is the one that I specify in the list of material and comes from Creatron by model number EC-11. It has three pull-up resistors all of which are tied to the "Gnd" connection. The L and R connections are for the rotary encoder while the Z is tied to the push button. To connect this to the board we need to make the proper connections. Looking at the EGB board we see there is a connection spot labeled "knob". This is the connection to the rotary encoder. There are 5 connections labeled Vc, Gnd, P, L and R. These correspond to the 5 volt input line (Vc), the ground connection (Gnd), the push button (P), the left hand quadrature encoder switch (L) and the right hand quadrature encoder switch (R). The encoder shown here gets wired as:



Connection on PCB gets wire to →	This connection on the rotary encoder
Vc	Vcc
Gnd	Gnd
P	Z
L	L

R	R

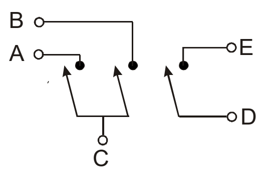
KT040 Encoder - Rectangular Break-Out Board

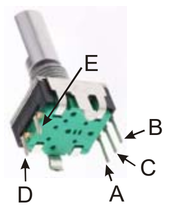


This is another possibility you can buy from Creatron with model number KY040. It has two pull-up resistors just for the rotary encoder portion. Here is the circuit. The (Clk) and (DT) connections are for the rotary encoder while the (SW) is tied to the push button. To connect this to the board we need to make these connections.

Connection on PCB gets wire to →	This connection on the rotary encoder
Vc	+
Gnd	Gnd
P	SW
L	DT
R	CLK

PEC-11R Bourns Rotary Encoder





Finally, this is a third kind which is even simpler. In fact the other two use this encoder on the break-out board as the encoder and push button devices. Here we just have the encoder without the break-out board and no pull-up resistors. It looks like this. To wire this we don't need to use the Vc connection on the PCB. Instead the wiring looks like this.

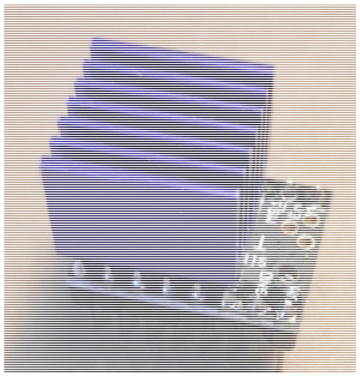
Connection on PCB gets wire to →	This connection on the rotary encoder
Vc	Not connected
Gnd	C and D or C and E

P	If D was grounded then use C If C was grounded then use D
L	A
R	B

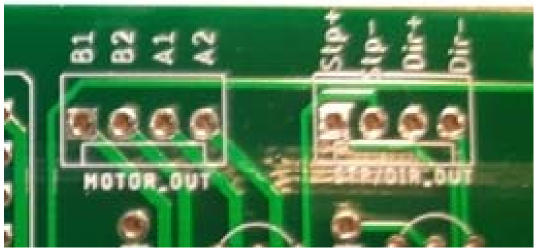
You don't really need the pull-up resistors as I have included them in the code for the software to add pull-up resistors on the Teensy3.2 board. This is by far the cheapest solution (~$2.5).

Using the Internal Stepper Motor Controller

If you use the TMC2088 controller inside a box, it might get a bit warm depending on your motor settings. I recommend experimenting with it first to see if it gets too hot when used in the final applications. If it does, you can add a small heat sink and might consider adding a small 5V fan to the box and provide venting holes. I have added a connector to power a small 5V fan on the board if needed. The fan together with the heat sink should manage the power issues. You will need to set the current through the motor as explained on the Trinamic data sheet[25]. This particular data sheet is for the TMC2130, but the procedure is the same for the TMC2088.



If you used the on-board motor controller, you will need to route the wires from the connector which shows connections B1,B2, A1,and A2 on the "MOTOR_OUT" connector. These go directly to the motor cables. The connector B1 and B2 should go across one motor windings while the A1 and A2 connections go to the second motor winding. If the motor goes the wrong direction for the application at hand, just reverse one set of these windings. For example, swap the wires going into B1 and B2 ***OR*** A1 and A2. It doesn't matter which you choose, but just swap one of the leads to the stepper motor. In this case, you will need to provide a second power supply to power the motor controller board. This supply can be anywhere from 9 – 24 volts and it connected to



25 https://wiki.fysetc.com/TMC2130/

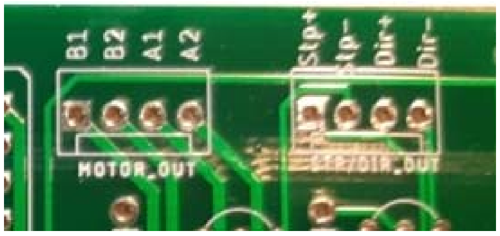
the barrel connector labeled V_motor or to the screw terminals of the same name. When mounted in a project box, the screw terminal would be the more convenient. I find the barrel connections are useful during trouble shooting and testing to allow convenient power connection with the board as all wall power supplies generally come with 2.1mm barrel connections. A SPDP switch can be used to switch both the V_motor and 9V power supplies to the board with a single switch.

Powering the TMC2088 Controllers

As mentioned earlier it is possible to provide a different power supply for the controllers if more power is needed. However, I have also used the same power supply that is regulated to give 5V for the motor power supply by jumping connector J1 and J8. This means if you use a 12 volt supply for the voltage regulator (→5volts) that same supply can be used for the motors. The LM7805 can safely regulate up to 18 volts although I rarely used anything above 12 volts. Shorting the positive terminals of J1 to J8 will provide the power to the TMC2088 at whatever voltage you apply to J1. I have found this useful for situations that don't require a lot of torque. However, if more torque is needed the TMC2088 can be run with a power supply of up to 28 volts (1.5 amps) with a little heat sink added. I've also provided a connector (J5) to power a small 5 volt fan to cool the voltage regulator and TMC2088 if needed.

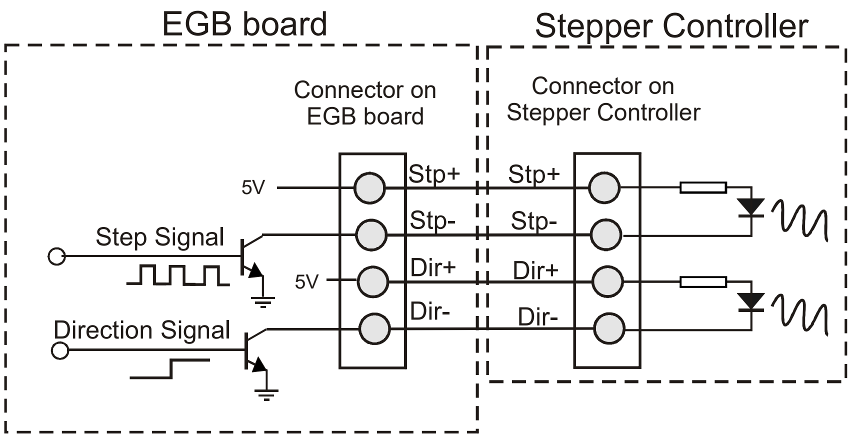
Using a Separate Stepper Motor Controller

In this case, the objective will be to use the board to drive a separate stepper motor controller which can deliver more torque to the application. In this situation don't insert the TMC2088 into its header. Just leave those header empty. I have provided a set of output connections for this situation. These are labeled Stp+,Stp-, Dir+, Dir- on the "STP/DIR Out" connector (J10).



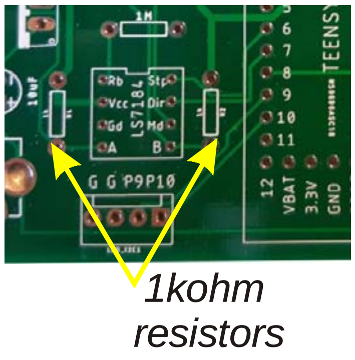
This shows what is going on with respect to the stepper motor controller board and the ECB board. Most stepper motor controllers have opto-isolated inputs to prevent any high voltage transients influencing the MCU circuitry. This is done by pulsing LED's which communicate their signals optically to so-called "open collector" NPN transistors. To wire it up properly, we connect the Stp+ on the EGB to Stp+ on the controller board. Do the same for the rest of the connections to their corresponding label on the controller board. Usually the stepper controller has an "enable" connection which needs to be managed as suggested in the motor controller specifications.

When using a separate motor controller you can find many on the market. Probably the most standard is made by Gecko[26]. They are something of an industry standard and very high quality. They provide for micro-stepping and other clever power management techniques. However, they are expensive. An alternative can be found at StepperOnline. I have experimented with these and found them to work well and are very cost effective, capable of driving up to 4.5 Amps[27]. The same company provides very inexpensive power supplies[28] which you will need. In this case, we don't need to add the 2nd power supply for V_motor and require only the 9V supply. Again a switch is helpful to turn the circuit on-off.



Special Considerations for the Encoder Wiring

The encoder I have suggested for this project is the YUMO E6B2-CWZ-3E. Each output channel runs in one of two possible modes: "open collector mode" or "voltage output mode" depending on the details of the encoder you get. Voltage output is simple and you don't need to add anything to it to make it work. In contrast, "open collector mode" means that it needs to be correctly wired through a load resistor. I have provided a location for these resistors as shown here. Assuming you are using a YUMO encoder, you can consult (http://www.yumoelectric.com/6mm-24VDC-shaft-incremental-encoder-E6B2-CWZ6C-Rotary-Encoder-pd41603666.html) to determine which mode your encoder is using. For example, the YUMO



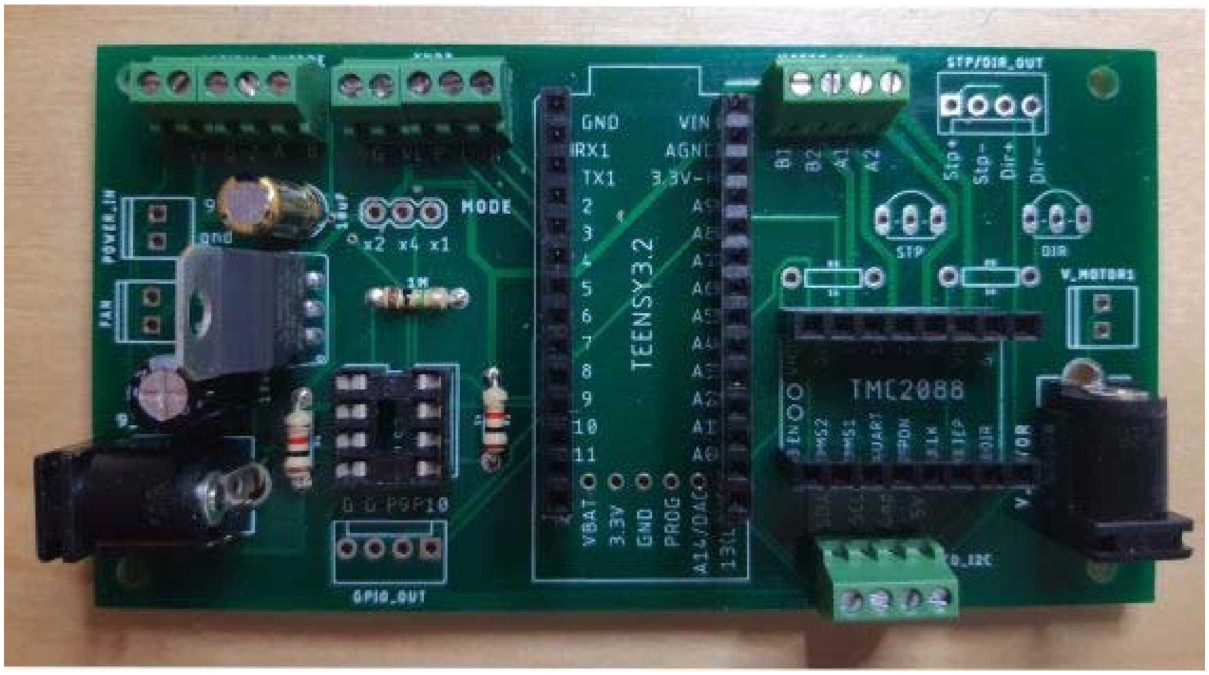
[26] https://www.geckodrive.com/
[27] https://www.omc-stepperonline.com/digital-stepper-driver
[28] https://www.omc-stepperonline.com/power-supply

E6B2-CWZ-3E runs in voltage output mode and doesn't require a resistor. Alternatively, the YUMO E6B2-CSZ-3C runs in open-collector mode which means that you need to add the two 1 Kohm resistors in the locations shown in this photograph. You might be using a different encoder altogether and thus it is best to look at the specification of the encoder you are using to determine if it is open collector or not. Here is what you need to know.

Type of encoder output	**1 kohm Resistor**
Open collector	Include
Voltage output	Do **NOT** include

Finished Board

When assembled the finished board looks something like the one shown on the next page.. You will note that not all the socket locations are filled. This is because I am planning on using this board to drive a small stepper motor with the on-board TMC2088 controller from the encoder. In this configuration, I just need to occupy the Motor Out, KNOB, LCD_I2C and Optical Encoder locations with screw connectors. In this configuration I don't need the STP/DIR_out connectors or the two output transistor/resistors. I did not put in the POWEWR_IN, Fan, V_motor1 or GPIO connectors as I will not use them. Instead I will use the two 2.1mm Barrel Jack to power the logic and stepper motor directly .



Depending on what configuration of use, different connectors might be used. In this photo, I've not inserted the Teensy3.2, the TMC2088 nor the LS7184 chips to make it easier to see what a finished board should look like. Obviously, these chips will need to be included for proper operation. Thus it is important when inserting these components be to align them so that they are in the correct orientation to ensure proper pin connections.

APPENDIX 1 - Arduino Code for Gear Hobbing

In this section, I am assuming that the Arduino IDE is available to the reader and he knows how to write basic Arduino sketches and upload them to the Teensy MCU. The structure of all possible applications of the EGB will be similar to that shown below but change somewhat depend on how much complexity is desired and the application at hand. To show how to do this, I will explain the basic components of the code. I will use the code for the electronic hobbing system as an example and then show the code for the electronic lead screw in an Appendix. You will see that they are very similar in layout and function .

Note: any code followed by // is a comment statement. Lines ending in ";" are lines of functioning code. Here is the code for the Hobbing application.

```
/* Code for a hobbing machine.
 - May 20  just gathers the pitch and data
 - May 21 - added the interrupt routine for the spindle encoder
 - May 24 - added the button routine
 - May 24 -modified ELS software to work for hobbing machine
 - MAY 30 - Adjusted the text on the LCD to add a title
 - May 31 - added the input revolutions and output revolutions as a factor
 - June 6 - added the code for the microstepping in the comment section only for electronics reference... no code implemented

    TMC2088 Stepper Motor microstepping code.........................................
              MS2=HIGH  MS2=LOW
      MS1=HIGH      1/16     1/2
      MS1=LOW       1/4      1/8
    .........................................................................
 -July 17      - modified for the Digital Gear Box PCB which uses the LS7184 Quadrature decoder chip.
               - set up Teensy to drive MS1 and MS2 levels on the TMC2088 driver. Working on this date

 -August 6     - modified for the Electronic Gear Box PCB (rev 2.0) which has both TMC2088 outputs and open collector outputs to separate driver
               - added two GPIO pins (9 and 10) as inputs for general purpose applications in future.

 -Oct 8 - added code to input the gear hobb pitch (imperial and metric) to calculate the gear blank diameter. Following definitions: N=number of teeth, PD=pitch diameter (mm or inches)
        For imperial we use 'diametral pitch'(DP)=N/PD(inches),  PD(inches)PD=N/DP,  blank diameter=(N+2)/DP, tooth depth=2.25/DP ()
        For metric we use 'module' (m)=PD(mm)/N, PD(mm)=m*N, blank diameter=PD+2m, tooth depth=2.25m
               these are taken from websites:
               https://khkgears.net/new/gear_knowledge/abcs_of_gears-b/basic_gear_terminology_calculation.html
               https://www.engineersedge.com/gear_formula.htm

 -Oct 8 - This code is for the PCB "Electronic Gear Box Rev 2.1" . This one has two power supplies with the printing on the inside of the PCB for each connector
*/

 #include <Adafruit_LiquidCrystal.h>
 Adafruit_LiquidCrystal lcd(0);

 static byte pinA = 2;                  // Our first hardware interrupt pin is digital pin 2 for data select encoder to drive the PinA interrupt routine
 static byte pinB = 3;                  // Our second hardware interrupt pin is digital pin 3 for data select encoderto drive the PinB interrupt routine
 static byte button_pin=4;                // the button for the knob rotary encoder push button;
 static byte trig_Z_pin=5;               // the pin which is connected to the index pin of the encoder.  We are not using this is this routine but still available.
 static byte trig_AB_pin=6;               // the pin which is triggered by the LS7184 as input stepper pulses to the count interrupt division routine
 static byte direction_in_pin=7;            // the pin which senses the direction by the LS7184
 static byte direction_out_pin=8;            // the output pin which sets the direction level
 static byte GPIO1_pin=9;                // the pin which is triggered by the LS7184 as input stepper pulses to the count interrupt division routine
 static byte GPIO2_pin=10;                // the pin which is triggered by the LS7184 as input stepper pulses to the count interrupt division routine
 static byte MS2_pin=11;                 // the pin to drive MS2
 static byte MS1_pin=12;                 // the pin to drive MS1
 static byte stepper_pin=13;               // the output pin for the stepper pin for the motor driver
 static byte MS1=0;                     // set value of MS1 (see table above for settings)
 static byte MS2=0;                     // set value of MS2 (see table above for settings)
 volatile byte aFlag = 0;               // let's us know when we're expecting a rising edge on pinA to signal that the encoder has arrived at a detent
 volatile byte bFlag = 0;               // let's us know when we're expecting a rising edge on pinB to signal that the encoder has arrived at a detent (opposite direction to when aFlag is set)
 volatile int encoderPos = 0;              // this variable stores our current value of encoder position. Change to int or uin16_t instead of byte if you want to record a larger range than 0-
255
 volatile int oldEncPos = 0;              // stores the last encoder position value so we can compare to the current reading and see if it has changed (so we know when to print to the serial
monitor)
```

```
 volatile byte reading = 0;                        // somewhere to store the direct values we read from our interrupt pins before checking to see if we have moved a whole detent
 int lower_tooth=6;                               // defines the smallest number of teeth that can be cut
 int upper_tooth=160;                              // defines the largest number of teeth that can be cut
 int hob_encoder_resolution=4096;                   // number of pulses per revolution of the spindle encoder after the LS7184 demodulation
 int spindle_steps_per_rev=16000;                  // # motor steps needed to drive the spindle one revolution. .
 int gear_tooth_number;                            // the desired number of teeth to cut
 byte ButtonState = 0;                            // a parameter for the button push algorithm
 byte oldButtonState=0;                            // a parameter for the button push algorithm
 byte mode_select=1;                               // a parameter to define the programming versus operation settings, "0" allows the machine to hobb, "1" puts it in programming mode
 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 for each hob rotation, this is calculated in the programme
 volatile long delivered_stepper_pulses=0;         // number of steps delivered to the lead screw stepper motor
 volatile float Direction;                         // a variable to register direction
 volatile int old_time;                           // a dummy variable
 volatile float calculated_stepper_pulses=0;       // defines the lower divisor for the initial condition which is turning-normal (divisor 128)
 float rev_in;                                   // debugging parameter to track the number of input rotations to encoder
 float rev_out;                                  // debugging parameter to measure the number of spindle rotations delivered
 float gear_diameter;                              // the gear blank diameter
 float cut_depth;                                 // the depth of the gear cut
 float DP;                                      // the diametral pitch of the hobb cutter in imperial units
 float mod;                                      // the module of the hobb cutter in metric units
 int button_status=0;                              // the button status, this number ranges from 0-->1-->2 to run the hobbing machine (0), innput the number of teeth(1) or input the hobb_pitch (2)
and
 int encoder_counter;


void setup() {
  pinMode(pinA, INPUT_PULLUP);                    // set pinA as an input, pulled HIGH to the logic voltage (5V or 3.3V for most cases)
  pinMode(pinB, INPUT_PULLUP);                    // set pinB as an input, pulled HIGH to the logic voltage (5V or 3.3V for most cases)
  pinMode(button_pin,INPUT_PULLUP);                // set the button pin as an input
  pinMode(trig_Z_pin,INPUT);                     // enables trig_Z_pin to act as an input from the optical encoder
  pinMode(trig_AB_pin,INPUT);                     // set trig_pin_A&B as an input pin from output of the LS7184 quad decoder
  pinMode(direction_in_pin,INPUT);                 // set the direction_in_in as an input
  pinMode(direction_out_pin,OUTPUT);                // set the direction_out_pin as an output
  pinMode(GPIO1_pin,INPUT);                      // set the GPIO1_pin as an input
  pinMode(GPIO2_pin,INPUT);                      // set the GPIO1_pin as an input
  pinMode(MS1_pin,OUTPUT);                       // enable MS2_pin as an output
  pinMode(MS2_pin,OUTPUT);                       // enable MS2_pin as an output
  pinMode(stepper_pin,OUTPUT);                    // set stepper pin as an output pin
  attachInterrupt(2,PinA,RISING);                  // set an interrupt on PinA, looking for a rising edge signal and executing the "PinA" Interrupt Service Routine (below)
  attachInterrupt(3,PinB,RISING);                  // set an interrupt on PinB, looking for a rising edge signal and executing the "PinB" Interrupt Service Routine (below)
  attachInterrupt(6,count, RISING);                // enable the interrupt for the hobb encoder driven by the LM7184 encoder quad chip
  Serial.begin(9600);                            // start the serial monitor link
  lcd.begin(20,4);                              // initiate the LCD driver for a 20x4 pixel display
  gear_tooth_number=lower_tooth;                   //start of with gear number at lowest value
  digitalWrite(MS1_pin,MS1);                     //assigns value of MS1 to the TMC2088
  digitalWrite(MS2_pin,MS2);                     //assigns value of MS2 to the TMC2088
}

void PinA(){
  cli(); //stop interrupts happening before we read pin values
  reading = PIND & 0xC; // read all eight pin values then strip away all but pinA and pinB's values
  if(reading == B00001100 && aFlag) {          //check that we have both pins at detent (HIGH) and that we are expecting detent on this pin's rising edge
    encoderPos --; //decrement the encoder's position count
    bFlag = 0; //reset flags for the next turn
    aFlag = 0; //reset flags for the next turn
  }
  else if (reading == B00000100) bFlag = 1; //signal that we're expecting pinB to signal the transition to detent from free rotation
  sei(); //restart interrupts
}

void PinB(){
  cli(); //stop interrupts happening before we read pin values
  reading = PIND & 0xC; //read all eight pin values then strip away all but pinA and pinB's values
  if (reading == B00001100 && bFlag) {       //check that we have both pins at detent (HIGH) and that we are expecting detent on this pin's rising edge
    encoderPos ++; //increment the encoder's position count
    bFlag = 0; //reset flags for the next turn
    aFlag = 0; //reset flags for the next turn
  }
  else if (reading == B00001000) aFlag = 1; //signal that we're expecting pinA to signal the transition to detent from free rotation
  sei(); //restart interrupts
}

void button_detect()             //detects a button push
      {
         ButtonState = digitalRead(button_pin);                  // get the current state of the button
         if (ButtonState == HIGH && oldButtonState == LOW)        // has the button gone high since we last read it?
        { mode_select++;
        encoderPos=1; }                                  // toggle the mode_select parameter
        if(mode_select>2){mode_select=0;}                       // allows the mode_select parameter to range from 0-->2
         oldButtonState = ButtonState;                         // a parameter for toggling the button state
        }
void gear_parameters()              //this defines the factors
{

        if (mode_select == 0)                                    //mode_select==0 to arm the hobbing machine to cut a gear, which I call "run"
         {
                         rev_in=1.0*input_counter/hob_encoder_resolution;
                         rev_out=1.0*delivered_stepper_pulses/spindle_steps_per_rev;
                 //        lcd.setCursor(0,3);  lcd.print("in="); lcd.print(rev_in); lcd.print("   out="); lcd.print(rev_out);


        lcd.setCursor(15,0);                                   // set up position on the LCD screen
        lcd.print(" run ");                                   // display "run" on the LCD screen to show that we are armed to operate the machine for hobbing
          }


  if(mode_select==1)                                         //mode_select==1 for inputing the gear tooth number which calculates the reduction factor
         {
                lcd.setCursor(16,0); lcd.print("prog");     // displays "prog" on the LCD screen near the bottom right and the stepper motor is inhibited.
                if(encoderPos>upper_tooth){gear_tooth_number=upper_tooth;encoderPos=upper_tooth;}
                if(encoderPos<lower_tooth){gear_tooth_number=lower_tooth;encoderPos=lower_tooth;}
                if((encoderPos>=lower_tooth)&&(encoderPos<upper_tooth)){gear_tooth_number=encoderPos;}
                factor=spindle_steps_per_rev*1.0/(gear_tooth_number*hob_encoder_resolution);              //the factor needed for stepper motor #pulses/rev and encoder resolution
                lcd.setCursor(0,0);  lcd.print("# Teeth= "); lcd.print(gear_tooth_number);lcd.print("   ");    // record number of desired teeth
                lcd.setCursor(0,1); lcd.print("                 ");
                lcd.setCursor(0,2);lcd.print("                 ");
                lcd.setCursor(0,3);lcd.print("                 ");
         }

  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, DP=diametral pitch, mod=module
                                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)
                           }

                        delivered_stepper_pulses=0;
                        input_counter=0;
         }
```

```
    }


void count()    //this is the interrupt routine for the floating point division algorithm
 {
   Direction=digitalRead(direction_in_pin);
   digitalWrite(direction_out_pin,Direction);
   input_counter++;                                    // increments a counter for the number of spindle pulses received
   calculated_stepper_pulses=round(factor*input_counter);       // calculates the number of stepper pulses 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(5);                                  // 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 by one to reflect the pulse just delivered
     }
  }

void loop()
{
 button_detect();
 gear_parameters();
}
```

The program has 220 lines of tedious code, most of which will never be of interest to most people. However, depending on the application, there are a few bits of code which need to be understood so that one could change the operation of the EGB if needed.

First however, a few words about Arduino programmes or “sketches” as they are called in Arduino jargon. All Arduino sketches are composed on a minimum of 4 parts:

Part 1) definition of constants and other files needed to link to the code,
Part 2) a “void setup()” section which runs some basic code once and often used to initiate various devices or other routines and
Part 3) a “void your_favorite_name()”section which is similar to a subroutine. This performs some specific function. Simple programmes don’t need these and everything can be done in the next section called “void loop()”.
Part 4) a “void loop()” section which pulls all the subroutines or other code together to run the actual code in a repetitive fashion.

Unlike most software one encounters, the routines which run on a microcontroller (MCU) run continuously in loops. That means there is a section which continuously repeats the programmer over and over until the power to the MCU is terminated. This section is called *loop().* So, a little progam would look like:

```
Void loop()
{
        a line of code1;
        another line of code2;
        .
        .
        another line of codeN;
}
```

Everything between the brackets { and } is repeated over and over. That means that the machine first runs *code1*, then runs *code2*, moving down until it runs *codeN*. At this point it returns back to the top to run *code1, code2… codeN* again. It repeats this loop continuously. That is why this section of the code is called *void loop().* It is not important that you understand the significance of the word “void” (it means that the routine does not return parameters). Now, depending on the content and complexity of the lines of code, the time taken to complete one loop will vary. However, the Teensy3.2 runs at 72 Mhz. So, it is likely for the code I show above would take about 100 microseconds for each loop. Thus it will loop about 10,000 times every second. This is not considered fast for proper computers which have much higher clock speeds, but it sufficient for most machine control problems for which we use MCU’s. If you look at the code for the hobbing machine you will see the loop routine reads:

```
void loop()
{
  button_detect();
  gear_parameters();
}
```

What this is saying is that it first looks to see if the button on the rotary switch has been pressed. Then it runs another routine called *gear_parameters* which chooses which gears we want to build.

Now, when it sees the code *button_detect(),* it reverts to an earlier section of code with the same name. Looking at the code again, you can see that section earlier in the code which reads:

```
void button_detect()              //detects a button push
 {
  ButtonState = digitalRead(button_pin);     // get the current state of the button
  if (ButtonState == HIGH && oldButtonState == LOW)        // has the button been pressed?
  { mode_select++;
  encoderPos=1; }                              // toggle the mode_select parameter
  if(mode_select>2){mode_select=0;}      // allows the mode_select parameter to range from 0-->2
  oldButtonState = ButtonState;            // a parameter for toggling the button state
 }
```

This is a simple piece of code that starts with its name: *void button_detect().* Next you'll see the text: *//detects a button push.* Anything that starts with a " // " means that the text that follows on that line is not code but is just some text to explain what that line is doing. This is referred to as a "comment statement". It is just there to remind the programmer what she was thinking when she wrote the code and what it did. Good programming practice encourages lots of comments statements so that another person can understand what the code is doing.

Next in line, you see a *{* symbol. At the bottom of this section of code you see another *}* symbol. That means that the code between these brackets are the content of the code by the name *button_detect*.

Looking between these brackets you see 6 lines of code which read.

```
 ButtonState = digitalRead(button_pin);     // get the current state of the button
 if (ButtonState == HIGH && oldButtonState == LOW)        // has the button gone high since we last read it?
{ mode_select++;
encoderPos=1; }                              // increment the mode_select parameter
if(mode_select>2){mode_select=0;}                   // allows the mode_select parameter to range from 0-->2
 oldButtonState = ButtonState;                      // a parameter for toggling the button state
```

This is the actual program that is being run when we encounter *button_detect.* What is going on here? I'll go through it line by line to explain how it works.

First we see:

```
ButtonState = digitalRead(button_pin);     // get the current state of the button
```

This means that we are going to read whatever signal appear on a specific pin on the Teensy3.2. That pin has the name *button_pin*. Near the very top of the whole program we will find in the parameter definition section a line of code which states:

```
static  byte button_pin=4;                    // the button for the knob rotary encoder push button;
```

This is saying the pin #4 on the Teensy3.2 is reserved for that button. So, the line

```
ButtonState = digitalRead(button_pin);     // get the current state of the button
```

Is saying, “go to pin 4”, measure the voltage on that pin. If it is ~ 5 volts assign the value of 1 or “HIGH” to the parameter *ButtonState*. However, if the voltage is around 0 volts, assign the value 0 or “LOW” to the parameter *ButtonState*.

Then we go to the next line which states:

```
if (ButtonState == HIGH && oldButtonState == LOW)        // has the button gone high since we last read it?
```

This is saying if *Button_State* is *HIGH* **and** the last time looked at Button_State (“*oldButtonState*”) which we measured a few microseconds earlier was LOW means we just pushed the button a new time. If that happened then do:

```
{ mode_select++;                              // increment the mode_select parameter
encoderPos=1; }
```

This says let us increment the value of a parameter call *mode_select* by a value of 1. So, if it was previously at a value of 2, it is now 2+1=3. For the time being let’s not worry about the code which says *encoderPos=1;*

Next we test *mode_select* with the line of code which reads:

```
if(mode_select>2){mode_select=0;}     // forces the mode_select parameter to range from 0-->2
 oldButtonState = ButtonState;           // a parameter for toggling the button state
```

This says if the value of mode_select is greater than 2, set its value to zero. This means that if mode_select was 0, leave it at zero, if it was 1 leave it at 1, if it was 2, leave it as 2. But if it was 3, now set it back to 0. This means that the value of mode_select can have only 3 values: 0, 1 or 2… nothing else.

The next line of code reads:

```
oldButtonState = ButtonState;                    // a parameter for toggling the button state
```

This says to set the value of "oldButtonState" to the current value of Button_state. Recall that we used this a few lines above in *if (ButtonState == HIGH && oldButtonState == LOW) .* That is how *oldButtonState* got set and is a good way to know that you just push a button.

Perhaps from this little review of this tiny piece of code, you can see how the Arduino code works. It is entirely logical and fairly transparent to its operation. The rest of the code, while long and tedious, is just the same kind of thinking. However, this is just to give you a sense of how it works.

Editing the Hobb Code

The description in the previous section was just to provide a crude understanding of how a tiny piece of the overall code for the hobbing example works. For the most part, there is no need to adjust the code. However, let's say you had a hobbing machine which is not the same as the one for which this code was written. Also, perhaps the hobbing encoder had a different number of pulses per revolution. Alternatively, we might have different pitches of hobbs available. How would the code need to be changed?

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. How, was this parameter defined? 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. The letters "*int*" in front of this is just telling the MCU that this is an "integer" and tells it how to store that number as a 4 byte number.

Now why is the value 16000? Recall that we are 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 are setting 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 througha 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 metrial 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 hobbs and the change between the number of imperial and metric hobbs. That should be all the changes needed to incorporate new hobb parameters.

APPENDIX II – Arduino Code for an Electronic Lead Screw

In this section, I show the Arduino IDE to use the Digital Gear Box for an Electronic Lead Screw. It is very similar to the code for the Hobbing Application, but rewritten to account for the new task. In this copy, the lines for the comment statement feed over onto the next line but if you want to use this code, just cut and paste it (as is) into the Arduino IDE editor and the lines should rearrange OK.

Here it is:

```
        /* code to be used to drive an electronic lead screw.  It is modelled on my earlier 'PELS3' code and updated to run on the Electronic
Gear Box Rev2.1 printed circuit board. Written on 10/25/2019.


        */
         #include <Adafruit_LiquidCrystal.h>
         Adafruit_LiquidCrystal lcd(0);

         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 inches
         int motor_steps=400;                // the number of steps per revolution of the lead screw
         int tpi;                            // parameter to indicate the initial turning pitch

         //.....................................................................................................................
         // System parameters which need no alteration.
         static byte pinA = 2;                           // Our first hardware interrupt pin is digital pin 2 for data select encoder to drive the PinA
interrupt routine
         static byte pinB = 3;                           // Our second hardware interrupt pin is digital pin 3 for data select encoderto drive the PinB
interrupt routine
         static byte button_pin=4;                       // the button for the knob rotary encoder push button linevolatile byte aFlag = 0;
         static byte trig_Z_pin=5;                      // the pin which is connected to the index pin of the encoder.  We are not using this is this
routine but still available.
         static byte trig_AB_pin=6;                      // the pin which is triggered by the LS7184 as input stepper pulses to the count interrupt
division routine
         static byte direction_in_pin=7;                   // the pin which senses the direction by the LS7184
         static byte direction_out_pin=8;                  // the output pin which sets the direction level to the external motor controller
         static byte stepper_pin=13;                     // the output pin which generates stepper pulses to the external motor controller

         static byte GPIO1_pin=9;                        // an extra GPIO pin for future use, not used here
         static byte GPIO2_pin=10;                       // an extra GPIO pin for future use, not used here
         volatile byte aFlag = 0;              // let's us know when we're expecting a rising edge on pinA of the rotary encoder to signal
that the encoder has arrived at a detent
         volatile byte bFlag = 0;              // let's us know when we're expecting a rising edge on pinB of the rotary encoderto signal
that the encoder has arrived at a detent (opposite direction to when aFlag is set)
         volatile int encoderPos = 0;          // this variable stores our current value of encoder position. Change to int or uin16_t
instead of byte if you want to record a larger range than 0-255
         volatile int oldEncPos = 0;           // stores the last encoder position value so we can compare to the current reading and see
if it has changed (so we know when to print to the serial monitor)
         volatile byte reading = 0;            // somewhere to store the direct values we read from our interrupt pins before checking to
see if we have moved a whole detent

         float lathe_pitch=4.0;                // the pitch of the lathe lead screw
         byte newButtonState = 0;              // a parameter for the button push algorithm
         byte oldButtonState=0;                // a parameter for the button push algorithm
         byte mode_select=1;                   // a parameter to define the programming versus operation settings, "0" allows the
machine to hobb, "1" puts it in programming mode
         volatile long input_counter=0;        // a parameter for the interrupt to count input pulses
         volatile float factor;                // the factor to calculate the number of stepper pulses per encoder pulses, this is calculated in
the programme
         volatile long delivered_stepper_pulses=0;     // number of steps delivered to the lead screw stepper motor
         volatile float Direction;             // a variable to register direction
         volatile float calculated_stepper_pulses=0;   // defines the lower divisor for the initial condition which is turning-normal (divisor
128)
         float pitch=0.085;                    //a parameter to define the metric thread pitch which is initially defined for "normal" turning
         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

        void setup() {
         pinMode(pinA, INPUT_PULLUP);         // set pinA as an input, pulled HIGH to the logic voltage (5V or 3.3V for most cases)
         pinMode(pinB, INPUT_PULLUP);         // set pinB as an input, pulled HIGH to the logic voltage (5V or 3.3V for most cases)
         pinMode(button_pin,INPUT_PULLUP);    // set the button pin as an input
         pinMode(trig_Z_pin,INPUT);           // enables trig_Z_pin to act as an input from the optical encoder
         pinMode(trig_AB_pin,INPUT);          // set trig_pin_A&B as an input pin from output of the LS7184 quad decoder
         pinMode(direction_in_pin,INPUT);     // set the direction_in_in as an input
         pinMode(direction_out_pin,OUTPUT);   // set the direction_out_pin as an output
         pinMode(GPIO1_pin,INPUT);            // set the GPIO1_pin as an input
         pinMode(GPIO2_pin,INPUT);            // set the GPIO1_pin as an input
         pinMode(stepper_pin,OUTPUT);         // set stepper pin as an output pin
         attachInterrupt(2,PinA,RISING);      // set an interrupt on PinA, looking for a rising edge signal and executing the "PinA"
Interrupt Service Routine (below)
         attachInterrupt(3,PinB,RISING);      // set an interrupt on PinB, looking for a rising edge signal and executing the "PinB"
Interrupt Service Routine (below)
```

```
  attachInterrupt(6,count, RISING);               // enable the interrupt for the hobb encoder driven by the LM7184 encoder quad chip
  Serial.begin(9600);                             // start the serial monitor link
  lcd.begin(20,4);                                // initiate the LCD driver for a 20x4 pixel display

  factor= (motor_steps*pitch)/(lead_screw_pitch*spindle_encoder_resolution); //calculates the initial factor when turning on the
ELS to deliver a normal turning operation of "medium"

   //................This next section sets the system to Nnormal Imperial Turning when the unit if first turned
on.............................................
   lcd.setCursor(0,0);
   lcd.print("Turning");
   lcd.setCursor(0,1);
   lcd.print("Normal    ");
}

void PinA(){
  cli(); //stop interrupts happening before we read pin values
  reading = PIND & 0xC; // read all eight pin values then strip away all but pinA and pinB's values
  if(reading == B00001100 && aFlag) {          //check that we have both pins at detent (HIGH) and that we are expecting detent on
this pin's rising edge
    encoderPos --; //decrement the encoder's position count
    bFlag = 0; //reset flags for the next turn
    aFlag = 0; //reset flags for the next turn
  }
  else if (reading == B00000100) bFlag = 1; //signal that we're expecting pinB to signal the transition to detent from free rotation
  sei(); //restart interrupts
}

void PinB(){
  cli(); //stop interrupts happening before we read pin values
  reading = PIND & 0xC; //read all eight pin values then strip away all but pinA and pinB's values
  if (reading == B00001100 && bFlag) {       //check that we have both pins at detent (HIGH) and that we are expecting detent on
this pin's rising edge
    encoderPos ++; //increment the encoder's position count
    bFlag = 0; //reset flags for the next turn
    aFlag = 0; //reset flags for the next turn
  }
  else if (reading == B00001000) aFlag = 1; //signal that we're expecting pinA to signal the transition to detent from free rotation
  sei(); //restart interrupts
}

void thread_parameters()                            //this defines the parameters for the thread and turning for both metric and
imperial threads
 {
        newButtonState = digitalRead(button_pin);          // 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)
           {
            if(encoderPos>34){                          //the next four lines allows the rotary select to go around the menu as a loop in
either direction
                     encoderPos=34;
                     }
              if(encoderPos<1){
                     encoderPos=1;
                     }
                       switch(encoderPos) {
                      //..........................................................................................turning data
                            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(5):    tpi=12;   break;
                            case(6):    tpi=13;   break;
                            case(7):    tpi=16;   break;
                            case(8):    tpi=18;   break;
                            case(9):    tpi=20;   break;
                            case(10):    tpi=24;   break;
                            case(11):    tpi=28;   break;
                            case(12):    tpi=32;   break;
                            case(13):    tpi=36;   break;
                            case(14):    tpi=40;   break;
                            case(15):    tpi=42;   break;
```

```
                    case(16):   tpi=44;   break;
                    case(17):   tpi=48;   break;
                    case(18):   tpi=52;   break;
               //..........................................................................................metric data
                    case(19):   pitch=0.4;   break;
                    case(20):   pitch=0.5;   break;
                    case(21):   pitch=0.7;   break;
                    case(22):   pitch=0.75;  break;
                    case(23):   pitch=0.8;   break;
                    case(24):   pitch=1.0;   break;
                    case(25):   pitch=1.25;  break;
                    case(26):   pitch=1.5;   break;
                    case(27):   pitch=1.75;  break;
                    case(28):   pitch=2.0;   break;
                    case(29):   pitch=2.5;   break;
                    case(30):   pitch=3.0;   break;
                    case(31):   pitch=3.5;   break;
                    case(32):   pitch=4.0;   break;
                    case(33):   pitch=5.0;   break;
                    case(34):   pitch=7.0;   break;
                     }
                       if(encoderPos<4){
                       factor= (motor_steps*pitch)/(lead_screw_pitch*spindle_encoder_resolution);
                       switch(encoderPos) {
                         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(encoderPos<19)
                        {
                         depth=pitch_factor/tpi;   //the depth of cut in inches on the compound slide I need for each thread
pitch.
                         factor= motor_steps*25.4/(tpi*lead_screw_pitch*spindle_encoder_resolution);   //the imperial
factor needed to account for 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 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;
                     }



          void count()    //this is the interrupt routine for the floating point division algorithm
           {
             Direction=digitalRead(direction_in_pin);
             digitalWrite(direction_out_pin,Direction);
             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(5);                              // 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 by one to reflect
the pulse just delivered
              }
            }

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

Editing the Lathe Code:

In order for it the code to work generally there are a number of user parameters which will likely need to be altered depending on characteristics of the lathe on which it will be used, the type of encoder for the spindle and the way you linked the stepper motor to the lead screw. If that is the case, one must replace a minimum of three parameters with the correct values for your lathe setup. These are:

'spindle_encoder_resolution' which is the number of pulses that the optical encoder attached to the spindle per spindle revolution

'lead_screw_pitch' which is the pitch of the lead screw in mm measured as the number of mm of lathe saddle translation per lead screw revolution

'motor_steps' which is the number of stepper pulses per lead screw rotation.

spindle_encoder_resolution

For example, if you had a spindle encoder of 200 step/rev... then edit the line which reads

spindle_encoder_resolution=1024;

to read

spindle_encoder_resolution=200;

lead_screw_pitch

In this code the default lead screw pitch is 4.0mm. Very likely your lead screw has a different pitch. For example, assume that you lead screw has a pitch of 3/8", then change

lead_screw_pitch =4.0;

to read

lead_screw_pitch =9.525;

The reason it is 9.525 is that this parameter is expecting the value to be in mm's. So we convert 3/8" to mm which is 3/8*25.4=*9.525* mm

motor_steps

In this code the default stepper motor has 200 steps/revolution. However, it also assumes a timing belt linkage of 2:1 to give a final resolution of 400 steps per lead screw revolution. So let's say your stepper motor is 200 steps revolution and you use a linkage of 3:1 . Then your motor would require 600 steps to rotate the lead screw exactly once. So, in that case change

motor_steps=400;

to read

motor_steps=600;

Adding your own thread or turning parameters

Also note that you can add a new thread pitch or alternatively remove one from the list of those available. You can find them in the routine called "thread_parameter()". You do this by simply added your new thread to the list. For example, if you were to add a new Imperial thread, say tpi=60. The new code would look like this noting that the lines which I have changed are indicated in ***bold italic***: This involves changing the limits on the ecoderPos parameter a few places and reording the list case numbers.

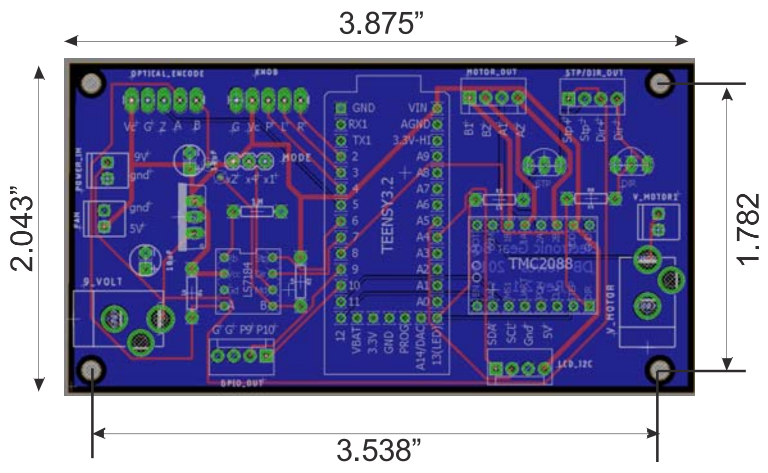
```
if(mode_select==1)
      {
        if(encoderPos>35){
                    encoderPos=35;
                    }
            if(encoderPos<1){
                    encoderPos=1;
                    }
                       switch(encoderPos) {
 //............................................................................turning data
                              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(5):    tpi=12;   break;
                              case(6):    tpi=13;   break;
                              case(7):    tpi=16;   break;
                              case(8):    tpi=18;   break;
                              case(9):    tpi=20;   break;
                              case(10):   tpi=24;   break;
                              case(11):   tpi=28;   break;
                              case(12):   tpi=32;   break;
                              case(13):   tpi=36;   break;
                              case(14):   tpi=40;   break;
                              case(15):   tpi=42;   break;
                              case(16):   tpi=44;   break;
                              case(17):   tpi=48;   break;
                              case(18):   tpi=52;   break;
                              case(19):   tpi=60;   break;    // new thread added here
          //...........................................................................metric data
                              case(20):   pitch=0.4;   break;   // reorder all following case numbers
                              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(encoderPos<4){
                                            factor=
(motor_steps*pitch)/(lead_screw_pitch*spindle_encoder_resolution);

 switch(encoderPos) {

        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(encoderPos<20)
```

What I did here was to change the limits on the ecoderPos parameter a few places and reording the list case numbers. Thus by this means you can add as many thread choices as you would like and even special ones for your unique application by editing, compiling and reloading the code into the Teensy3.2 You should only need to do this once once you have chosen your favorite thread sizes.

APPENDIX – III: EGB Mechanical Specifications

Here is a drawing of the outline of the PCB for this board from the Gerber files.


3.875”
2.043”
1.782
3.538”
OPTICAL_ENCODE
KNOB
MOTOR_OUT
STP/DIR_OUT
TEENSY3.2
TMC2088
LS7184
MODE
POWER_IN
FAN
9_VOLT
V_MOTOR
V_MOTOR1
GPIO_OUT
LCD_I2C