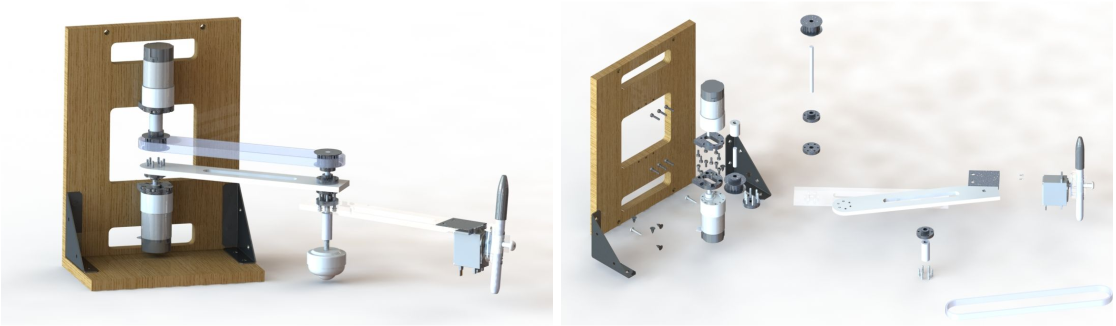
COAXIAL 2DOF PEN PLOTTER



A Report

Presented to

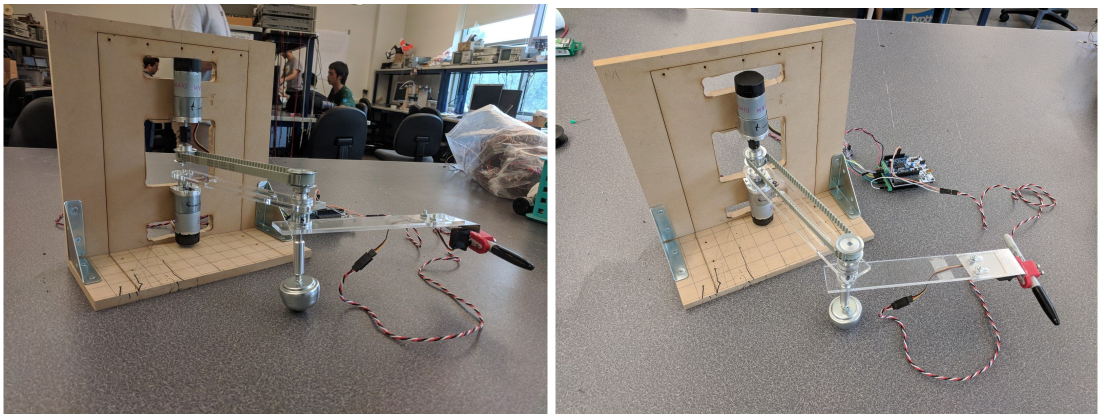
Professor Ridgely and Charlie Refvem

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Course

ME 405 Mechatronics



by

Dima Kyle || Samuel Lee

June 12th, 2018

Table of Contents

Table of Tables

Table of Figures

Introduction

The objective of this project was to design, build, program, and document a 2 (and a half axes) pen plotter, where the half axis comes from some mechanism that could simply engage or disengage a mechanism. The plotter must fulfill some given a set of design specifications and also some of our own, explained in Specifications. In the end, though we could not make a fully functional pen plotter, we were able to make pen plotter that could go through an hpgl file, parse it, raise and drop a pen, and draw (though not what we intended it to draw).

Our coaxial 2 degree of freedom pen plotter design is inspired by a senior project we came across online by a 4th year student named Gregory Bourke at Nelson Mandela Metropolitan University, Port Elizabeth, South Africa [1]. After initially researching and comparing various designs of current pen plotters that either existed on the market or have been done has home projects by others online, we decided it would be plausible to construct a robotic arm of some sort for our pen plotter. Although most of the designs researched consisted of a railing system which may have been more simple to design, we wanted to try a different approach of using coaxial fixed motors.

Our coaxial 2DOF pen plotter consists of two arms linked together from two concentric motors mounted over each other on a back plate using a gear and pulley system. A servo mounted at the end of the second arm control position of the pen hitting up or down when plotting a drawing. Our design can accept drawings saved as HPGL formatted files and plot on a standard 8.5” x 11”sheet of paper. We also wanted to make a system that could easily be modified for bigger plots. With longer link arms, our device could possibly have a larger print area, something that railing systems are limited by. The maximum footprint of our pen plotter is 18 × 8 × 11 in, so our design is targeted to be as small and portable as possible. Additionally our design allows the user to plot upside down if any such situation would arise, as the plotter can be positioned to lay flat on it’s motor plate side. This would suit independent digital artists or the general public that might wish to have tool which creates quick prototype sketches or drawings as part of a larger creative projects, from paintings, comic scripts, or animations. Additional applications of this product could be used as a signature machine for political or educational purposes, calligraphers, woodworkers, or any sort retailer wanting to generate notes to their customers.

Overall, we learned, practiced, and applied mechatronics skills of mechanical design and analysis, programming design and debugging, and the fusion of the two aforementioned. This report serves as the overall documentation so that anyone could learn and recreate our project.

Specifications

Before design and programming, an important step in any project is to delineate the specifications or requirements. These are the indication or the benchmarks that determine whether a final product addressed all the needs. For our project, the main specifications we decided are in Table 1.

Table 1. Table of specifications, final design, and completion.

Required Specifications	Final Design Specifications Used	Specifications Met?: Y/N
Chains, lead screws, toothed belts used	T5 timing belt and pulley	Y
Maximum footprint: 18” × 24”	18” × 8”	Y
Easy to assemble, fix, and adjust	N/A	Y
Positioner moves an implement	Pen	Y
“Half axis” of motion for implement	Servo	Y
Device plots of full area of standard 8.5” × 11” paper	2 DOF range of motion within 8.5” × 11”	Y
Device accepts specification from a PC file to produce drawing	HPGL format file used	Y
Complete and legible drawing*	______ drawing plotted	N
Draw time**	-	?
Drawing resolution**	-	?
Modularity**	-	?

* We unfortunately did not meet this specification fully.

** These are possible tests that could be done to characterize the full capability of the pen plotter.

Design Development

In order to avoid issues with implementation, much of our time was spent creating a sturdy design both mechanically and in terms of our programming. We brainstormed, created 3D CAD models, and diagrams before fully delving into the manufacturing and coding. We derived the kinematics by hand, which can be seen in Appendix D.

Hardware Design Development

The following designs in Figures 1 and 2 were initially researched and considered before we decided to go with the coaxial 2DOF pen plotter design. A combination of various railing systems, as well as 2 degree-of-freedom arm designs were considered to be used for our project. All of the design were researched and referenced online.

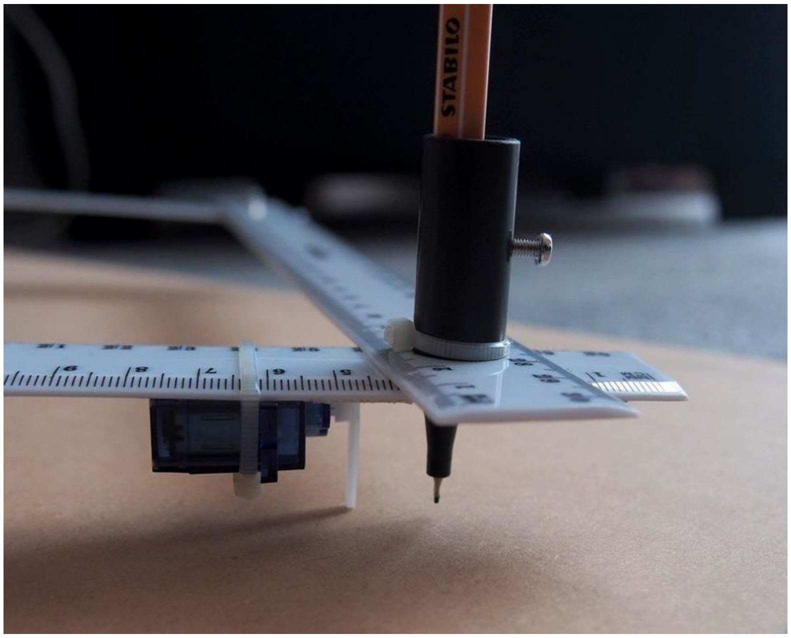


Figure 1. A dual arm pen plotter design. This system was considered, which consisted of two seperate two-bar linkage arms connected to a pen holder. The particular project pictured above was made with four plastic rulers, two NEMA17 stepper motors, and an SG90 servo pen-lift and was designed to be used with g-code output from Inkscape. Each 2DOF arm was connected to a seperate stepper motor. Additional details of the Dual Arm pen plotter design can be found on the Instructables website from the References section [2].

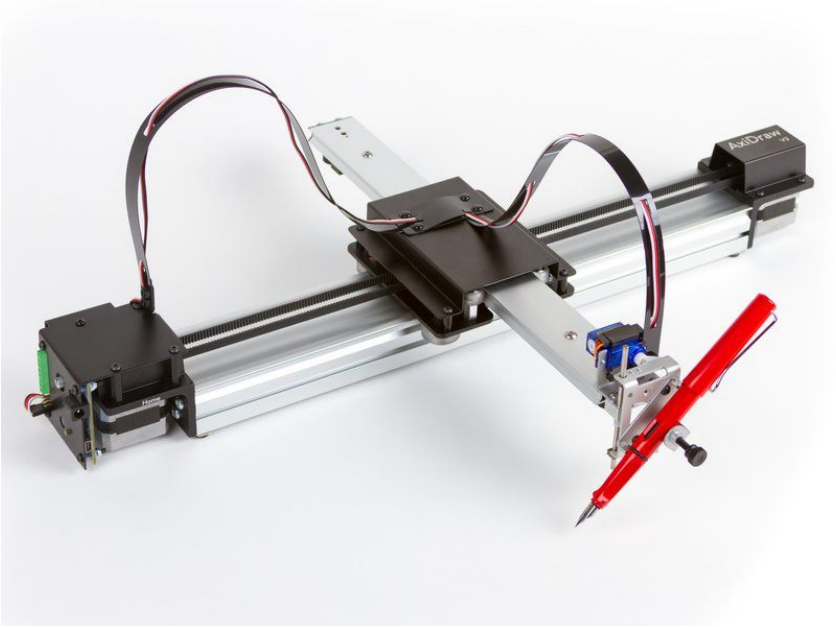
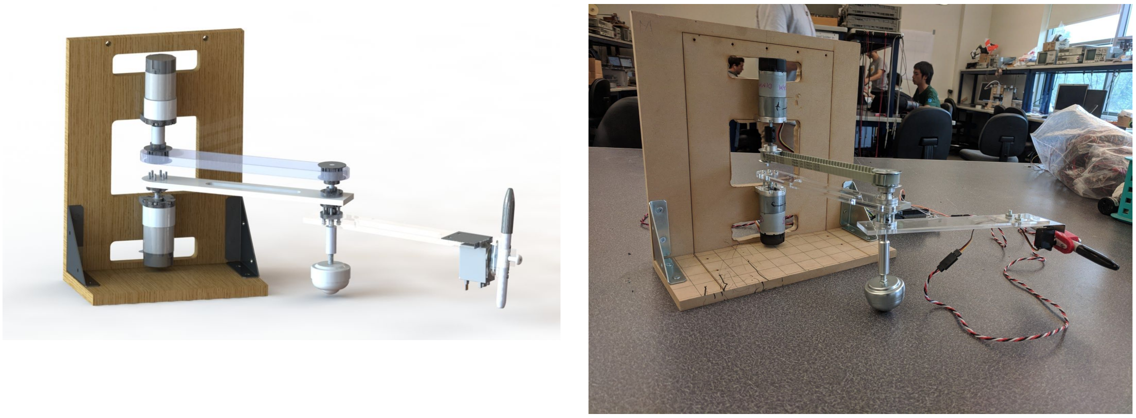


Figure 2. The AxiDraw V3 pen plotter from the Evil Mad Scientist. This was another design we looked at consisting of a dual railing system with a servo to control the pen. In particular we researched a current product from the Evil Mad Scientist which incorporates this design with their AxiDraw V3 pen plotter. The portable, compact design inspired us to make our pen plotter with the smallest footprint possible. Additionally, the pen holder design mounted at one end of the center railing using a servo as pictured above was a feature we tried to implement in our design. Additional specifications on the product can be found from their datasheet in the References section [4].

We wanted to use a design that combined both of these ideas. Thus, we decided on making a coaxial fixed motor pen plotter because of its low “volume” needed for plotting, modularity, and also the challenge of the kinematics of deriving the motion of the arms.

Hardware Design

In order to make the mechanical design, we wanted to require as little machining as possible and also avoid andy issues with alignment. Thus, we tried using mostly stock parts and also a laser cutter. The use of stock parts makes this project easily reproducible for others. Figure 3 below shows our final CAD model of our design compared to our final actual product.



(a) CAD model rendering

(b) Final product

Figure 3. Comparison between CAD model and physical final product.

The detailed drawings for all the manufactured parts, laser templates, stock parts, and assembly are attached in Appendix A. All files are available upon request.

In the end, our final design had the following components, shown by our bill of materials, Table 2.

The difference between the total and the purchase price is how much we actually spent on all the components for this project. Many of the components were found, bought discounted, or recycled from scrap bins. Most of all, a special thanks to the Cal Poly Robotics club for providing the motors at a third of the market price along with the Aluminum mounting hubs. This list does not include some miscellaneous nuts and washers we had to use as well as the fasteners that came included with some of the parts. Furthermore, the acrylic sheet was not full used. Many of these components bought could be used for other purposes because more than the needed number come in a single purchase.

Table 2. Bill of materials

#	**Part Name**	**Price**	**Qty.**	**Total**	**Distributor No.**
1	Motor and Encoder	$39.95	2	$79.90	Pololu: 2824
2	Aluminum Mounting Hubs	$7.95	4	$31.80	Pololu: 1083
3	Aluminum Machined Brackets	$7.95	2	$15.90	Pololu: 1995
4	Clear Cast Acrylic Sheet	$28.65	1	$28.65	McMaster-Carr: 8560K593
5	T5 Timing Belt	$6.07	1	$6.07	McMaster-Carr: 1679K544
6	T5 Timing Pulleys	$9.80	2	$19.60	McMaster-Carr: 1428N24
7	18-8 SS Screws M3 X 0.5, 8 mm long	$4.12	1	$4.12	McMaster-Carr: 91292A112
8	18-8 SS Screws M3 X 0.5, 14 mm long	$5.77	1	$5.77	McMaster-Carr: 91292A027
9	MDF Wood Sheet	$10.95	1	$10.95	Home Depot Model # 1508108
10	Reinforcing Brackets	$2.92	2	$5.84	McMaster-Carr: 1088A32
11	Sheet Metal Screws	$3.04	1	$3.04	McMaster-Carr: 90048A192
12	A2 Tool Steel Rod	$5.47	1	$5.47	McMaster-Carr: 8116K35
13	Set Screw Shaft Coupling	$2.93	2	$5.86	McMaster-Carr: 5395T111
14	HS-65MG Servo	$29.49	1	$29.49	ServoCity: 32065S
15	Stud-Mount Ball Roller	$4.00	1	$4.00	McMaster-Carr: 6460K31
16	4-40 SS Screw, 3/4" long	$3.07	1	$3.07	McMaster-Carr: 90604A555
			Total	$259.53	
			Purchase	**$103.58**	

Manufacturing

There were only 8 parts that had to be manufactured for this project. The base plate, motor plate, arm 1, arm 2, a couple rods (simply cut to length), a threaded shaft coupler/holder, sheet metal servo mount, and a pen holder. All the manufacturing was done in the Cal Poly Mechanical Engineering Mustang 60 Machine Shop.

The base plate and motor plate were both made with the MDF wood. A laser cutter/engraver shown in Figure 4 was used to etch the layout to position the through holes needed for screws as well as the slots for mounting the motors. After the etching, the holes were drilled and the slots were cut out using a forstner bit and then roughly cut out using a skill/scroll saw. Precision was not needed for the slots because they are large openings mainly for access or clearance for wiring. The holes did require a little precision but not much because they were made with tolerances in mind.

Figure 4. Universal Laser Systems laser cutter/engraver. This machine requires files to be in .dwg or .ai. The components must be scaled to be 1:1 in order to be printed correctly. This particular machine has a print area of 32” x 18”. These could have been used to drill and make the holes if multiple passes were done.

The same laser cutter was also used to cut out the arms of the pen plotter. Both wood and acrylic test pieces were made. An example of the laser cut arms are shown in Figure 5. Multiple were made to test different arm lengths and to optimize the center to center distance between the motor and the rod for arm 2. Different arm 1 lengths would give different belt tensions in the belt to control arm 2. For our final product, the 207 mm C-C arm 1 resulted in the best belt tension. It is important to note that this adjustment had to be done because we had ordered the wrong belt

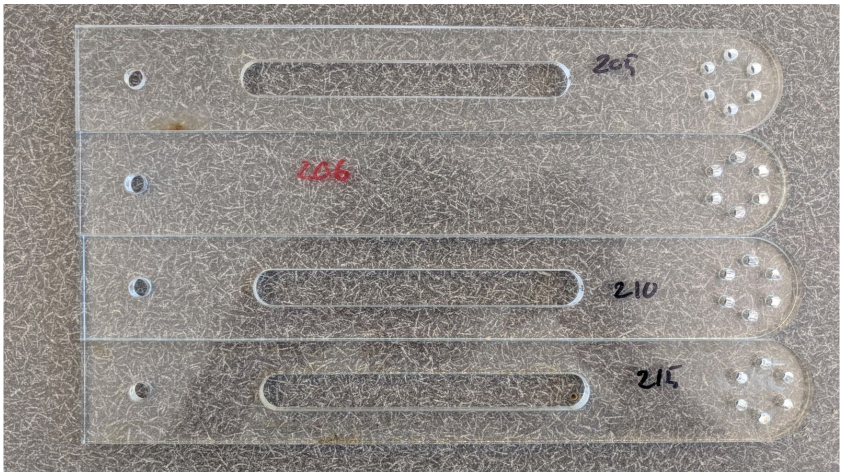


Figure 5. Arm 1 test pieces of different lengths and types. Stiffness was a concern for our idea and so we slightly played around with the idea of having slots to save weight but this would also come with the cost of some stiffness. For larger arms this may become an issue, but for our scale, only 8.5” x 11”, we learned that neither stiffness of the arm was not an issue with the thickness of acrylic used. Also, we made a ball roller shaft holder to alleviate any moment arms and weight.

A couple rods were also made simply using stock rods that are the same diameter of the output shafts of the motors we had. This made it easy for us to couple with the output shaft of the motors. The rods were cut to length using a horizontal band saw and then the ends were simply grinded down to have a small chamfer. A couple cut small rods were then press fit into the timing pulleys. We did not want to permanently press fit the pulleys to the motor so a shaft coupler was used.

The next manufactured piece was an aluminum shaft coupler shown below in Figure 6. This coupler interfaced a ball roller with the end of the arm 2 pulley shaft. One end of the coupler was thread while the other was a simply a drilled clearance hole for the shaft to sit in.

Figure 6. A close image of arm two shaft rod. From top to bottom, the components are a timing pulley, a hub mount, arm 1, two hub mounts, arm 2, shaft coupler, and ball roller element. The shaft coupler could be raised and lowered by using the threads to help balance both arms.

(a) Southbend MicroLathe

(b) Tools used

Figure 7. These tools were used to make the aluminum shaft coupler. Most operations were done on the lathe (a). The tools used from left to right in (b) are a tap handle, tapping fluid, ¼-20 bottoming tap, number 7 standard drill bit, 6 mm drill bit, center drill, facing tool, turning tool, parting tool, and chuck.

The servo mount was made by first getting the dimension of the servo. We could not find any drawings or CAD models for the particular model of servo we had and so by using an MicroVu optical CMM, coordinate measuring machine, we able to get the dimensions of the holes as well as other important dimensions of the servo which we used to make the servo mount. The output of the file are in Appendix A along with the drawings.

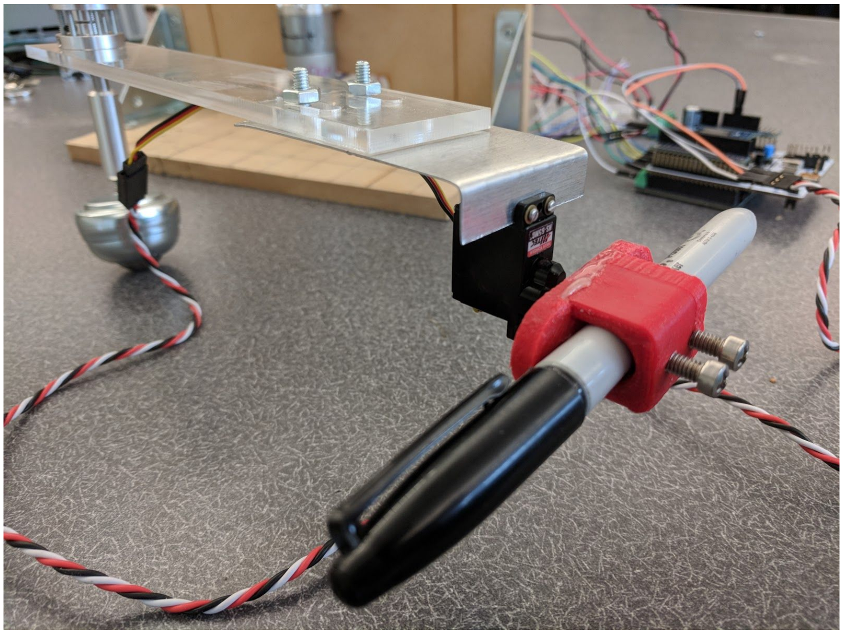


Figure 8. A close image of the pen holder, servo, and servo mount attached to arm 2. Arm 2 is slotted to make it easy to adjust how far out the servo and a result the drawing point also moves out farther.

The last part manufactured was the pen holder. This was made using a 3D printer, specifically a FlashForge 3D Printer Creator Pro.

Assembly

For the assembly of the entire product, see the drawings in Appendix A as well as the assembly video that can be found on Youtube here, https://youtu.be/rqMjX1NTY1M.

Electrical Wiring

The following section shows how we wired our microcontroller to each of our components. A collection of male/male jumper wires were used with a breadboard to initially wire the microcontroller to the motors, encoders and servo as shown in Figure 9. Ideally, if we had more time and a finalized product, all of the jumper wire connections would be replaced with wires soldered to a perfboard. The wiring was kept as organized and consistent as possible by color coding the jumper wires to the motor wires and being aware of spacing out groups of wires coming from each motor and servo for easy visual recognition of each connection to the microcontroller.

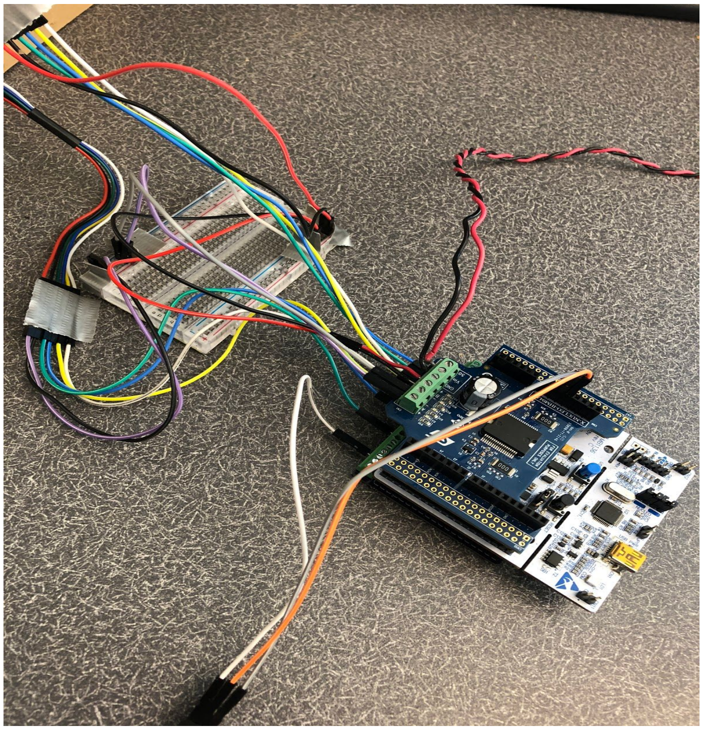


Figure 9. Microcontroller wiring apparatus during the final stages of testing our coaxial 2DOF pen plotter. A breadboard was initially used to wire the motors and encoders to the Shoe of Brian purple micropython board. Additionally, the power supply and servo pins are connected to the blue Nucleo IHM04A1 motor driver board that is pin connected on top of the white Nucleo L476RG.

Two seperate 50:1, 37Dx70L mm metal gearmotors with 64 CPR encoders were used and wired to our microcontroller. Table 4 references the function of each color wire from the motor. The red and black power and ground wires were connected to a blue Nucleo IHM04A1 motor expansion board that is pin connected on top of the white Nucleo L476RG microcontroller. The remaining encoders were connected to the purple Shoe of Brain micropython board connected to the bottom of the other two boards. Table 3 and Figure 10 reference the physical pin labels on the Shoe of Brain to the CPU pins names when instantiating Encoder 1, 2, and the servo in micropython. Additionally, the table shows the specific timer channels required for each pin connection which were referenced from the STM32L476 datasheet [6]. Timers 3 and 5 are not used for the encoder nor servo since these are needed for the motor PWM. Thus, we found from Table 17 on the STM32L476 datasheet we could use timers 4 and 8 for each encoder.

Table 3. Encoder and Servo pin connections on Shoe of Brain purple micropython board [6].

Component	Pin	CPU Pin	Timer	Ch.
Encoder 1	C6	PC6	TIM8	1
	C7	PC7	TIM8	2
Encoder 2	B6	PB6	TIM4	1
	B7	PB7	TIM4	2
Servo	A5	PA5	TIM2	1

Table 4. 50:1 Metal gearmotor 37Dx70L mm with 64 CPR Encoder wire labels [7].

Red	motor power (connects to one motor terminal)
Black	motor power (connects to the other motor terminal)
Green	encoder GND
Blue	encoder Vcc (3.5 – 20 V)
Yellow	encoder A output
White	encoder B output

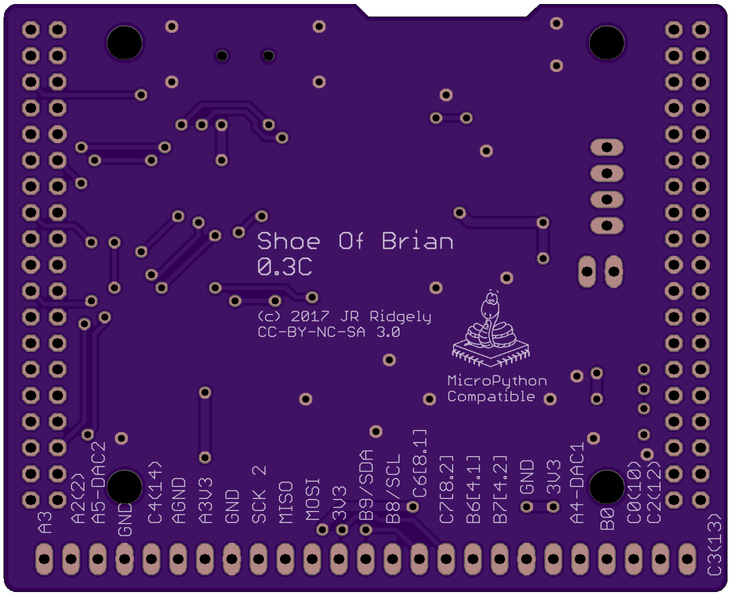


Figure 10. Custom Shoe of Brian micropython board: 2 layer board of 2.70 x 2.25 inches (68.6 x 57.1 mm) designed by Professor John Ridgely.

All the motor encoder wires were connected to the Shoe of Brain screw terminals with male/male jumper wires, as pictured in Figure 11. Since the servo only needs a small voltage to run, the servo leads are connected to the 5V and GND pin terminals on the Nucleo L476RG microcontroller through the blue Nucleo IHM04A1 motor driver board. The gearmotors consisted of 12V brushed DC motors with a 50:1 metal gearbox and an embedded quadrature encoder that provides a resolution of 64 counts per revolution of the motor shaft, corresponding to 3200 counts per revolution of the gearbox's 16 mm-long, 6 mm-diameter D-shaped output shaft. The encoder uses a two-channel Hall effect to sense the rotation of a magnetic disk on a rear protrusion of the motor shaft [7].

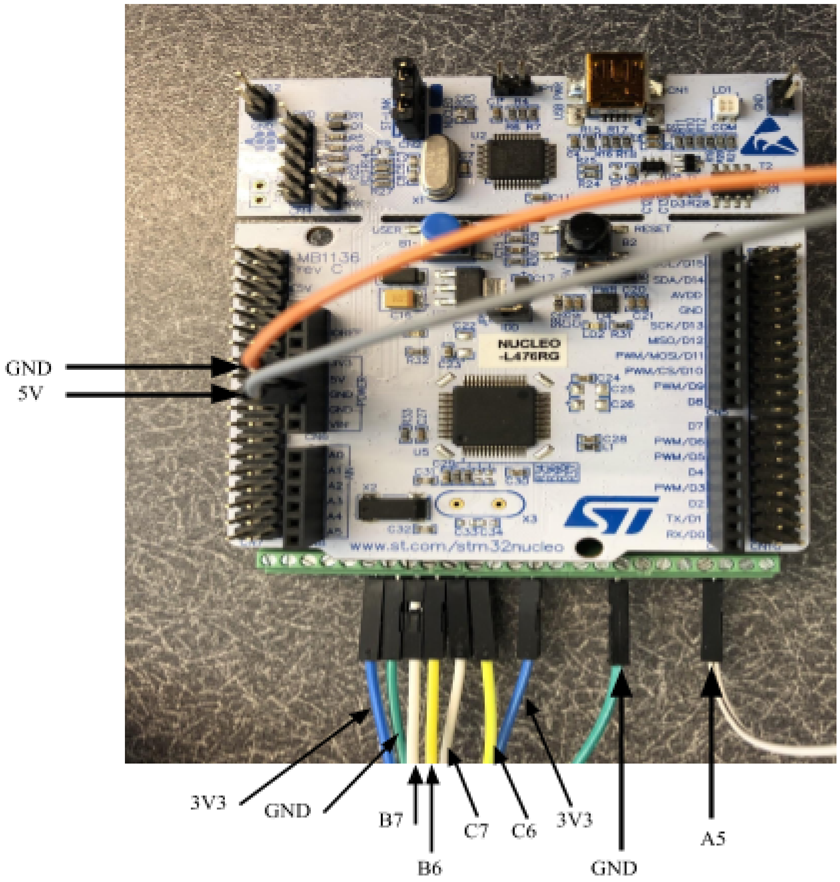


Figure 11. Shoe of Brian and Nucleo L476RG microcontroller wiring apparatus during the final stages of testing our coaxial 2DOF pen plotter for both motors and encoders. Note the servo GND and 5V wires would be physically connected to the Nucleo IHM04A1 which is not shown in the picture.

To program a MotorDriver class, a mini USB cable is connected to the bottom Shoe of Brian MicroPython board and our Pololu motors are connected to the Motor A or B screw terminals on the Nucleo driver board. The ST Microelectronics L6206 dual H-bridge motor driver chip datasheet was referenced when initializing instances of each motor in our code. The link to the data sheet can be found on page 2, Figure 2 of the datasheet in the References section [5]. As can be seen from Figure 12, the motor is connected to pins OUT1A and OUT2A and physically wired to L6206 pins IN1A and IN2A on the motor driver board. The microcontroller controls pins ENA, IN1A, and IN2A. For our MotorDriver class, pin ENA is set to high to enable the motor, IN1A is set low, and a PWM signal is sent to IN2A by setting it high, to power the motor in one direction. To control a pin on the Nucleo board for powering the motor, the truth table in the L6206 datasheet was first referenced in Table 5 to find the connections between the logic input pins on the L6206 and the CPU pins on the Nucleo motor control expansion board. The INx pins were set up as regular push-pull outputs, but needed to be configured with af = 2. This chosen alternate function was referenced from the STM32L476 datasheet on page 88, Table 17 [6].

Table 5. Connections between logic pins on L6206 and CPU pins on Nucleo IHM04A1 board [5].

L6206 Pin	CPU Pin	Timer	Ch.
ENA/OCDA	PA10	-	-
IN1A	PB4	TIM3	1
IN2A	PB5	TIM3	2
ENB/OCDb	PA10	-	-
IN1B	PA0	TIM5	1
IN2B	PA1	TIM5	2

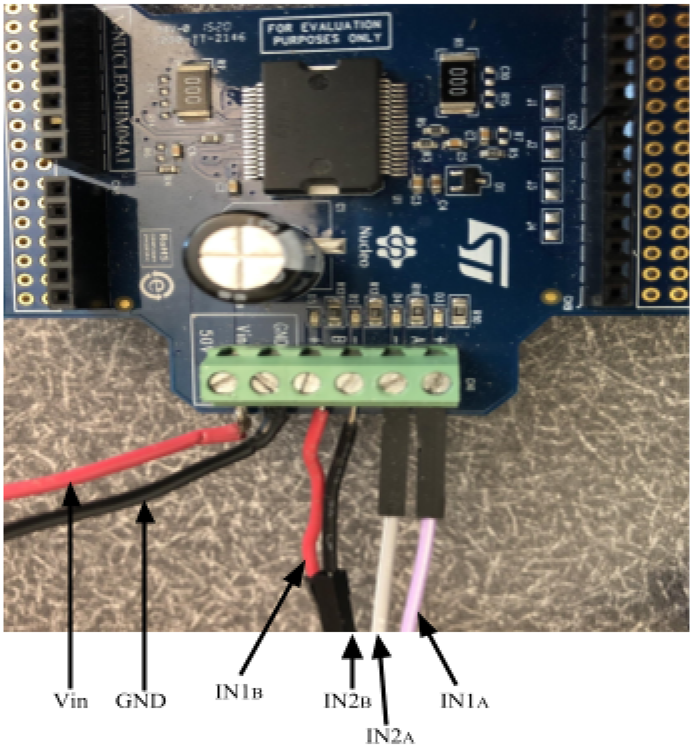


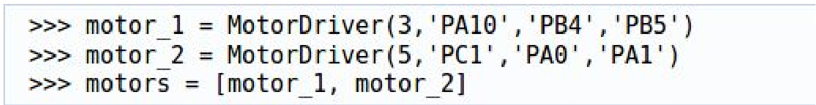
Figure 12. Blue Nucleo IHM04A1 motor driver board wiring apparatus during the final stages of testing our coaxial 2DOF pen plotter for power supply terminals and motor channels..

Software Design

The following section describes how the code was developed and utilized during the project. The state and task diagrams used to develop the code can be seen in Appendix B, and the doxygen on the actual code itself is in Appendix C.

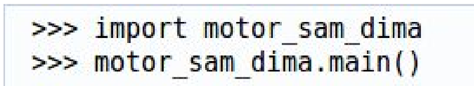
Motor Driver

The Pololu DC brushed motors are powered by 12 volts and a 3A current limit by connecting power from a benchtop supply to the motor driver board with the Gnd and Vin screw terminals. Two MotorDriver class instances on our motor_task.py were created for each motor with their corresponding timer channels and CPU pins similar to the example below in Figure 13. In order to test the MotorDriver class in motor_sam_dima.py, the following can be typed on a Micro-python terminal like Putty with a main function created in motor_sam_dima.py to test the program. The test code is written to exercise the motor driver and test for any bugs in the code. It needs to be an if_name_=='_main_' block to allow the user the test the motor from an REPL as shown in Figure 14. Note that the motor_sam_dima.py file must be imported first to operate the program from the REPL using the class MotorDriver.



```
>>> motor_1 = MotorDriver(3,'PA10','PB4','PB5')
>>> motor_2 = MotorDriver(5,'PC1','PA0','PA1')
>>> motors = [motor_1, motor_2]
```

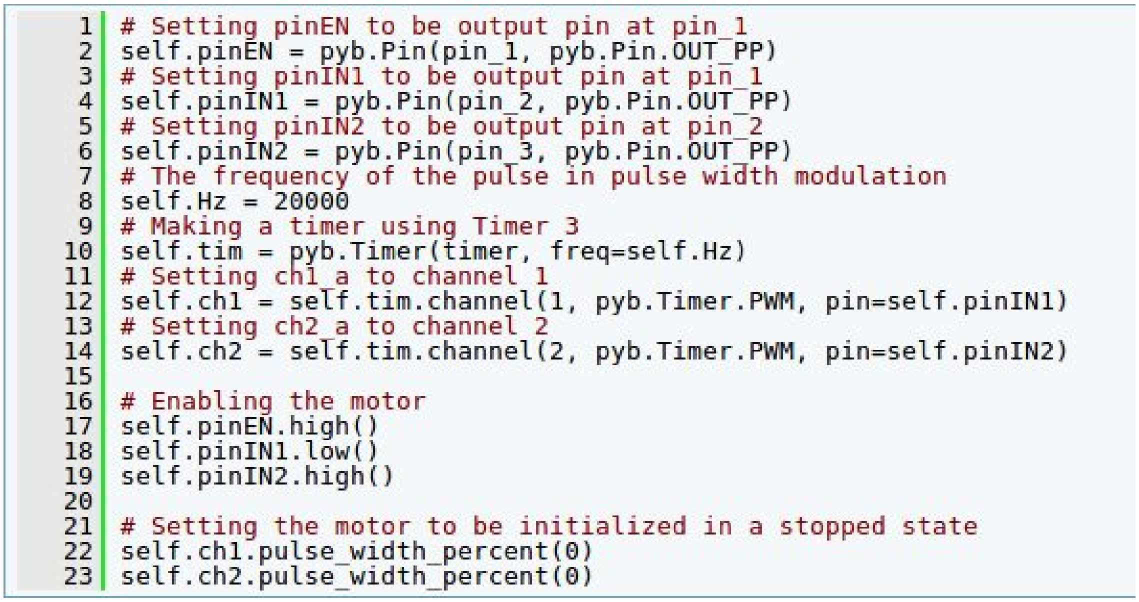
Figure 13. An example of MotorDriver above called motor_1 on Timer 3 and connected to the Nucleo board in pins B4 and B5. Additionally, you have a second MotorDriver called motor_2 on Timer 5 and connected to the board on pins A0 and A1.



```
>>> import motor_sam_dima
>>> motor_sam_dima.main()
```

Figure 14. REPL command code to test our MotorDriver class code written in the motor_sam_dima.py python file.

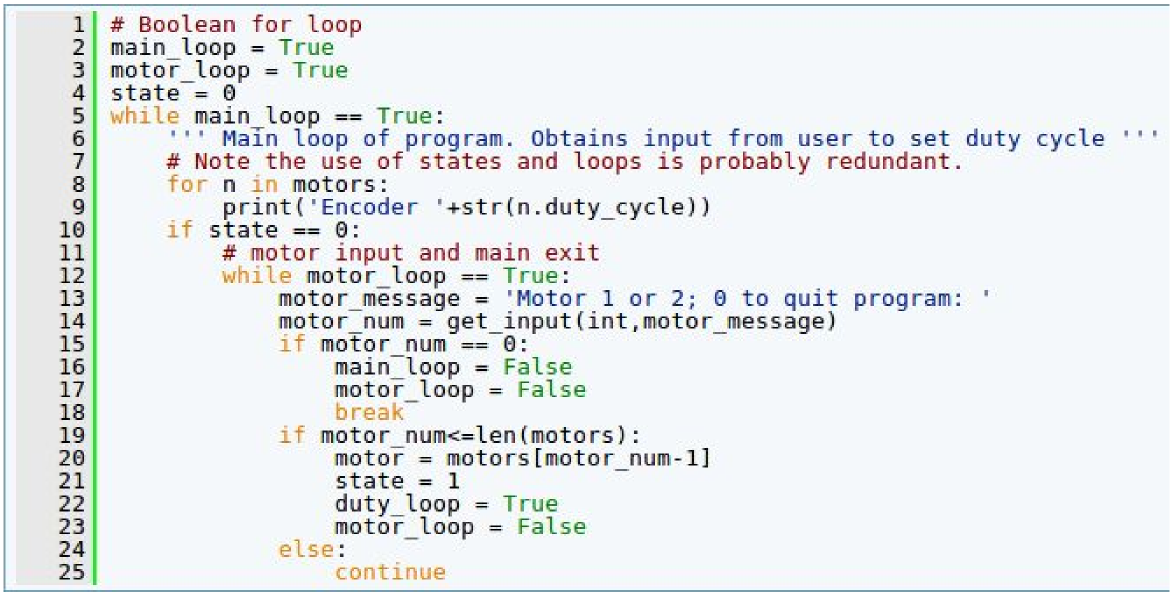
Two functions were written for the MotorDriver class: get_duty_cycle and set_duty_cycle. By calling these function in our main function test code, the user can input an integer from -100 to 100 to control the direction of the motor. A positive integer will spin the motor in one direction, negative integers will spin it in the opposite direction and a value of 0 will not spin the motor. The speed can be controlled by inputting a signed integer for parameter 'level' in the set_duty_cycle function which holds the PWM duty cycle of the voltage sent to power the motor. Figure 15 shows the code that initializes MotorDriver. Timer is a parameter that gets initialized with a chosen timer. Pin_1 parameter is initialized to enable the motor. Pin_2 and Pin_3 are parameters for the second and third pins for IN1 direction 1 and IN2 direction 2 in order to power the motor in one direction or the other. Refer to Mercurial for more details on the functions written in the motor_sam_dima.py file.



```
# Setting pinEN to be output pin at pin_1
self.pinEN = pyb.Pin(pin_1, pyb.Pin.OUT_PP)
# Setting pinIN1 to be output pin at pin_1
self.pinIN1 = pyb.Pin(pin_2, pyb.Pin.OUT_PP)
# Setting pinIN2 to be output pin at pin_2
self.pinIN2 = pyb.Pin(pin_3, pyb.Pin.OUT_PP)
# The frequency of the pulse in pulse width modulation
self.Hz = 20000
# Making a timer using Timer 3
self.tim = pyb.Timer(timer, freq=self.Hz)
# Setting ch1_a to channel 1
self.ch1 = self.tim.channel(1, pyb.Timer.PWM, pin=self.pinIN1)
# Setting ch2_a to channel 2
self.ch2 = self.tim.channel(2, pyb.Timer.PWM, pin=self.pinIN2)

# Enabling the motor
self.pinEN.high()
self.pinIN1.low()
self.pinIN2.high()

# Setting the motor to be initialized in a stopped state
self.ch1.pulse_width_percent(0)
self.ch2.pulse_width_percent(0)
```

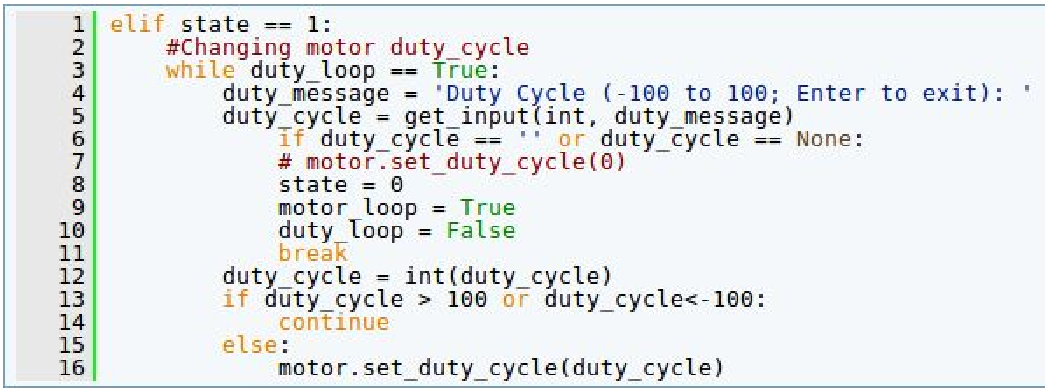
Figure 15. Initialization of class MotorDriver.

The following figures contain sections of code written in a main function for motor_sam_dima.py in order to test the functionality of our motors. A simple user interface was created on the REPL so that the user can follow a series of steps to control the duty cycle sent to the each motor separately. Figure 16 contains a while loop for the user to be able to select motor 1 or 2 to drive from the REPL. Figure 17 tests for the correct duty cycle entered on the REPL after a motor number was selected. The code checks to make sure an integer between -100 and 100 for the duty cycle was entered to ensure the correct PWM is sent to the motor. Figure 18 is a function created in our main file which tests for correct user input. Essentially if the user types in the wrong input type, such as a float instead of an integer, than the function will go through a list of types available and return a string to the user saying the incorrect input was typed and which type of input is needed. Lastly, Figure 19 was a custom exception error created to test for incorrect user input.



```
# Boolean for loop
main_loop = True
motor_loop = True
state = 0
while main_loop == True:
    ''' Main loop of program. Obtains input from user to set duty cycle '''
    # Note the use of states and loops is probably redundant.
    for n in motors:
        print('Encoder '+str(n.duty_cycle))
    if state == 0:
        # motor input and main exit
        while motor_loop == True:
            motor_message = 'Motor 1 or 2; 0 to quit program: '
            motor_num = get_input(int,motor_message)
            if motor_num == 0:
                main_loop = False
                motor_loop = False
                break
            if motor_num<=len(motors):
                motor = motors[motor_num-1]
                state = 1
                duty_loop = True
                motor_loop = False
            else:
                continue
```

Figure 16. Testing to see if user selects motor 1 or 2.

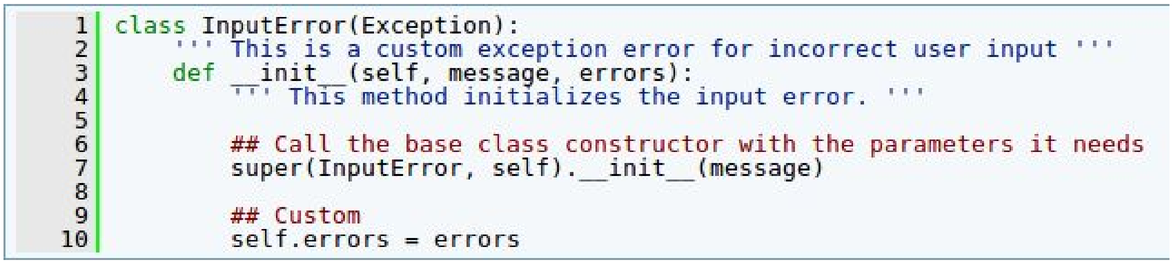


```
elif state == 1:
    #Changing motor duty_cycle
    while duty_loop == True:
        duty_message = 'Duty Cycle (-100 to 100; Enter to exit): '
        duty_cycle = get_input(int, duty_message)
            if duty_cycle == '' or duty_cycle == None:
            # motor.set_duty_cycle(0)
            state = 0
            motor_loop = True
            duty_loop = False
            break
        duty_cycle = int(duty_cycle)
        if duty_cycle > 100 or duty_cycle<-100:
            continue
        else:
            motor.set_duty_cycle(duty_cycle)
```

Figure 17. Testing correct user input duty cycle.

```
def get_input(type_needed, message):

    # List of types
    list_of_types = [list, str, int, float]
    if type_needed in list_of_types:
        # If the type needed is in the list of types, get the correct index
        type_index = list_of_types.index(type_needed)
    else:
        # If the type is not in list of types
        print('Type not found')
        return
    input_loop = True
    while input_loop == True:
        try:
            user_input = input(str(message))
            if user_input == '' or None:
                return None
            user_input = list_of_types[type_index](user_input)
            input_loop = False
        except:
            print('Incorrect input type. The needed input type is: '
            +str(type_needed)+'. Please try again.')
    return user_input
```

Figure 18. Testing for correct user input type.

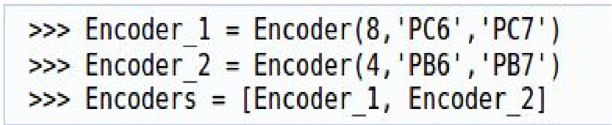


```
class InputError(Exception):
    ''' This is a custom exception error for incorrect user input '''
    def __init__(self, message, errors):
        ''' This method initializes the input error. '''

        ## Call the base class constructor with the parameters it needs
        super(InputError, self).__init__(message)

        ## Custom
        self.errors = errors
```

Figure 19. Creating a custom exception error for incorrect user input.

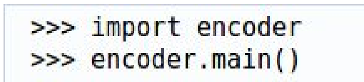
Encoder

Another essential section of code our pen plotter runs with is an encoder.py file which implements a quadrature encoder embedded in the 50:1 DC brushed motor for the Shoe of Brian purple MicroPython board. The encoder power leads were connected to the GND and 3V3 leads on the purple Shoe of Brian MicroPython board screw terminals. Next the encoder was connected to pins B6 and B7 of the Shoey board's 25-wire screw terminal block. A second encoder was also connected to pins C6 and C7 on the same screw terminal block. By connecting the encoders to these pins, a separate timer is set up to read an encoder using these pins. Timer 3 and 5 are not used for the encoder since these are needed for the motor PWM. Thus, we found from Table 17 on the STM32L476 datasheet we could use timers 4 and 8 for each of each encoder below. Similarly to the MotorDriver class, Figure 20 shows two class instances created on our motor_task.py file, specifying the timer channels and CPU pins assigned to each encoder. Additionally, to test our Encoder class properly, the commands shown in Figure 21 were typed from an REPL to run the main function inside our encoder.py file.



```
>>> Encoder_1 = Encoder(8,'PC6','PC7')
>>> Encoder_2 = Encoder(4,'PB6','PB7')
>>> Encoders = [Encoder_1, Encoder_2]
```

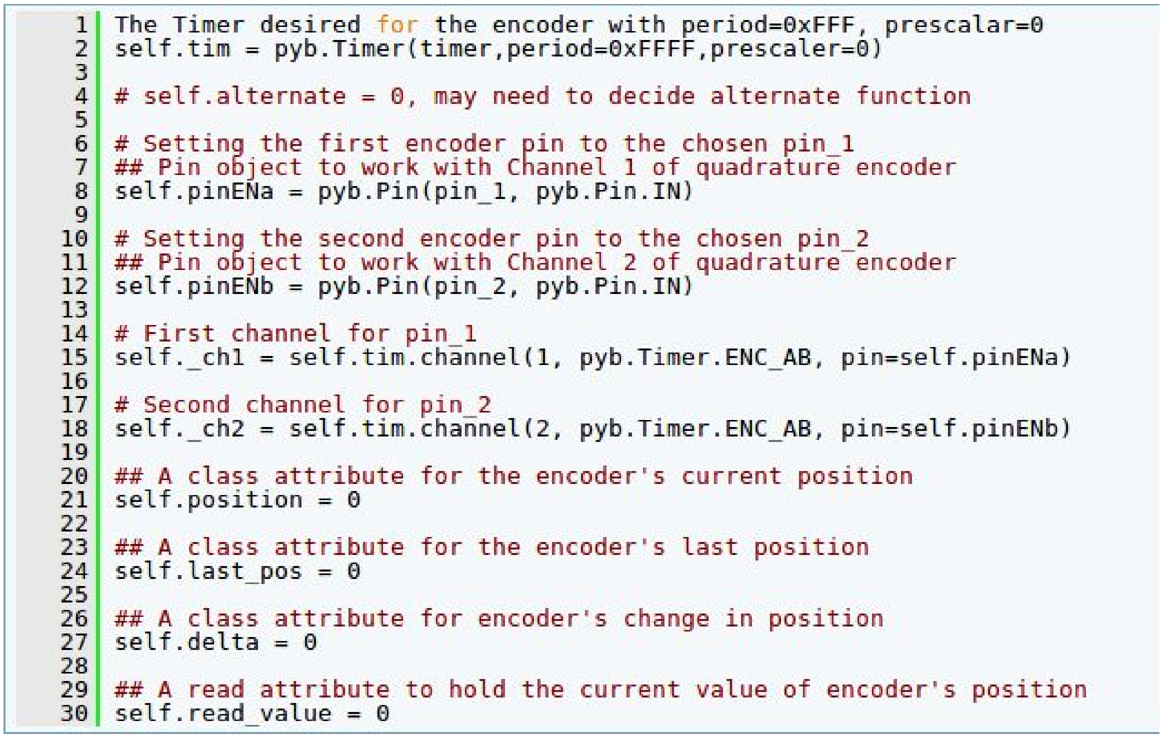
Figure 20. An example of Encoder called Encoder_1 on Timer 8 and connected to the board in pins C6 and C7. Additionally, you have a second Encoder called Encoder_2 on Timer 4 and connected to the board on pins B6 and B7.



```
>>> import encoder
>>> encoder.main()
```

Figure 21. REPL command code to test our Encoder class code written in the encoder.py python file. Test code is written in a main function to exercise the encoders and test for any bugs in the code. It needs to be an if_name_=='_main_' block to allow the user the test the encoders from an REPL. Note that you must also import the encoder.py file to operate the encoder from the REPL using the encoder class Encoder.

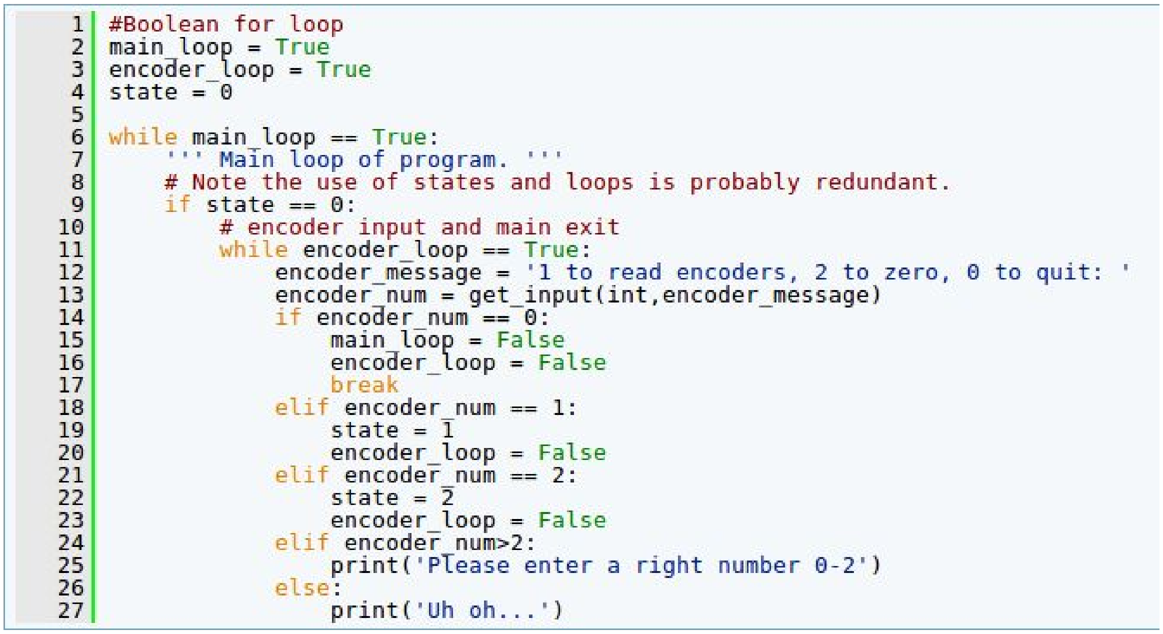
The following code in Figure 22 initializes Encoder. The timer parameter is the Timer the user wants to use. Pin_1 is the first pin on the board for encoder Ch A. Pin_2 is the second pin on the board for encoder Ch B. The encoder.py file also included the following function: read and zero. The read function was a method for returning the current position of the encoder and the zero function was a method for resetting the position of the encoder to zero. Refer to Mercurial for more details on the functions written in the encoder.py file.



```
The Timer desired for the encoder with period=0xFFF, prescalar=0
self.tim = pyb.Timer(timer,period=0xFFFF,prescaler=0)

# self.alternate = 0, may need to decide alternate function

# Setting the first encoder pin to the chosen pin_1
## Pin object to work with Channel 1 of quadrature encoder
self.pinENa = pyb.Pin(pin_1, pyb.Pin.IN)

# Setting the second encoder pin to the chosen pin_2
## Pin object to work with Channel 2 of quadrature encoder
self.pinENb = pyb.Pin(pin_2, pyb.Pin.IN)

# First channel for pin_1
self._ch1 = self.tim.channel(1, pyb.Timer.ENC_AB, pin=self.pinENa)

# Second channel for pin_2
self._ch2 = self.tim.channel(2, pyb.Timer.ENC_AB, pin=self.pinENb)

## A class attribute for the encoder's current position
self.position = 0

## A class attribute for the encoder's last position
self.last_pos = 0

## A class attribute for encoder's change in position
self.delta = 0

## A read attribute to hold the current value of encoder's position
self.read_value = 0
```

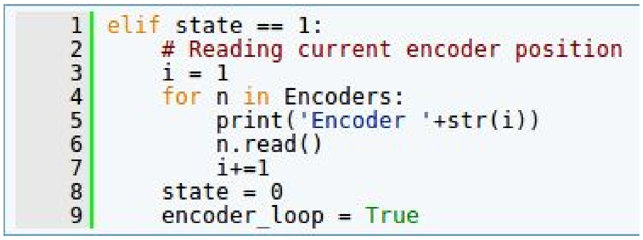
Figure 22. Initialization of class Encoder.

The following figures contain sections of code written in a main function for encoder.py in order to test the functionality of the encoders. A simple user interface was created on the REPL so that the user can follow a series of steps to see if the encoders were reading a correct position. Figure 23 contains a while loop which allows the user to initially type a 1 for reading the position of both encoders or a 2 to zero both encoders. Figure 24 is a state which runs the command to read and print the current position of both encoders from the REPL. Figure 25 is another state which zeros the position of each encoder.



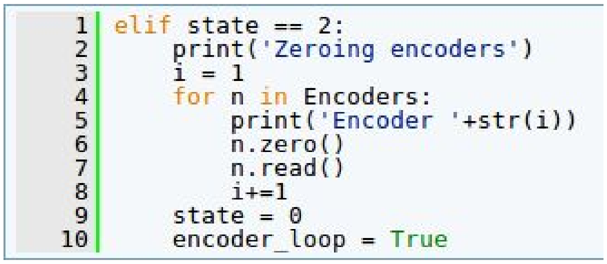
```
#Boolean for loop
main_loop = True
encoder_loop = True
state = 0

while main_loop == True:
    ''' Main loop of program. '''
    # Note the use of states and loops is probably redundant.
    if state == 0:
        # encoder input and main exit
        while encoder_loop == True:
            encoder_message = '1 to read encoders, 2 to zero, 0 to quit: '
            encoder_num = get_input(int,encoder_message)
            if encoder_num == 0:
                main_loop = False
                encoder_loop = False
                break
            elif encoder_num == 1:
                state = 1
                encoder_loop = False
            elif encoder_num == 2:
                state = 2
                encoder_loop = False
            elif encoder_num>2:
                print('Please enter a right number 0-2')
            else:
                print('Uh oh...')
```

Figure 23. Testing for encoder input and main exit.



```
elif state == 1:
    # Reading current encoder position
    i = 1
    for n in Encoders:
        print('Encoder '+str(i))
        n.read()
        i+=1
    state = 0
    encoder_loop = True
```

Figure 24. Testing to see if both encoders read position.

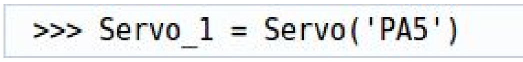


```
elif state == 2:
    print('Zeroing encoders')
    i = 1
    for n in Encoders:
        print('Encoder '+str(i))
        n.zero()
        n.read()
        i+=1
    state = 0
    encoder_loop = True
```

Figure 25. Testing to see if both encoders are zeroed.

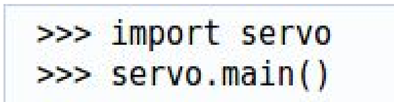
Servo

The servo.py python file was written to implement a Servo class which has methods to control the position of a servo by calculating and setting the duty cycle of the servo. The servo is used to control the up and down pen motion movements when drawing. See the following example in Figure 26 and 27 for creating an instance of class Servo, as well as the commands to run the test code in a main function written in the servo.py file.



```
>>> Servo_1 = Servo('PA5')
```

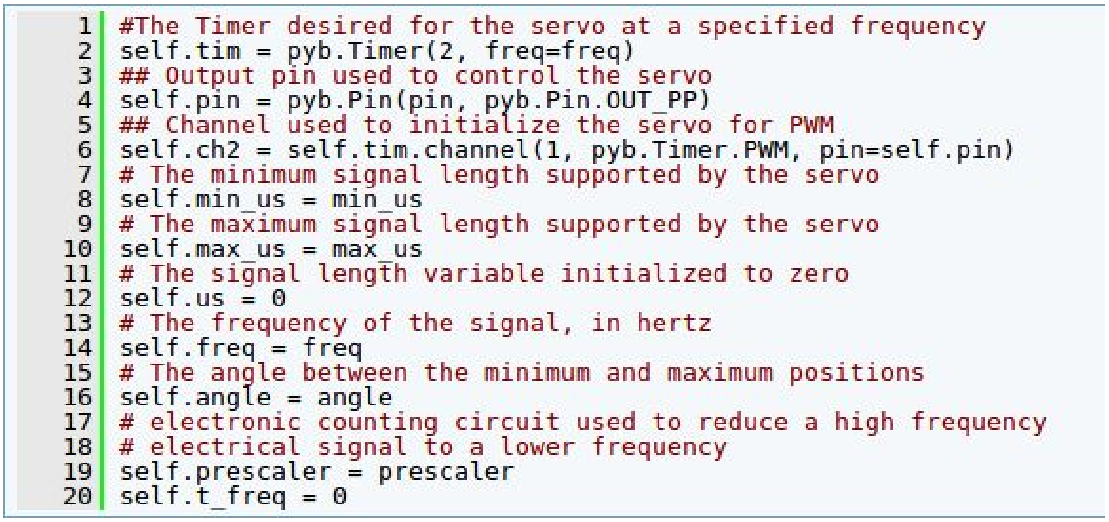
Figure 26. An example of Servo called Servo_1 on Timer 2 and connected to pin A5 on the Shoe of Brian board. A class was written to initialize and control the position for the HS-65MG used on our Pen plotter. A portion of our code was used from code found online, which can be accessed in the References[8].



```
>>> import servo
>>> servo.main()
```

Figure 27. Test code is written in a main function to exercise the servo and test for any bugs in the code. It needs to be an if_name=='main'_ block to allow the user to test the servos from an REPL. Note that you must also import the servo file to operate the motor from the REPL using the servo class Servo.

Additionally, the datasheet was referenced for the HS-65MG servo to set the specific frequency and microsecond range of the servo for the initialization. In order to properly initialize the servo, the pin where the servo is connected must be specified to support PWM, the frequency must be set, the minimum and maximum signal length supported by the servo must be specified, and the angle between the minimum and maximum positions must be set. Refer to the following code which properly initializes our servo in Figure 28. Refer to Mercurial for more details on the functions written in the servo.py file.

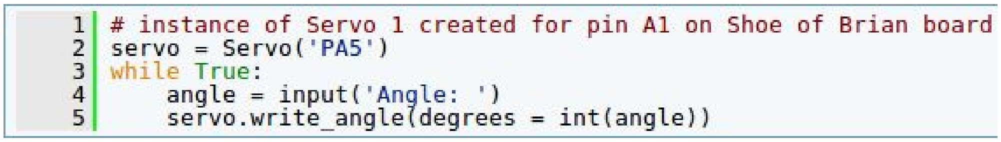


```
#The Timer desired for the servo at a specified frequency
self.tim = pyb.Timer(2, freq=freq)
## Output pin used to control the servo
self.pin = pyb.Pin(pin, pyb.Pin.OUT_PP)
## Channel used to initialize the servo for PWM
self.ch2 = self.tim.channel(1, pyb.Timer.PWM, pin=self.pin)
# The minimum signal length supported by the servo
self.min_us = min_us
# The maximum signal length supported by the servo
self.max_us = max_us
# The signal length variable initialized to zero
self.us = 0
# The frequency of the signal, in hertz
self.freq = freq
# The angle between the minimum and maximum positions
self.angle = angle
# electronic counting circuit used to reduce a high frequency
# electrical signal to a lower frequency
self.prescaler = prescaler
self.t_freq = 0
```

Figure 28. Initialization of class Servo.

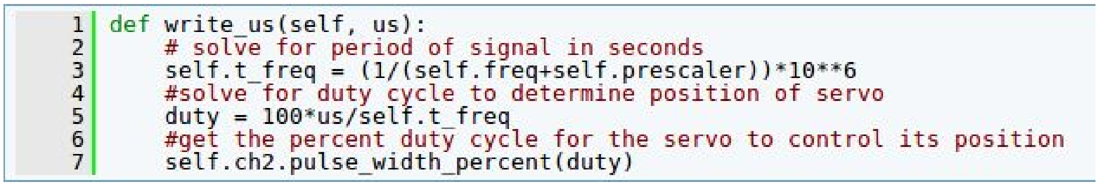
The following figures contain sections of code written in a main function for servo.py in order to test the functionality of the servo A simple user interface was created on the REPL so that the user can follow a series of steps to see if the servo is moving to the correct position. Figure 29 allows the user to input an angle from 0 to 180 degrees for the servo to move to. Figure 30 contains the function *write_us* which solves for the period of the signal in seconds and duty cycle to send to the servo for position control.

Figure 31 solves for a specified angle in degrees or radians and calculates the signal length of the servo to be sent as a parameter to the function *write_us*.



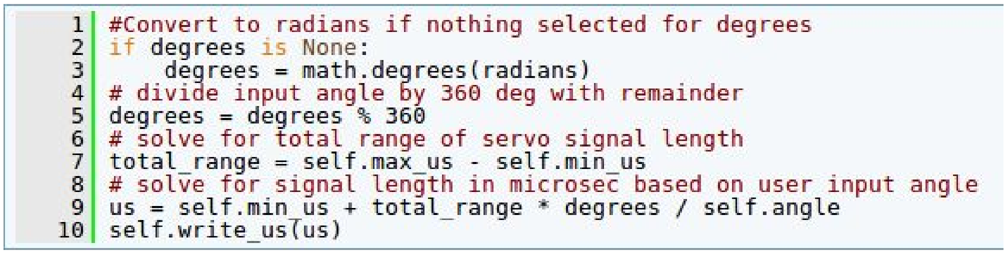
```
# instance of Servo 1 created for pin A1 on Shoe of Brian board
servo = Servo('PA5')
while True:
    angle = input('Angle: ')
    servo.write_angle(degrees = int(angle))
```

Figure 29. Initialization of Servo and user angle input.



```
def write_us(self, us):
    # solve for period of signal in seconds
    self.t_freq = (1/(self.freq+self.prescaler))*10**6
    #solve for duty cycle to determine position of servo
    duty = 100*us/self.t_freq
    #get the percent duty cycle for the servo to control its position
    self.ch2.pulse_width_percent(duty)
```

Figure 30. Function which sets the duty cycle for the servo to control its position.



```
#Convert to radians if nothing selected for degrees
if degrees is None:
    degrees = math.degrees(radians)
# divide input angle by 360 deg with remainder
degrees = degrees % 360
# solve for total range of servo signal length
total_range = self.max_us - self.min_us
# solve for signal length in microsec based on user input angle
us = self.min_us + total_range * degrees / self.angle
self.write_us(us)
```

Figure 31. Code that implements an angle unit conversion and solves for the *write_us* function parameter.

Motor Task

The motor_task python file contained a class which runs a motor task function and initializes two instances of our brushed DC motors and quadrature encoders to be used. After finding an optimal Kp value for our DC motor, we implemented our controller code into a real-time scheduler and created a *motor_task.py* file that can be used to control our controllers' timing with additional tasks being added without harming motor control response. More specifically, our controller task contains a class which contains a motor task function. This task initializes two instances of DC motors and quadrature encoders to be used and has two states to run the motor with the necessary data for finding the motor's position for one state, and another state to run the motor without any data. Additionally, the optimal proportional gain of Kp = 0.05 is set for each motor. Both encoder positions are then zeroed, and the setpoint is set to 3200 encoder ticks for both motors to turn one revolution when testing our motor_task.py file. The motor task was run at a slower and slower rate until the controller's performance noticeable worsened for an optimal timing to run our motors at. Refer to the *Limitations* section for more results on the controller's performance.

```
micropython.alloc_emergency_exception_buf (100)

if __name__ == "__main__":

    print ('\033[2JTesting scheduler in cotask.py\n')

    # Create a share and some queues to test diagnostic printouts
    share0 = task_share.Share ('i', thread_protect = False, name = "Share_0")
    q0 = task_share.Queue ('B', 6, thread_protect = False, overwrite = False,
                          name = "Queue_0")
    q1 = task_share.Queue ('B', 8, thread_protect = False, overwrite = False,
                          name = "Queue_1")

    # A task which prints characters from a queue has automatically been
    # created in print_task.py; it is accessed by print_task.put_bytes()
    # Period of 10 is when it starts to become less smooth.
    motor_periods = [10, 10]
    motor_tasks = []
    for motor_num in range (2):
        print(motor_num)
        newm = motor_task.Motor_control_task(motor_num)
        motor_tasks.append(newm)
        mname = 'Motor_' + str (newm.motor_number)
        cotask.task_list.append(cotask.Task(newm.run_motor, name = mname,
            priority = 3, period = motor_periods[motor_num], profile = True))

    print(str(motor_tasks))

    # Run the memory garbage collector to ensure memory is as defragmented as
    # possible before the real-time scheduler is started
    gc.collect ()

    # Run the scheduler with the chosen scheduling algorithm. Quit if any
    # character is sent through the serial por
    vcp = pyb.USB_VCP ()
    while not vcp.any ():
        cotask.task_list.pri_sched ()

    for n in motor_tasks:
        print(str(n.motor_number))
        n.control.print_response()

    # Empty the comm port buffer of the character(s) just pressed
    vcp.read ()

    # Print a table of task data and a table of shared information data
    print ('\n' + str (cotask.task_list) + '\n')
    print (task_share.show_all ())
    print ('\r\n')
```

Figure 32. Testing motor performance in *main.py* with a real-time scheduler from our *motor_task.py* file.

Our class contains a motor task function and initializes two instances of DC motors and quadrature encoders to be used. The code in Figure 32 calls the print_reponse function from our *controller.py* file to print the time and actual position in the REPL, from which we copy pasted the data to produce the step response plot. Also, the user can enter a specific period for each motor in the *motor_periods* listed created. In the main file, a for loop is run for each motor number, where the period of 10 ms is set for each motor. From this loop, both motors have an instance of the same motor controller task. A period of 10 ms resulted in the slowest rate at which motor performance is not significantly worse. Refer to Mercurial for more details on the functions written in the motor_task.py file.

Controller

Our controller implements two motors to run under PID control. However, when conducting the motor response plots, only the closed-loop proportional control was tested with no integral or derivative action, as well as no load on the motors. The integral and derivative control functions were added in later to fine tune our controller with our final pen plotter product. Our controller generated response plots by getting a setpoint of a desired encoder position, then it iteratively obtained the latest motor position using the encoder and set the duty cycle of the motor based on the error and the proportional gain, K_P. Multiple K_P values were tested to find the quickest and smooth response time. A K_P of 0.05 resulted in the best motor performance as can be seen from Figure 33 below for a setpoint of one revolution or 3200 counts. Figure 34 also illustrates the response of the motor for too low and too high of Kp values.

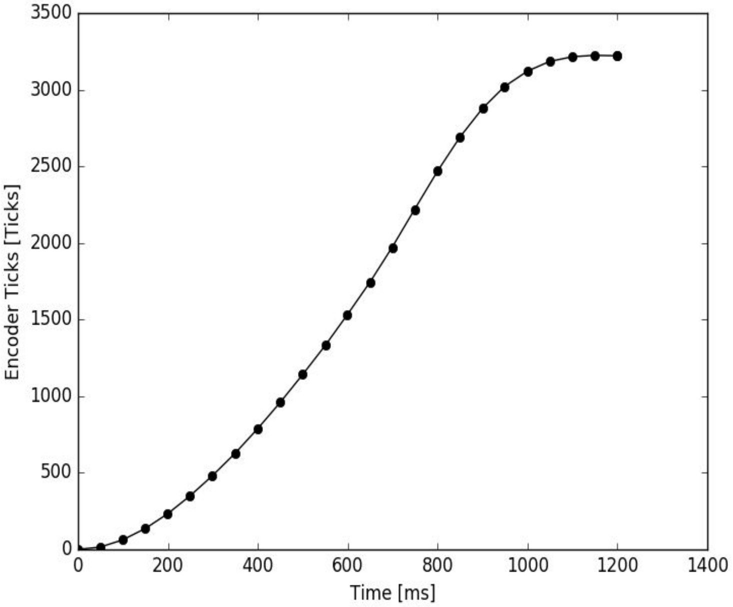
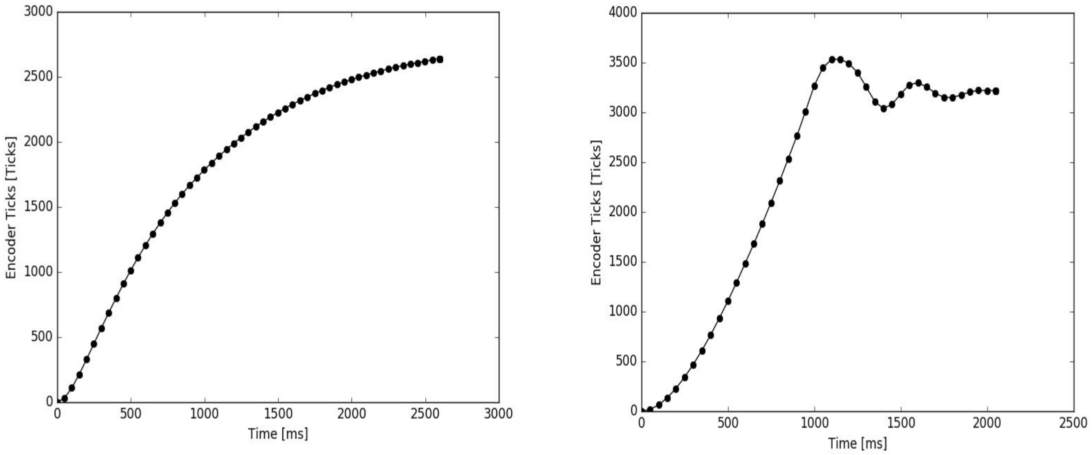


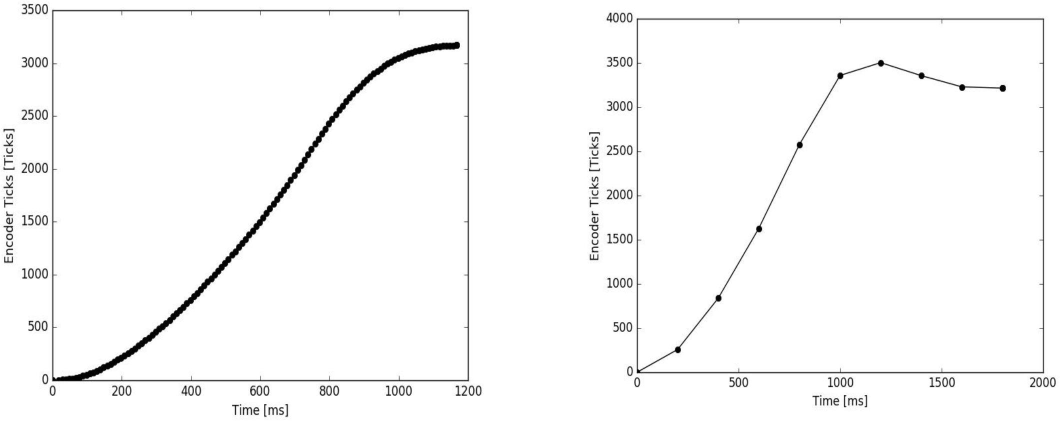
Figure 33. Optimal closed loop time step response plot for $K_P = 0.05$, period = 50 ms, and setpoint of 3200 under no load. The actual position reached around 3150 counts with a percent difference of 1.56%.



(a) Low $K_P = 0.01$ (b) High $K_P = 3$

Figure 34. Motor response plots of a 50:1 Metal Gearmotor 37Dx70L mm with 64 CPR. Period is kept constant and K_P is varied to show the response of the motor for very small and large K_P values at a constant period of 50 ms under no load. The low K_P response never fully reaches the setpoint and thus is not a desirable response. On the other hand, too high of a K_P may reach the setpoint faster than the chosen K_P, but it overshoots and oscillates, another undesirable effect.

Our motor task python file also implements two motors to run under PID control. We have a class that contains a motor task function and initializes two instances of the motors and quadrature encoders to be used. After the motor number instances are initialized, two states are created to run the motor with data for one state, and another state without data. In the main file, a *for* loop is run for each motor number, where the period is set for each motor. From this *for* loop, both motors have an instance of the same motor controller task. A period of 10 ms resulted in the slowest rate at which motor performance is not significantly worse, as can be seen from Figure 35 below. When conducting the motor response plots, only the closed-loop proportional control was tested again, as well as no load on the motors.



(a) Faster Period = 10 ms

(b) Slower period = 200ms

Figure 35. Motor response tests of a 50:1 Metal Gearmotor 37Dx70L mm with 64 CPR. K_P is kept constant at 0.05 and period is varied to show the response of the motor for small and large motor task periods under no load. The slower period begins to oscillate in actual position when the period is increased. On the other hand, although a faster period results in a smoother response, it is not desirable to run at period for which a slower period can still yield sufficient results for proper operation.

Figure 36 contains a portion of test code written as a main function to test our controller class. The main function calls MotorDriver, Encoder, and Controller classes and sets a specific Kp and desired setpoint for the motors to go to, upon which a step response plot was generated after time and position data were printed to the REPL. Refer to Mercurial for more details on the functions written in the controller.py file.

```
def main():
    motor_1 = motor_sam_dima.MotorDriver(5,'PC1','PA0','PA1')
    encoder_1 = encoder.Encoder(8,'PC6','PC7')
    control_1 = controller.Controller()
    control_1.set_gain(0.015)
    control_1.set_setpoint(0)
    encoder_1.zero()
    state = 0
    percent_diff_tolerance = 0.075
    position = 0

    set_point = 4000
    x = True
    y = True
    iterate = 0
    limit = 50
    while x == True:
        if state == 0:
            reference = input()
            if reference == None or reference == '':
                iterate = 0
                state = 1
                control_1.set_setpoint(set_point)
                control_1.act_value = []
                # List of time values for when motor is running
                control_1.time = []
                # List of calculated error signal values
                control_1.error_list = []
                position = 0
                encoder_1.zero()
                y = True
            else:
                print('uh oh')

        elif state == 1:
            while y == True:
                y = nearly_equal(set_point,position,percent_diff_tolerance)
                position = encoder_1.read()
                actuation = control_1.algorithm(position)
                motor_1.set_duty_cycle(actuation)
                utime.sleep_ms(10)
                iterate += 1
                if iterate>limit:
                    y = False
            motor_1.set_duty_cycle(0)
            control_1.print_response()
            state = 0
    print('Done')
```

Figure 36. Main function test code that runs from PC python file *response.py* to test and plot a motor step response plot for a given Kp and setpoint.

List of Operations

In order to properly run our 2DOF coaxial pen plotter product using our software, refer to the following procedure below.

1. Draw a shape or picture in Inkscape.
2. Save Inkscape drawing as an HPGL format file.
3. Measure length of pen plotter first arm (L1) and second arm (L2).
4. Run the parse_HPGL.py Python file with the HPGL file in the same folder with the proper system arguments.
5. Load the output text file onto the micropython Shoe of Brain board.
6. Open up an terminal emulator such as Putty or GTK Term on the computer.
7. Choose the category Serial set serial line /dev/ttyACM0. Set the speed and bits at 115200, 8, and 1. Set parity and flow control to None.
8. Reboot the board with Ctrl+D on the keyboard and follow the series of prompts on the REPL to run the pen plotter and print the drawing.

Limitations

All of the python files written for our pen plotter are limited to working on specific timer channels and pins for the microcontroller used on the project. The following classes which instantiate our motors, encoders, servo and controller all have their own limitations described below.

Both of our motors from *motor_sam_dima.py* are limited to working only on timers 3 and 5 for a motor frequency of 2000 Hz. Our encoders from *encoder.py* are limited to working with timers 4 and 8 on channels 1 and 2 only. Our servo from *servo.py* is currently limited to working on channel 1, timer 2 and pin A5 on the Shoe of Brain board. Our controller from *controller.py* was thoroughly tested to work for an optimal Kp value of 0.05. This value was determined from a step response test by experimenting with various Kp values and plotting the time and position data for one revolution of the motor set at 3200 encoder ticks. From the step response test for a Kp = 0.05 and a setpoint og 3200, the actual position reached about 3150 with an error of 50 and a percent error of 1.56%. Too low of a KP response never fully reaches the setpoint and thus is not a desirable response. On the other hand, too high of a KP may reach the setpoint faster than the chosen KP, but it overshoots and oscillates, another undesirable effect. If the motors are given too slow of a period, they begin to oscillate in actual position when the period is increased. On the other hand, although a faster period results in a smoother response, it is not desirable to run at period for which a slower period can still yield sufficient results for proper operation. The optimal period found for our brushed DC motors was 10 ms. Lastly, our pen plotter will only accept HPGL format files so pictures can only be drawn on programs which can save as this file format, such as Inkscape.

Location

The location of the mainpage file, along with the rest of the source code can be on our Mercurial webpage [9]. From Mercurial, all the code files for each Lab can be found under *browse* on the left side of the page. Additionally, refer to Appendix C for doxygen attachments of all the essential classes and functions used for the Pen Plotter project. All the files can also be found on Github [10].

Results and Future Steps

In the end, unfortunately, we did not finish or meet all of our initial specifications. We met all our specifications for this project except creating a legible drawing; however, we made great strides in developing a project that could potential work well. There were also some specifications we did not meet because they were stretch goals we had in mind.

When we wanted to get the pen plotter to go to a specific point, we were able to get the pen plotter to go to that point. This was tested by using a grid paper with 0.5 in lines we had made and printed out. We marked on both our device and the paper the origin of the global reference frame. With both aligned, we were able to make the arm go to a single point and confirm on the grid paper that it would reach that point within ±1 in. An example of the test and our final piece is shown below.

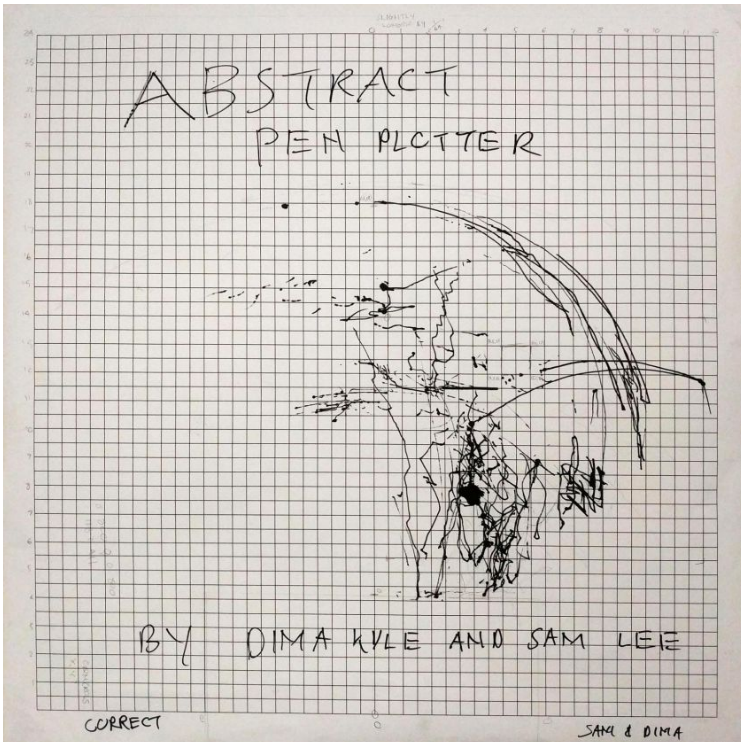


Figure 37. Example of grid paper for calibration and testing as well as our masterpiece abstract artwork.

The main issues we believe it did not work as a pen plotter are the stiction in the motors. We were not able to make the arms move small increments without overshooting the setpoint. We tried testing many different combinations of K_P, K_I, and K_D but since neither of us had much knowledge in controls, it was a crude slightly more educated guess and check of trying to see what values resulted in a qualitative better performance. Besides these issues, there were also some changes that could help make the plotter better.

If the gruesome kinematics wanted to be avoided, the second motor for arm 2 could be attached to arm one to move arm 2. This would get rid of the dependence of the angle of arm 2 on the changes in the angle of arm 1, but this would require a better design to hold the weight of the motor and avoid wire entanglement. The small servo jig could have been made to simply make the pen go up and down instead of a rotate up and down which resulted in some uncertainty on where the pen would go down as well as the dependence of the arms being parallel to the plot space. A small V-block-esque feature could be also be added to have the pen more well-seated and gripped. The ball roller element shaft coupler could have had a deeper drilled hole for the link rod to sit in. Electrically, the final wiring could have been done on a perfboard (or permaboard) but we were not sure if any changes would have to occur so we stuck with a breadboard, wires, and jumper wires. The MDF used was a bit too soft and brittle, and so it did crack from the wood screws. The belt did slip a little, and so some kind of belt tensioner would have helped tremendously.

In terms of software, the command function or HPGL task could possibly be split into different tasks since it is a complex task with states and substates.

Despite these, we did have successful results in other means. We created a relatively robust hardware design as well a process that could be recreated. It is light, sturdy, easily disassembled and reassembled, and all parts were easily accessible for maintenance. The greatest success was that we made a meticulously well documented project. The drawings, code comments, templates, images/figures/diagrams, and this report could probably be picked up easily by another person.

Throughout the project and even after, we were constantly thinking of cool ideas to expand this project. We did derive two separate kinematic equations for two different coordinate systems. There may have been better ways to approach this. We came across inverse kinematics but did not have the time to learn it. We thought of changes that could help the mechanical design, and so iterations would be helpful.

The software control of the motors could be improved a several ways. Perhaps an interrupt motor controller would have worked or a control gain scheduler. We definitely need to learn more controls, maybe such as learning and using the Ziegler Nichols tuning method. A serial command based program to control the pen plotter would be interesting. Even further, it would be very interesting to make the pen plotter follow the movements of someone controlling a mouse. Left click and hold could mean pen down and right click could be a dot or change color.

Our device has the foundation for modular arms so a modular belt system could be made. We also thought about self calibrating the position of the pen by simply knowing the length and width of a piece of paper, back-driving the motors to each corner in a particular order, and then, by using the change in the encoder ticks, the program could define the coordinate system of the plot space. The thought of this is shown in Appendix D along with the calculations.

We thoroughly enjoyed taking up this challenge. It was fun being to apply the knowledge of multiple classes and skills into one project. We will continue to work on this project and perhaps take it even further by implementing some of the ideas mentioned above.

References

[1] Bourke, Gregory. “2 DOF Articulated Pen Plotter.” Blogger, 1 Feb. 2014, 2dofpenplotter.blogspot.com/2014/02/2-dof-articulated-pen-plotter-beng.html.

[2] lingib. “CNC Dual Arm Plotter.” *Instructables*, Autodesk, 12 June 2017, www.instructables.com/id/CNC-Dual-Arm-Plotter/.

[3] Lacoste, Henri. “X-Y Plotter.” *Instructables*, Autodesk, 14 Mar. 2015, www.instructables.com/id/X-Y-Plotter-1/.

[4] “AxiDraw V3.” *Evil Mad Scientist* , shop.evilmadscientist.com/productsmenu/846.

[5] Marano, V. (2003). L6205, L6206, L6207 DUAL FULL BRIDGE DRIVERS. 1st ed. [ebook]. Available at: http://www.st.com/content/ccc/resource/technical/document/application_note/b4/77/b1/88/ab/63/40/4a/CD00004482.pdf/files/CD00004482.pdf/jcr:content/translations/en.CD00004482.pdf [Accessed 9 Jun. 2018].

[6] STM32L476xx. (2018). 1st ed. [ebook] Available at: http://www.st.com/content/ccc/resource/technical/document/datasheet/c5/ed/2f/60/aa/79/42/0b/DM00108832.pdf/files/DM00108832.pdf/jcr:content/translations/en.DM00108832.pdf [Accessed 9 Jun. 2018].

[7] “Pololu - 50:1 Metal Gearmotor 37Dx70L Mm with 64 CPR Encoder.” *Pololu Robotics & Electronics*, www.pololu.com/product/2824.

[8] Dopieralski, Radomir. “A Micropython Library for Hobby Servo Control for ESP8266.” *Bitbucket*, bitbucket.org/thesheep/micropython-servo/src.

[9] Kyle, Dima, and Sam Lee . “Mercurial.” *Mercurial*, 8 June 2018, wind.calpoly.edu/hg/mecha08.

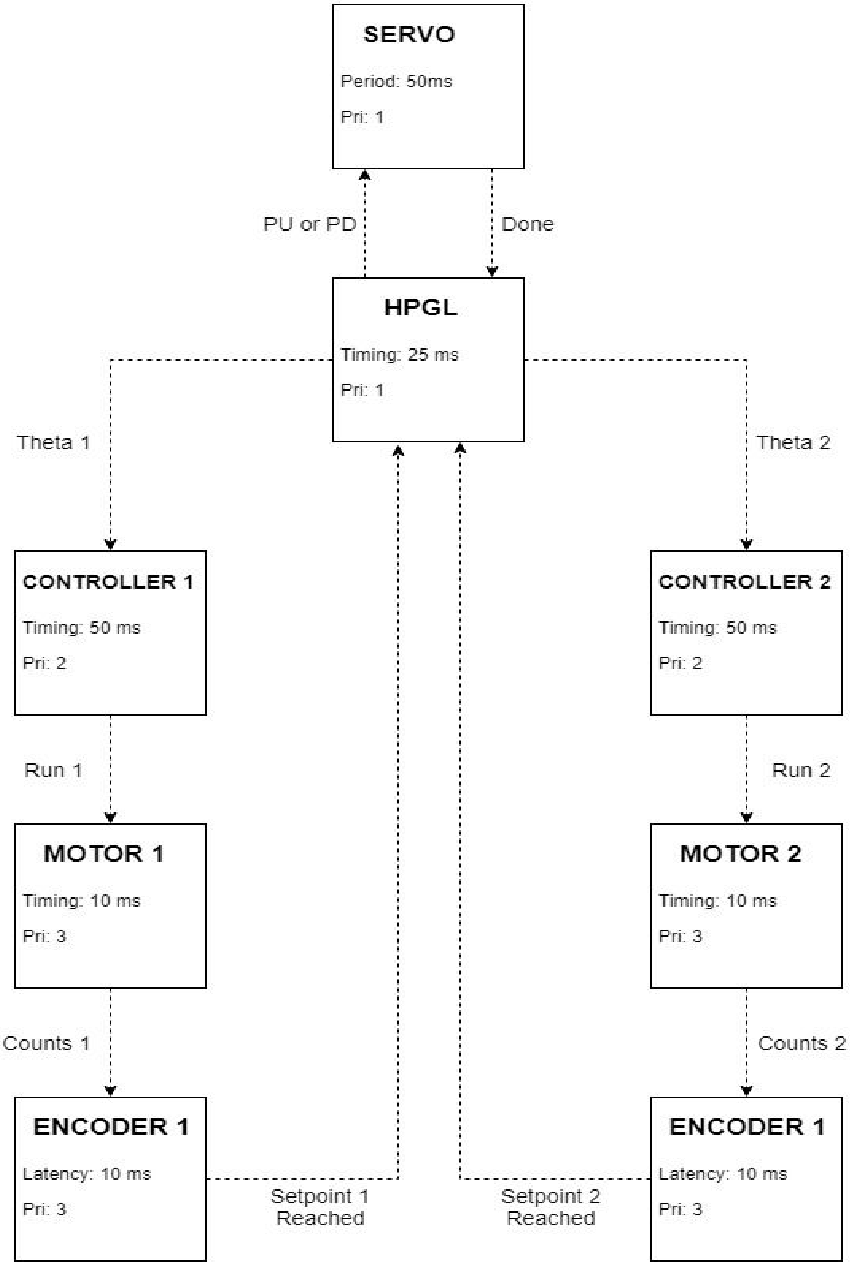
[10] Lee, Sam, and Dima Kyle. “Coaxial-Pen-Penplotter.” *GitHub*, 8 June 2018, github.com/slee32/coaxial-pen-penplotter.git.

Appendix A: Design Drawings and Assembly

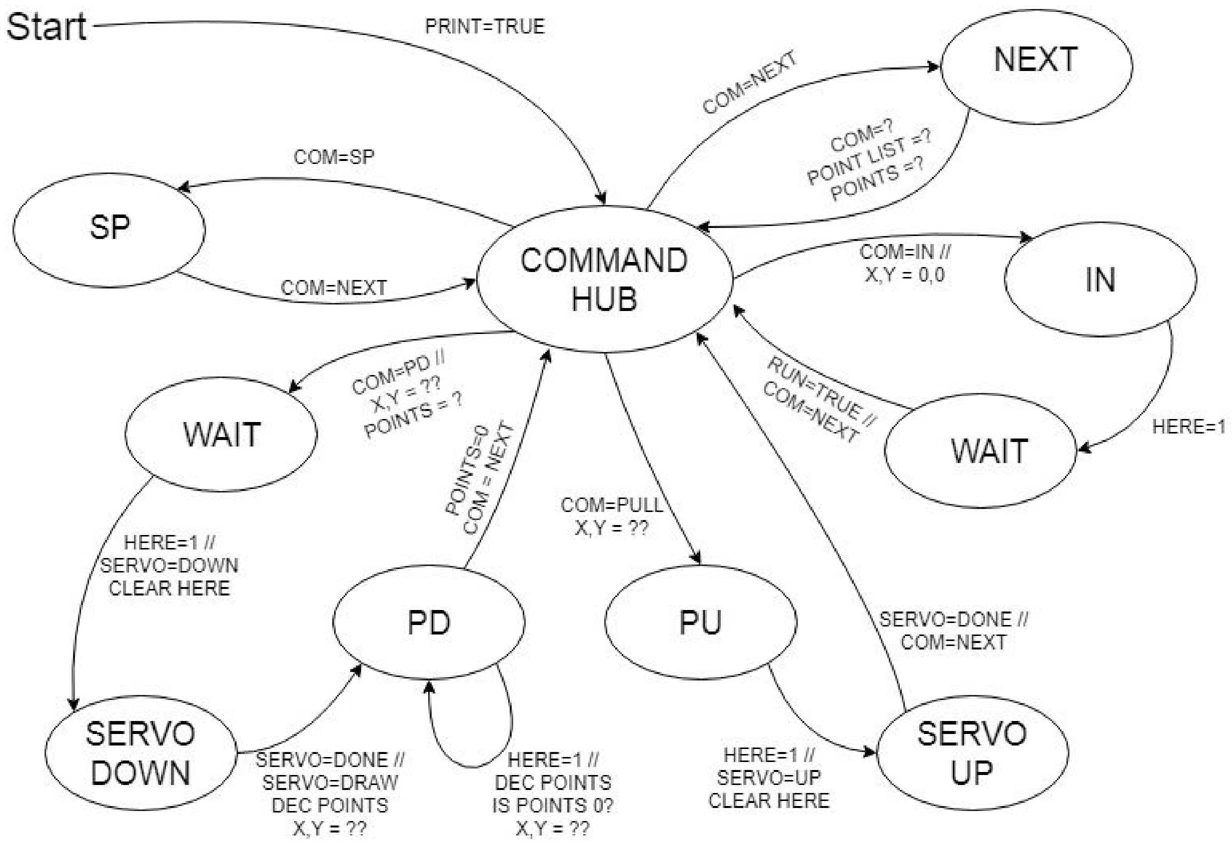
See attached.

1. Detailed drawings
2. Laser templates
3. Stock part drawings
4. Servo CMM

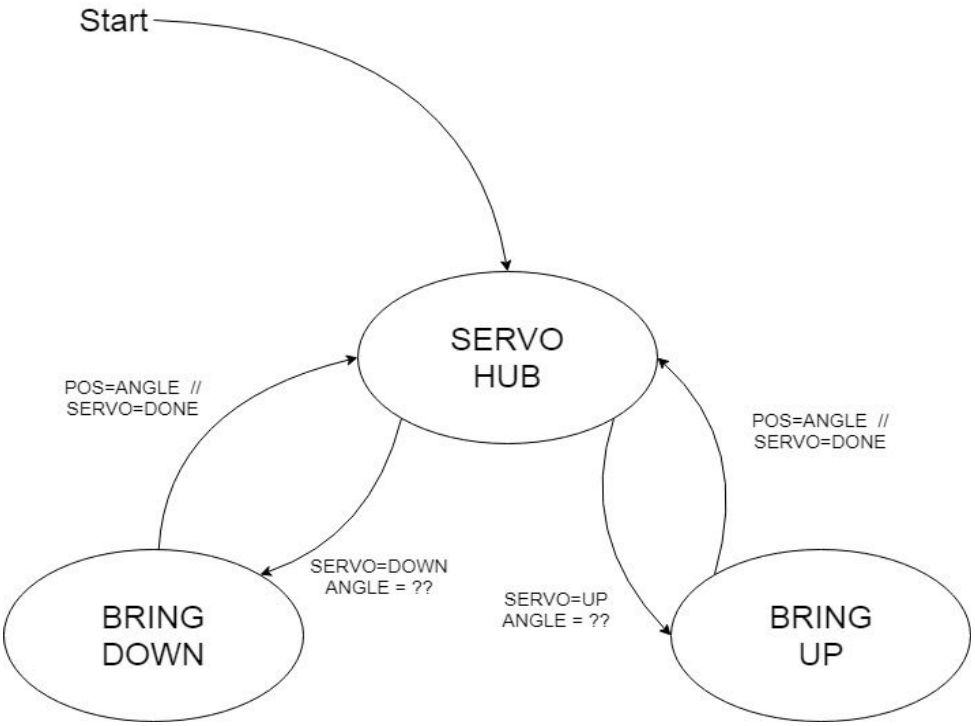
Appendix B: State Machines and Task Diagrams



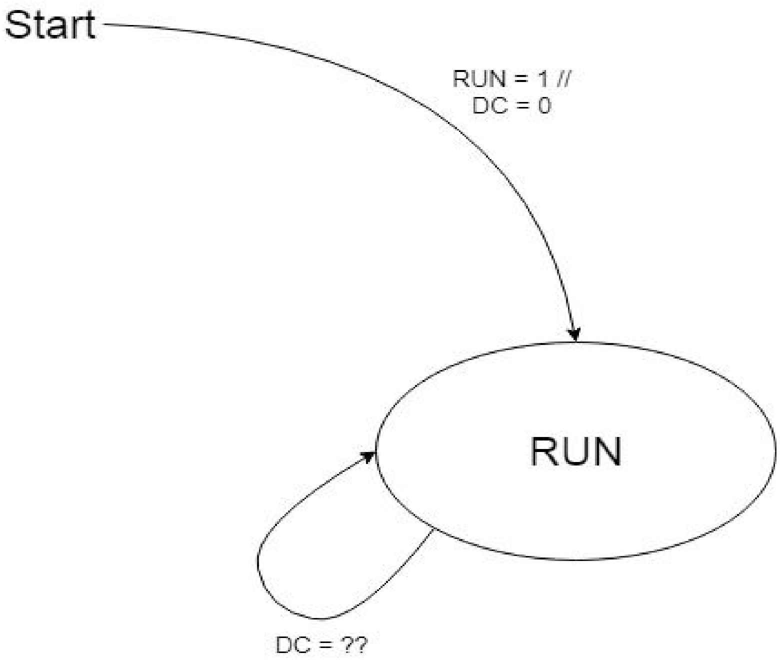
Pen Plotter Task Diagram



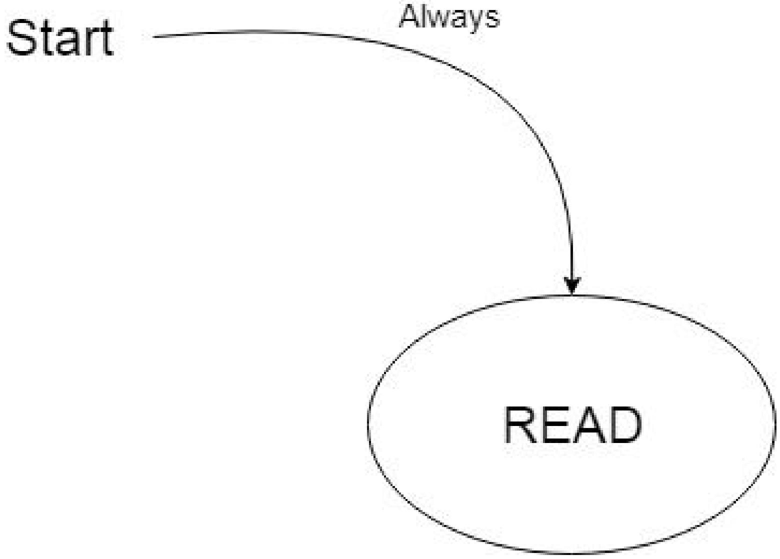
HPGL State Diagram



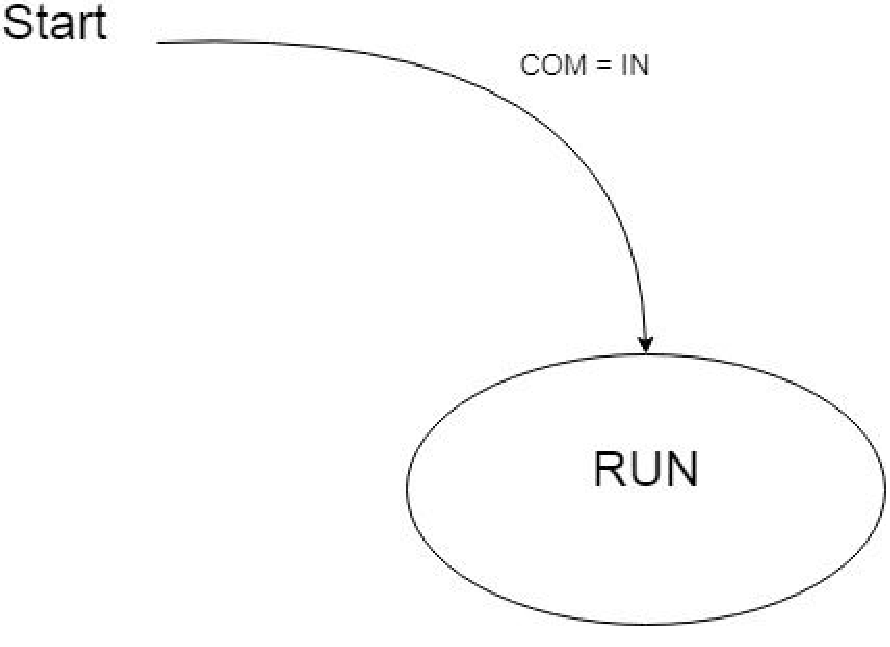
Servo State Diagram



Motor State Diagram



Encoder State Diagram



Controller State Diagram

Appendix C: Code

The location of the mainpage file, along with the rest of the source code can be found from the following link: http://wind.calpoly.edu/hg/mecha08. From Mercurial, all the code files for each Lab can be found under browse.Additionally, refer to Appendix C for doxygen printout of our code. They can also be found on Github (https://github.com/slee32/coaxial-pen-penplotter.git).

The doxygen documentation is attached.

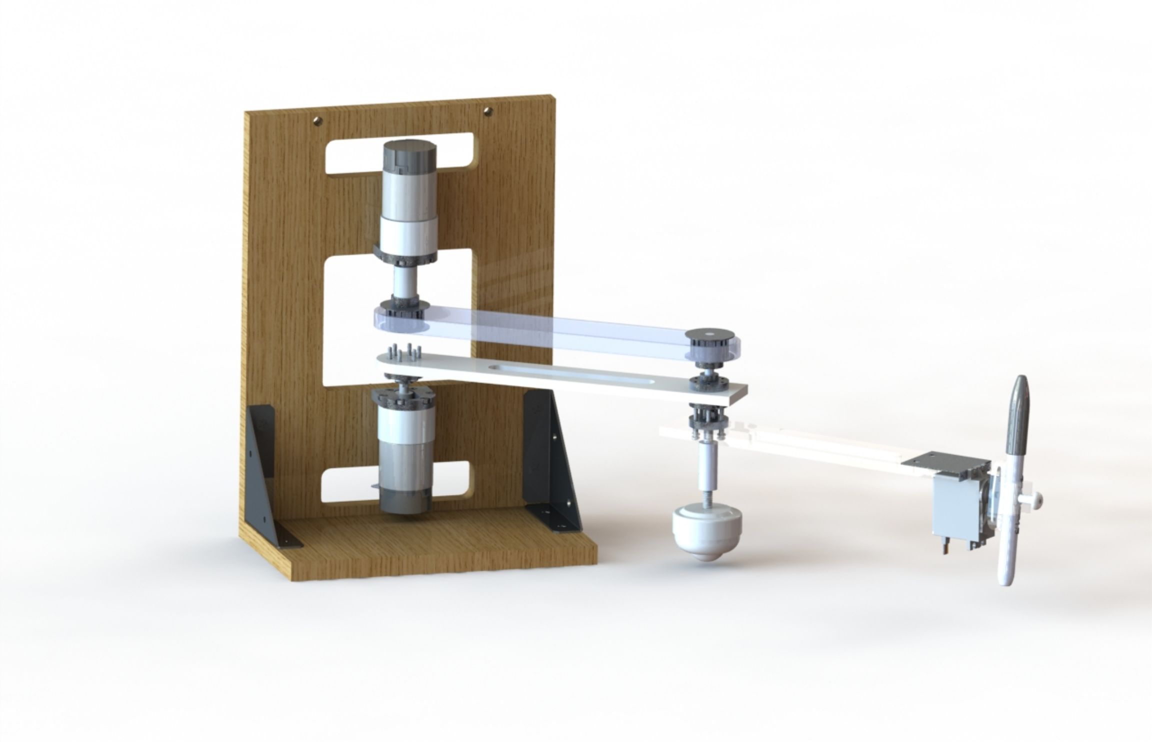
Appendix D: Calculations and Planning

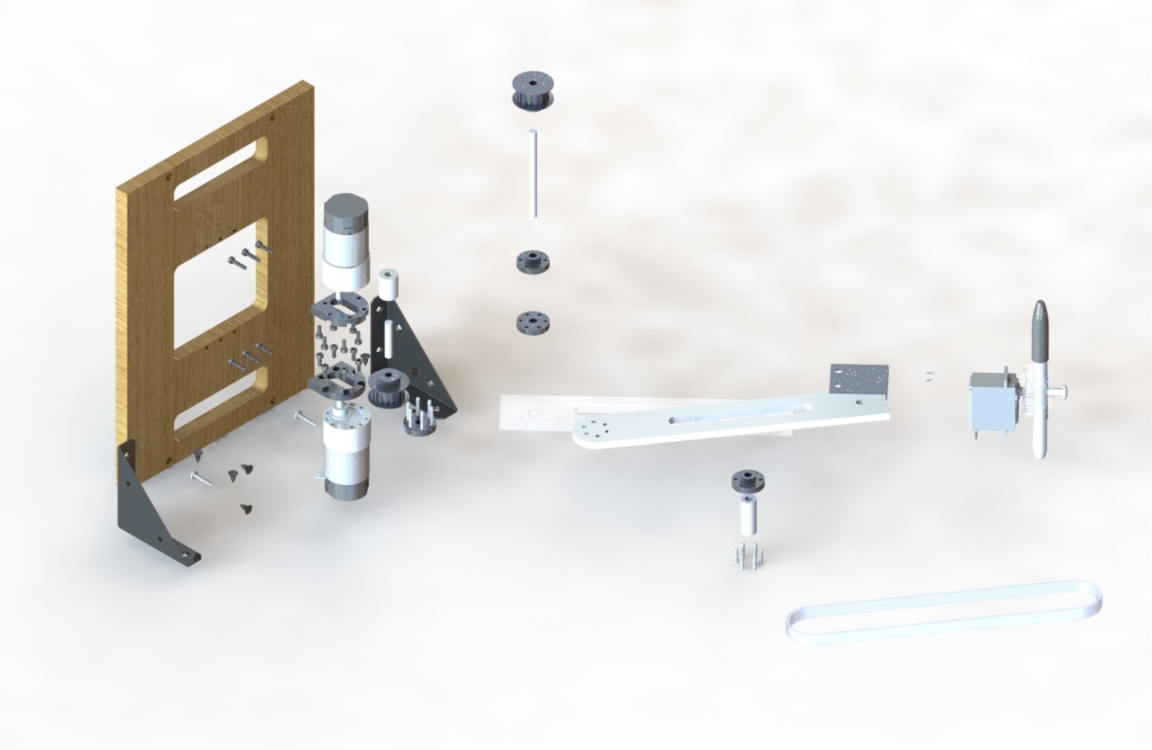
See attached.

1. Kinematics derivation
    a. First, aligned reference frame at arm 2 which rotates with arm 1
    b. CPR
    c. Calibration method and alternative method idea
    d. 2nd derivation of kinematics but with a fixed frame at the end of arm 1
2. Planning thoughts and ideas

APPENDIX A: DESIGN DRAWINGS AND ASSEMBLY

The following documents contain the assembly, part drawings, laser templates, and purchased part drawings.



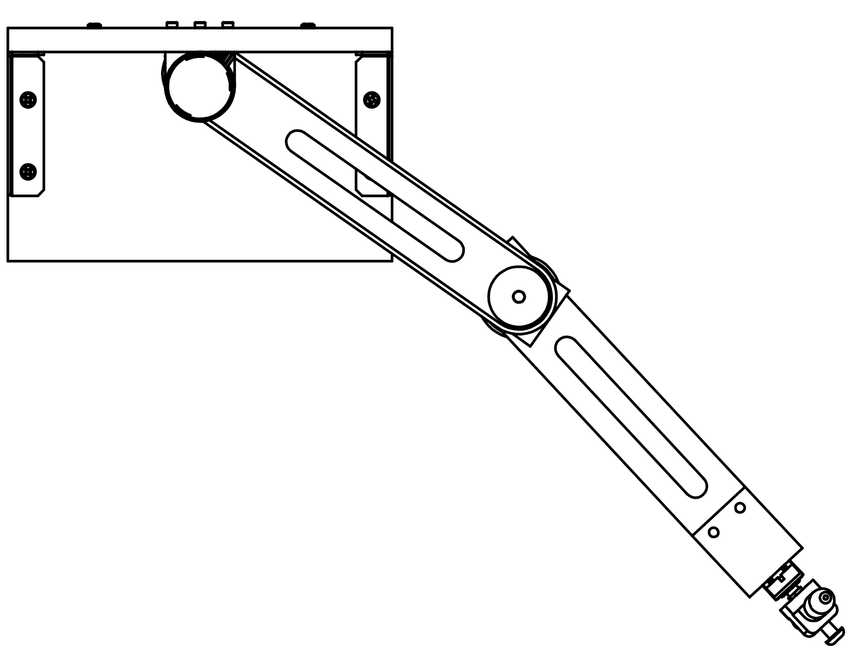


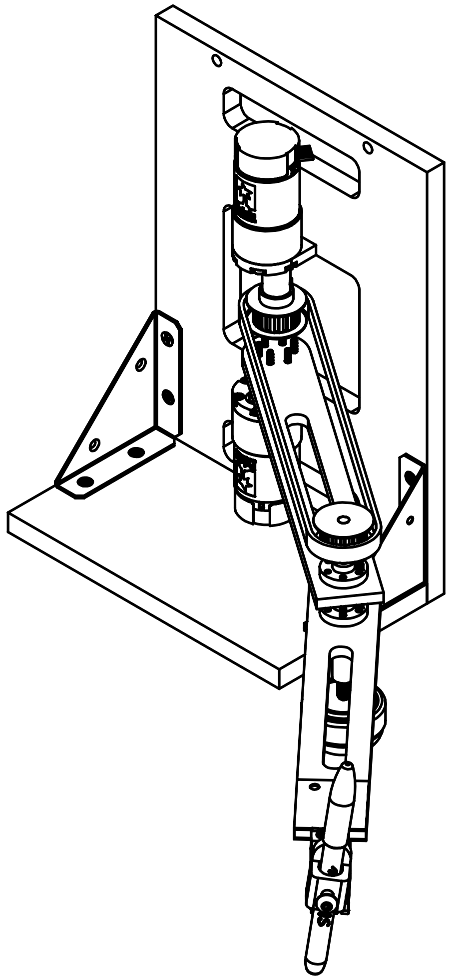
2DOF PEN PLOTTER

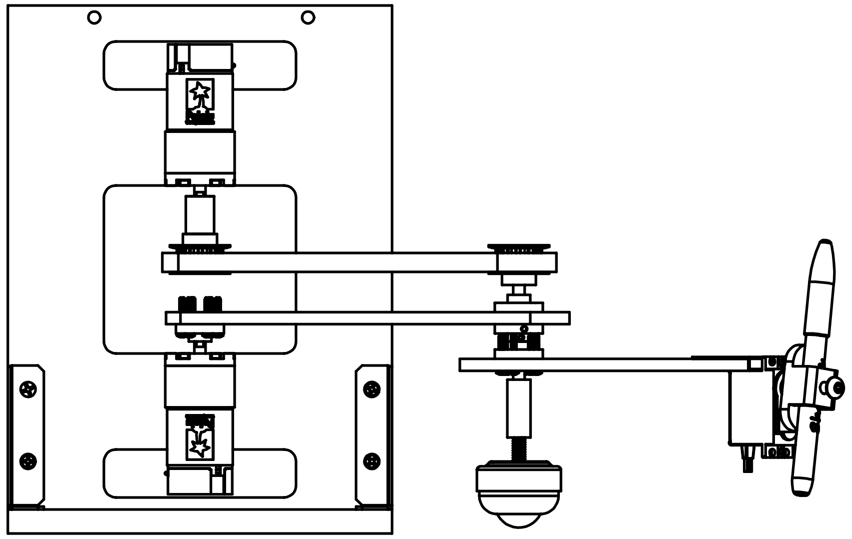
NOTES

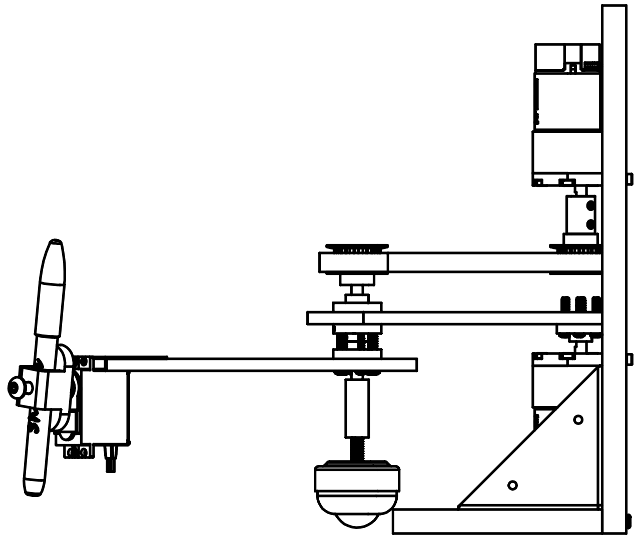
UNLESS OTHERWISE SPECIFIED:

1. ALL DIMENSIONS IN INCHES
2. TOLERANCES
   X.X = ±.1
   X.XX = ±.01
   X.XXX = ±.005
   ANGLES = ±2°
3. INSIDE TOOL RADIUS 0.5 MAX
4. BREAK SHARP EDGES 0.3 MAX
5. DRAWINGS FOR PURCHASEDPARTS AND STOCK PARTS ARE ATTACHED.







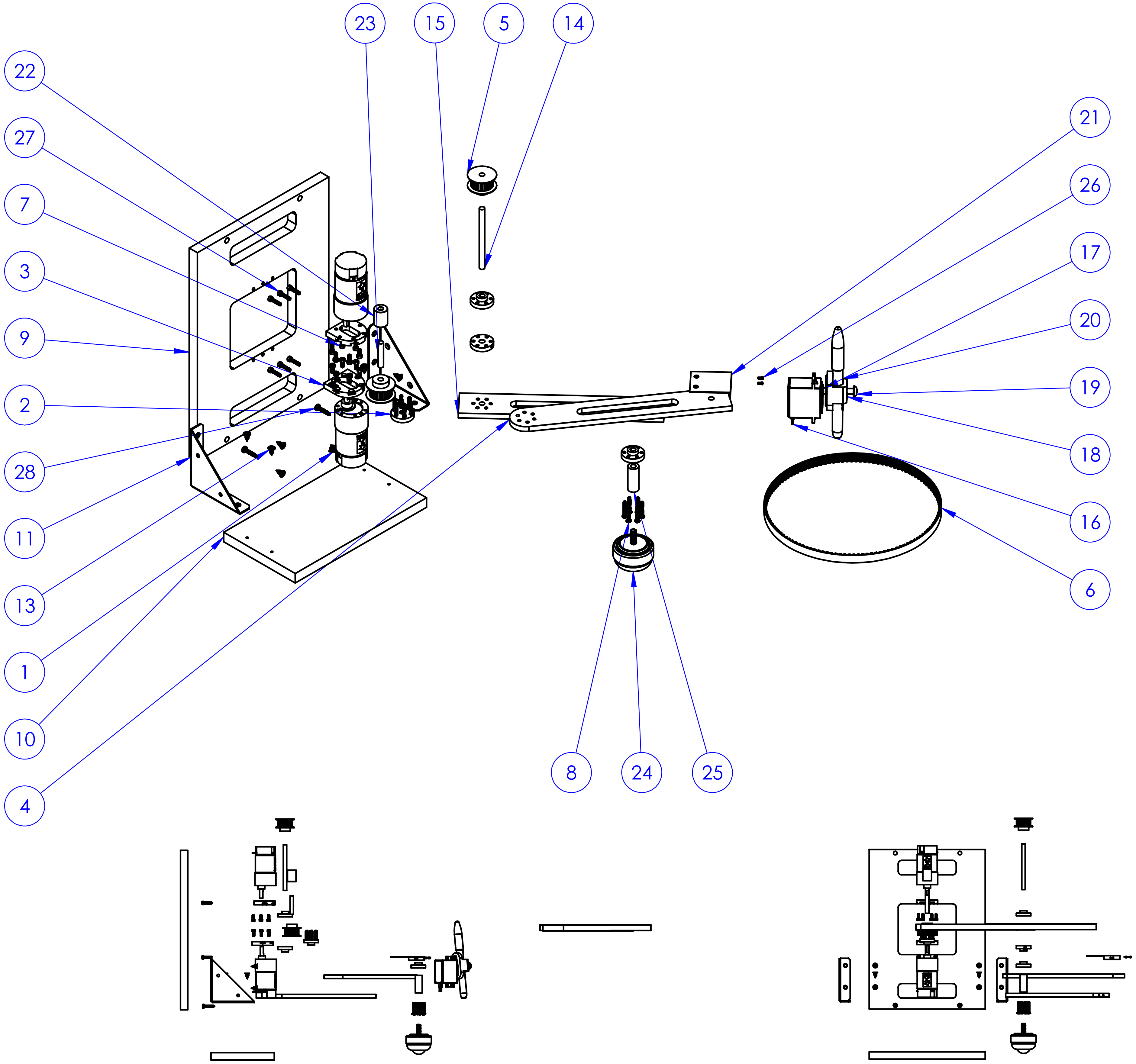


Cal Poly Mechanical Engineering	Lab Section: 04	Assignment #-1	Title: PEN PLOTTER ASSEMBLY		Drwn. By: SAM LEE & DIMA KYLE
ME 405 - SPR 2018	Dwg. #: 001	Nxt Asb: N/A	Date: 5/19/2018	Scale:	Chkd. By: ME STAFF

2DOF PEN PLOTTER ASSEMBLY

FOR ASSEMBLY INSTRUCTIONS, SEE VIDEO COLLAPSE.
https://youtu.be/rqMjX1NTY1M
SEE REPORT FOR FULL BOM INCLUDING PRICE AND DISTRIBUTOR.

ITEM NO.	DESCRIPTION	QTY.
1	MOTOR	2
2	MOTOR HUB	4
3	MOTOR BRACKET	2
4	ARM 1	1
5	1428N24	2
6	1679K544	1
7	91292A112	12
8	90604A555	12
9	MOTOR PLATE	1
10	BASE PLATE	1
11	1088A32	2
12	DERIVED BELT	1
13	90048A192	8
14	ARM LINK ROD	1
15	ARM 2	1
16	SERVO	1
17	SERVO ARM	1
18	PEN HOLDER JIG	1
19	PEN SET SCREW	1
20	SHARPIE	1
21	SERVO BRACKET	1
22	5395T111	1
23	COUPLING ROD	1
24	6460K31	1
25	STUD HOLDER	1
26	92196A053	2
27	91292A027	6
28	92325A303	2

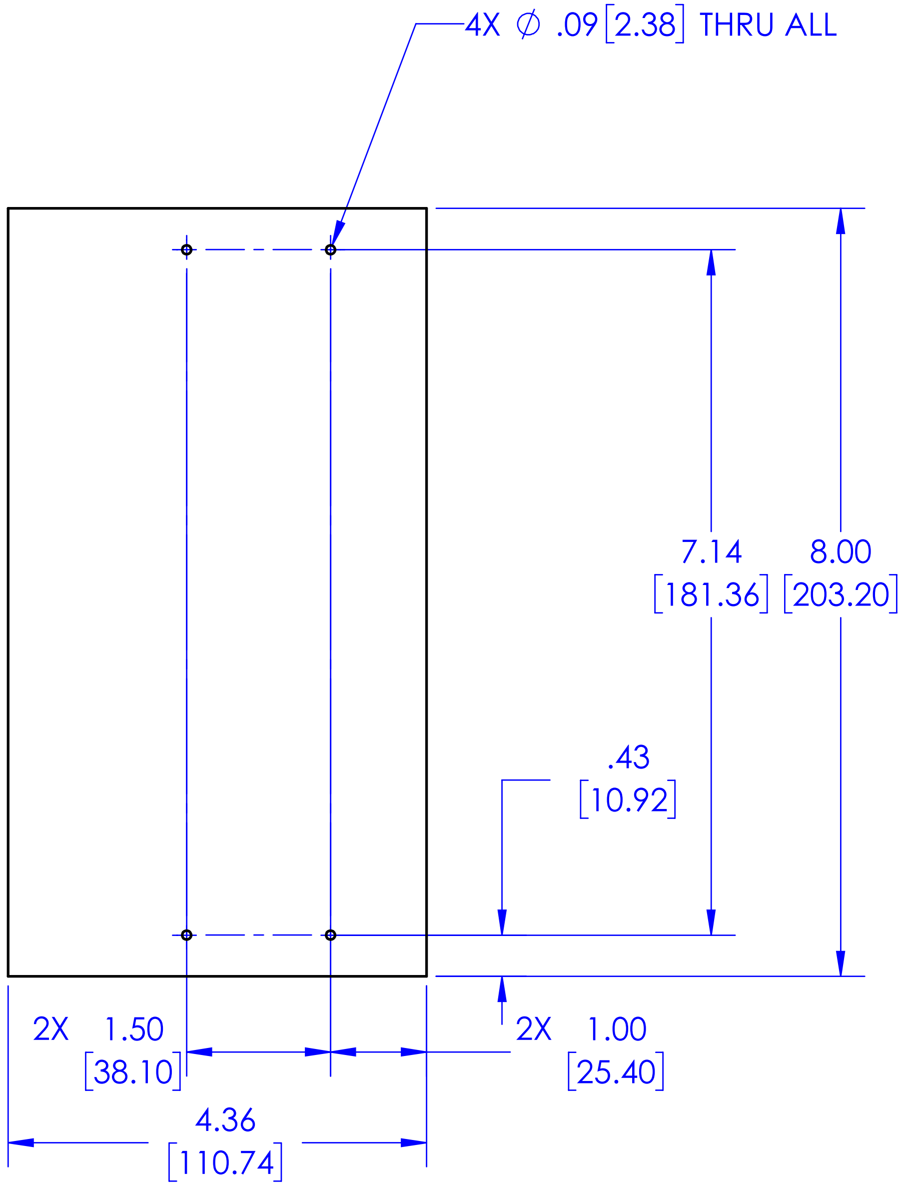


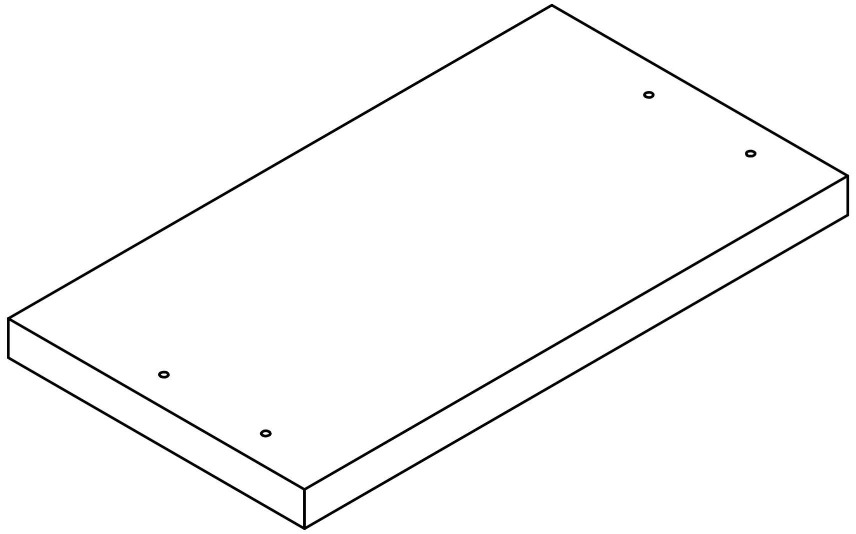
Cal Poly Mechanical Engineering	Lab Section: 04	Assignment #-1	Title: EXPLODED ASSEMBLY		Drwn. By: SAM LEE & DIMA KYLE
ME 405 - SPR 2018	Dwg. #: 002	Nxt Asb: N/A	Date: 5/19/2018	Scale:	Chkd. By: ME STAFF

BASE PLATE

NOTES

1. MATERIAL: MDF
2. THICKNESS: 1/2 IN STK.



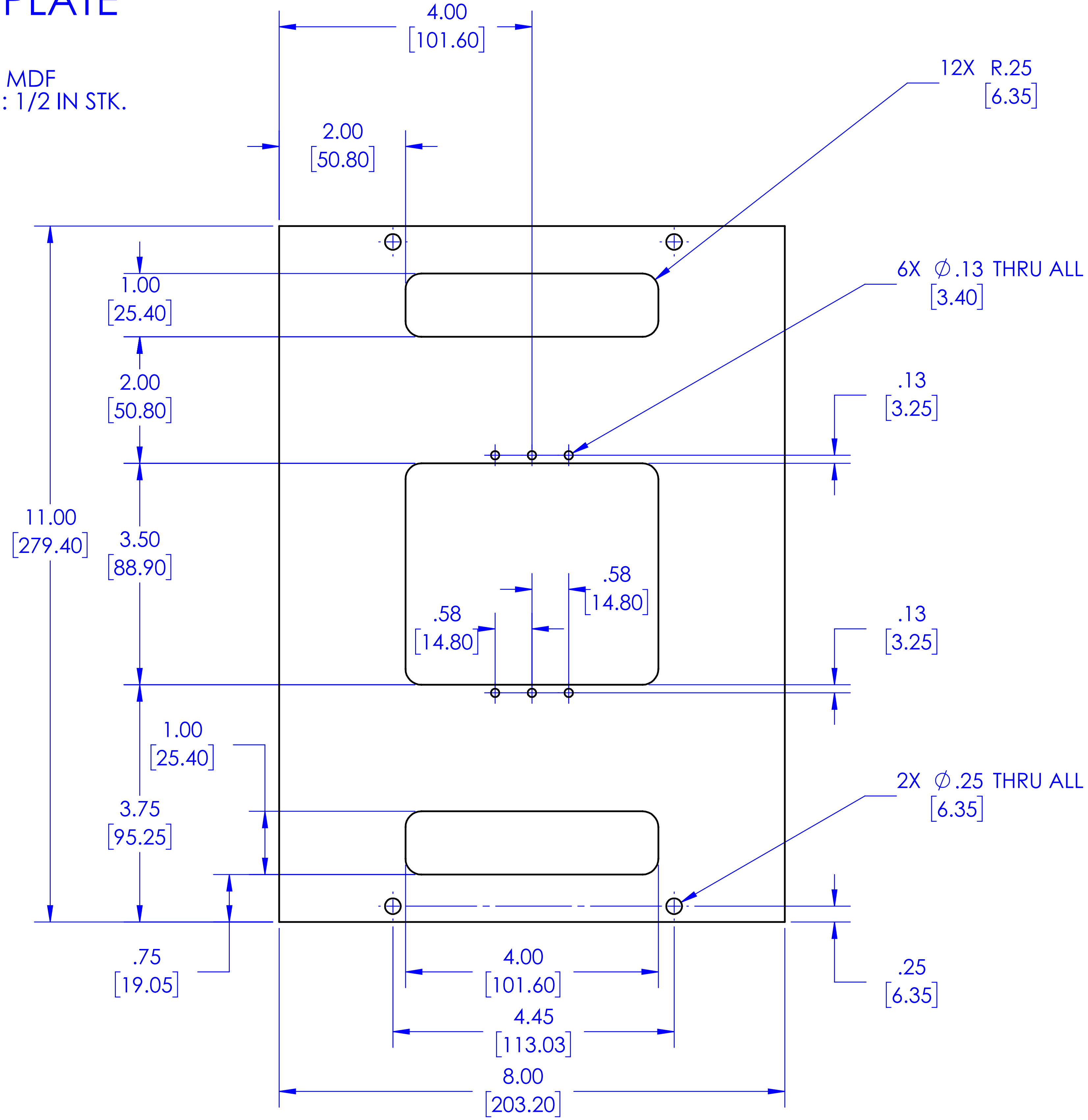


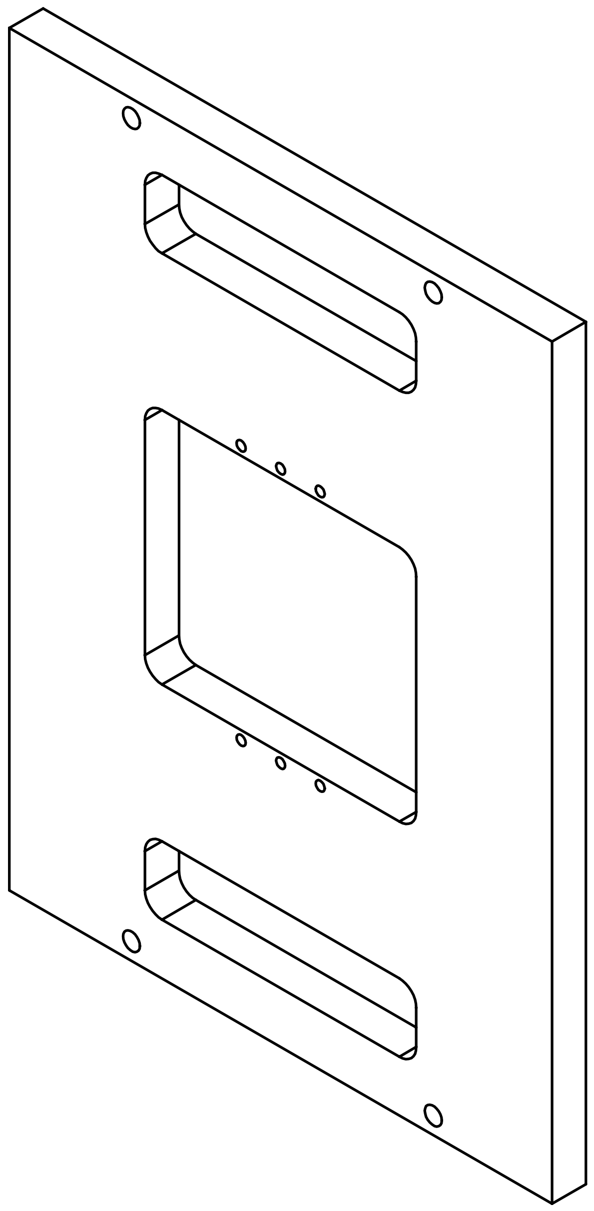
Cal Poly Mechanical Engineering ME 405 - SPR 2018	Lab Section: 04	Assignment #-1	Title: BASE PLATE	Drwn. By: DIMA KYLE & SAM LEE
	Dwg. #: 003	Nxt Asb: N/A	Date: 5/19/2018 Scale: 1:2	Chkd. By: ME STAFF

MOTOR PLATE

NOTES

1. MATERIAL: MDF
2. THICKNESS: 1/2 IN STK.



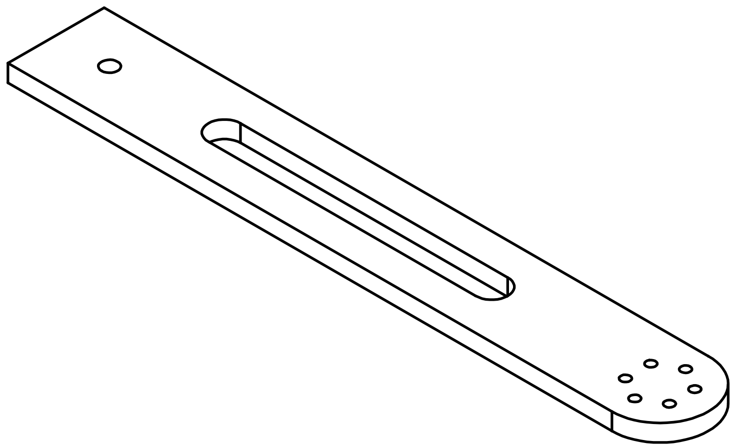


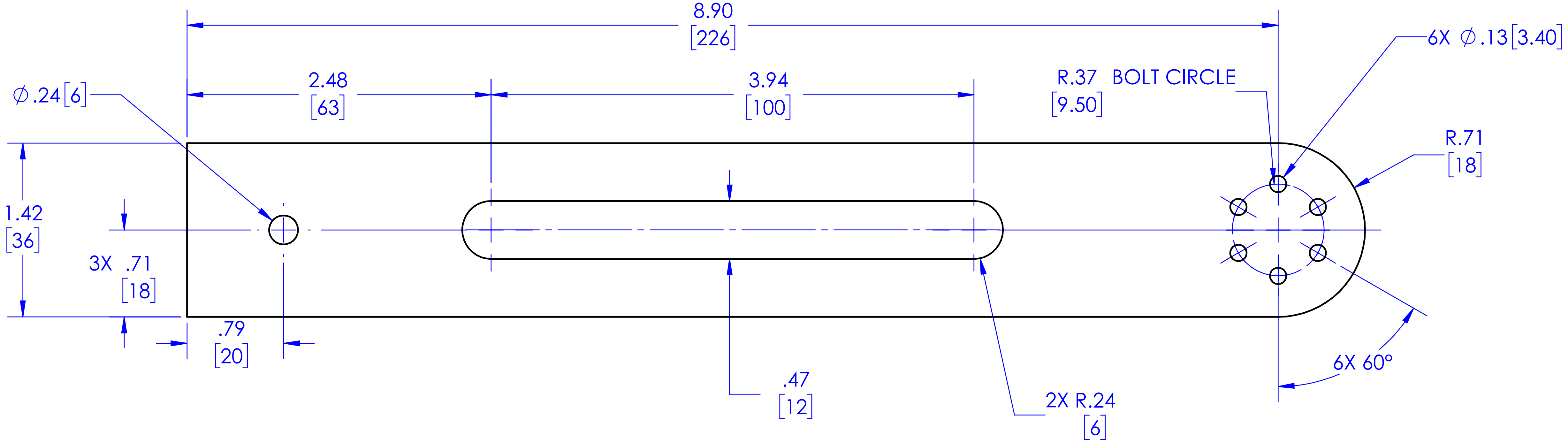
Cal Poly Mechanical Engineering	Lab Section: 04	Assignment #-1	Title: MOTOR PLATE		Drwn. By: DIMA KYLE & SAM LEE
ME 405 - SPR 2018	Dwg. #: 04	Nxt Asb: N/A	Date: 5/19/2018	Scale: 1:2	Chkd. By: ME STAFF

ARM 1

NOTES

1. MATERIAL: ACRYLIC
2. THICKNESS: 1/4 IN STK.





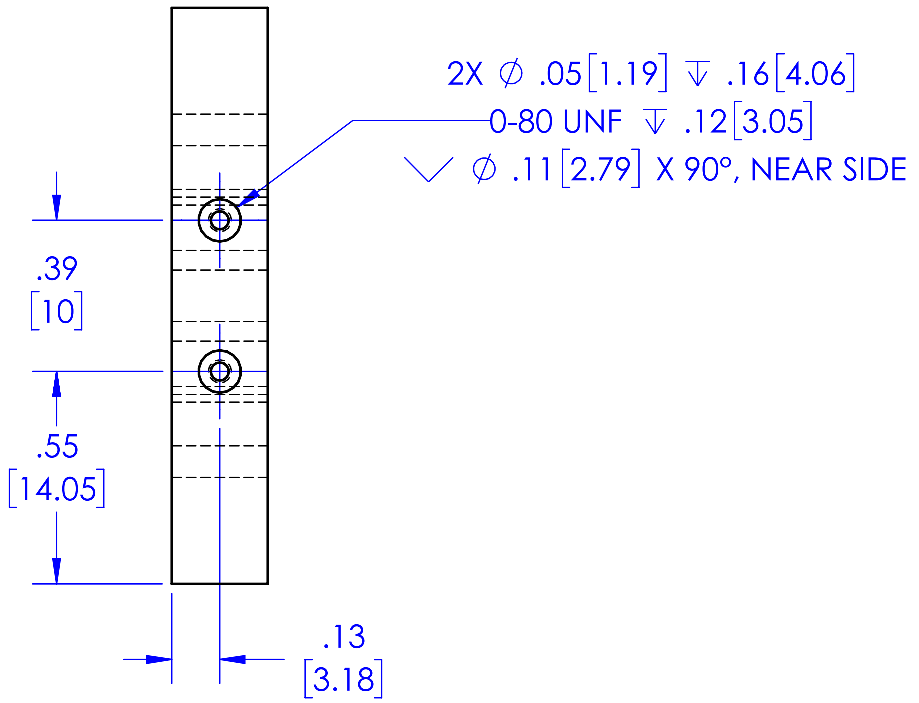
Cal Poly Mechanical Engineering
ME 405 - SPR 2018

Lab Section: 04	Assignment #-1	Title: ARM 1		Drwn. By: DIMA KYLE & SAM LEE
Dwg. #: 005	Nxt Asb: N/A	Date: 5/19/2018	Scale: 1:1	Chkd. By: ME STAFF

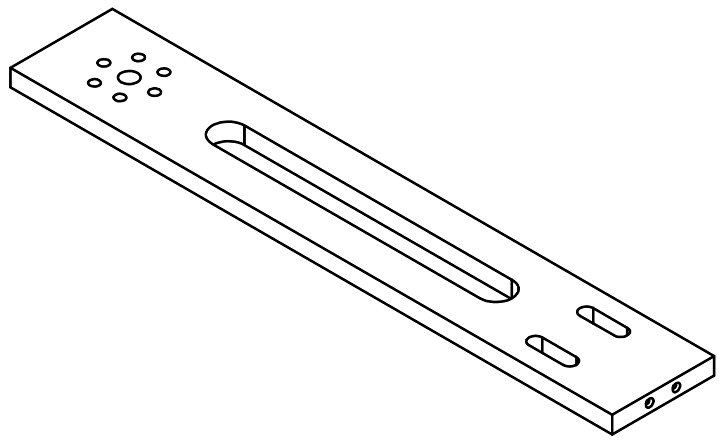
ARM 2

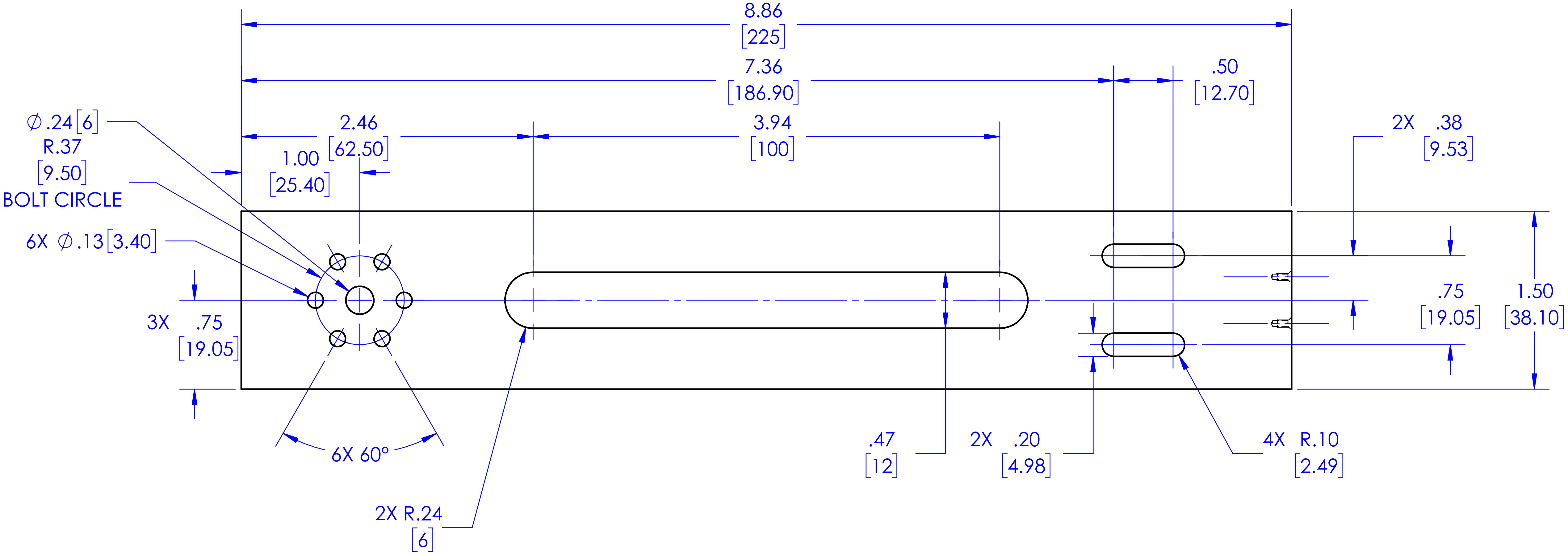
NOTES

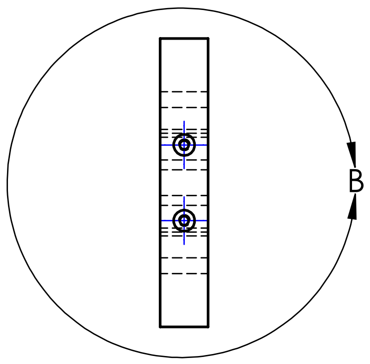
1. MATERIAL: ACRYLIC
2. THICKNESS: 1/4 IN STK.



DETAIL B
SCALE 2 : 1





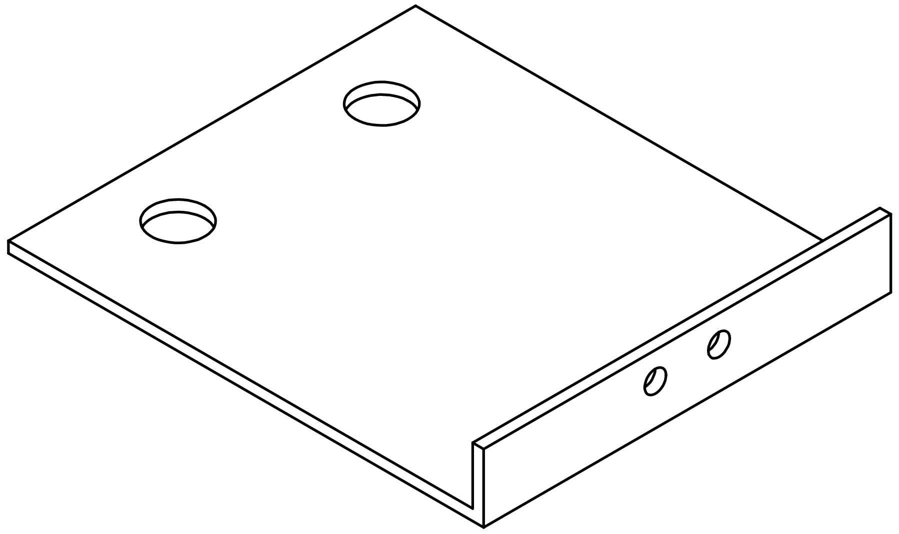


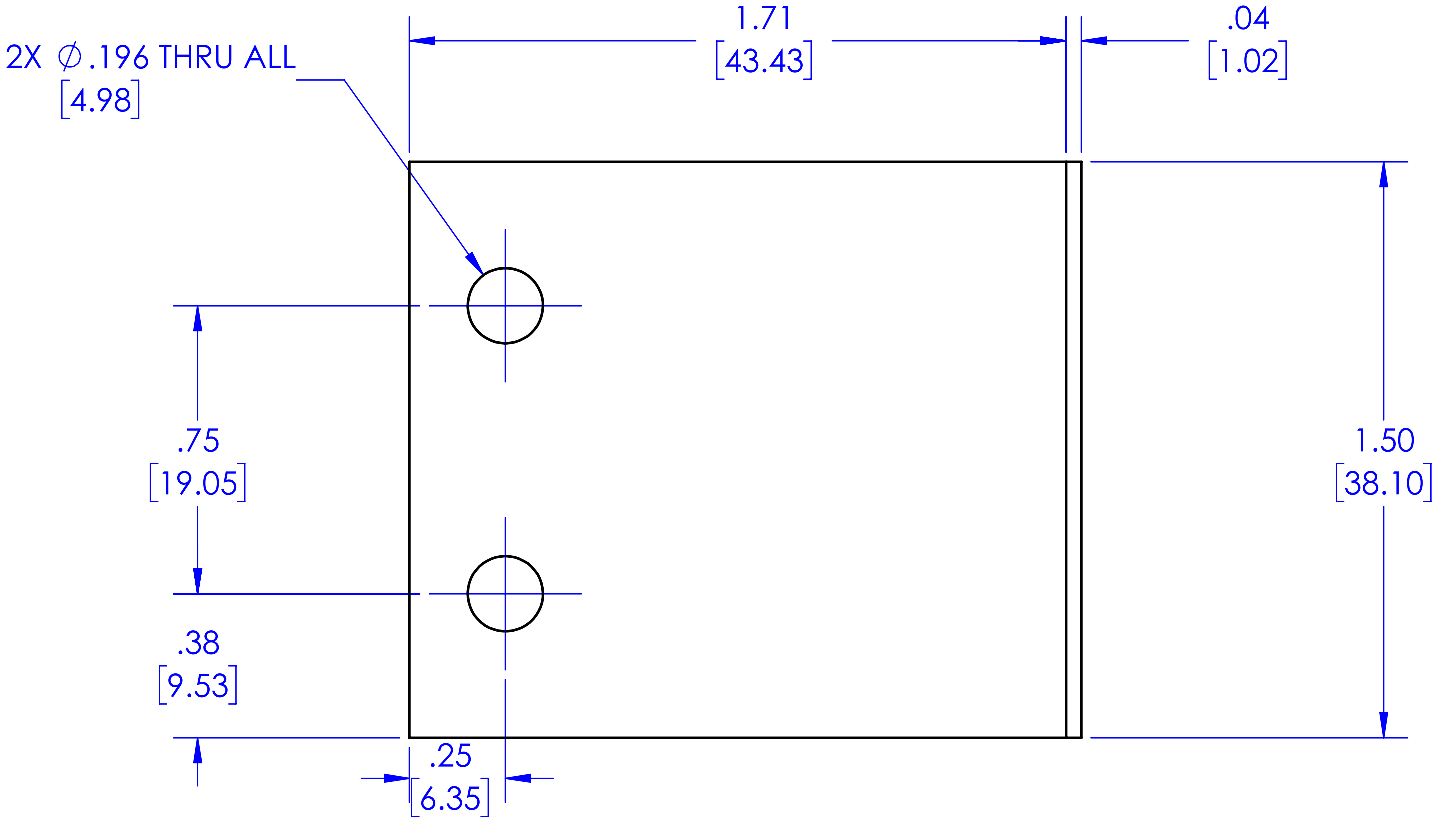
Cal Poly Mechanical Engineering ME 405 - SPR 2018	Lab Section: 04	Assignment #-1	Title: ARM 2	Drwn. By: DIMA KYLE & SAM LEE
	Dwg. #: 06	Nxt Asb: N/A	Date: 5/19/2018 · Scale: 1:1	Chkd. By: ME STAFF

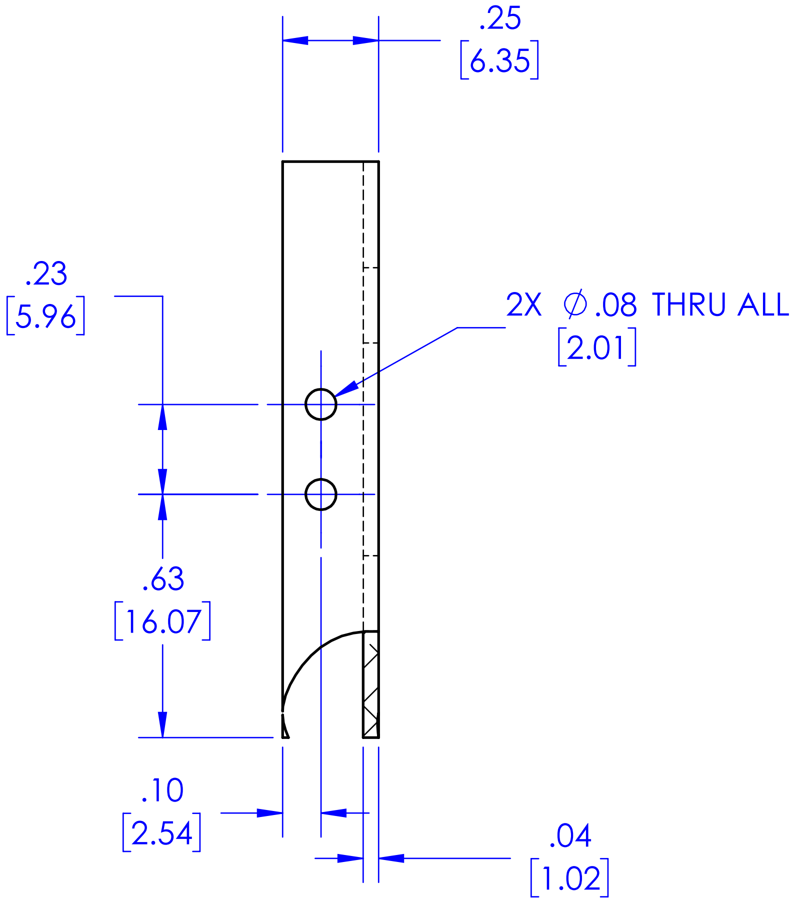
SERVO BRACKET

NOTES

1. MATERIAL: AL 6061
2. THICKNESS: 0.04 SHEET METAL
3. CORNERS MAY BE ROUNDED AND BENT
4. 90° BENDS





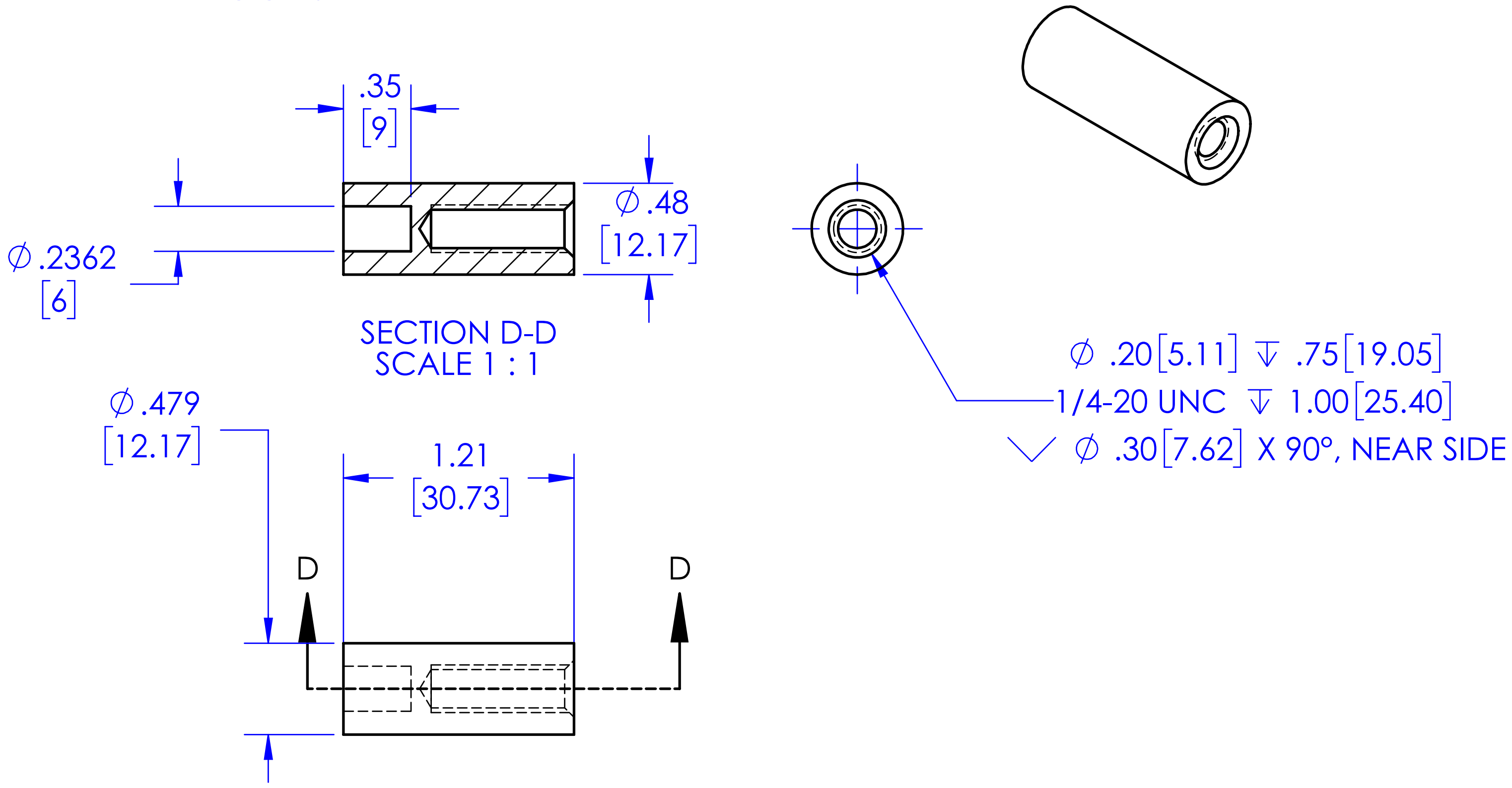


Cal Poly Mechanical Engineering	Lab Section: 04	Assignment #-1	Title: SERVO BRACKET		Drwn. By: SAM LEE & DIMA KYLE
ME 405 - SPR 2018	Dwg. #: 007	Nxt Asb: N/A	Date: 5/18/2018	Scale: 2:1	Chkd. By: ME STAFF

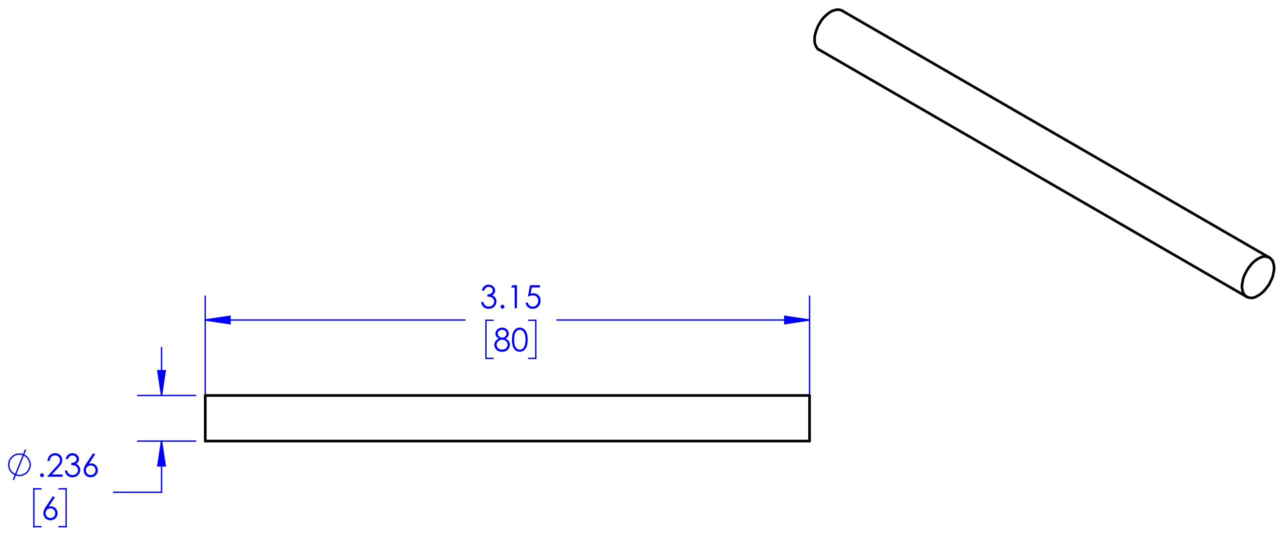
RODS

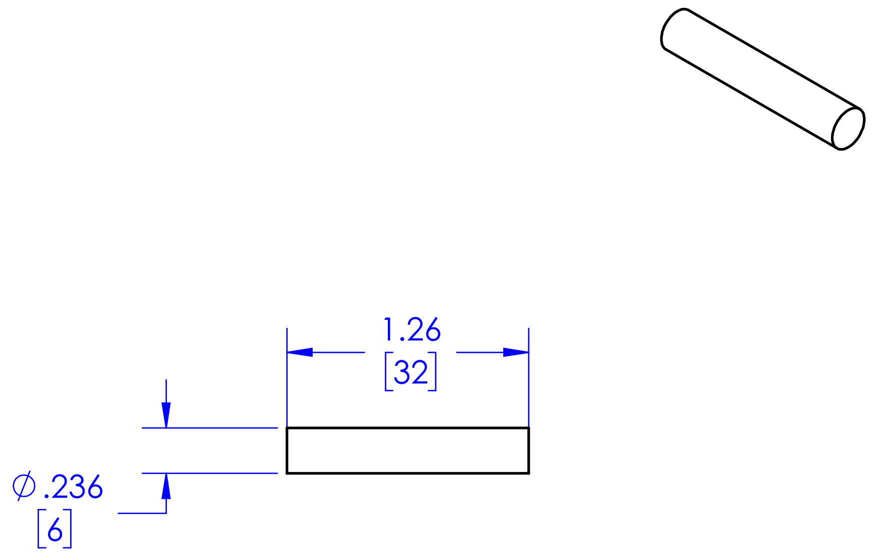
NOTES

1. MATERIAL: A2 TOOL STEEL



SECTION D-D
SCALE 1 : 1



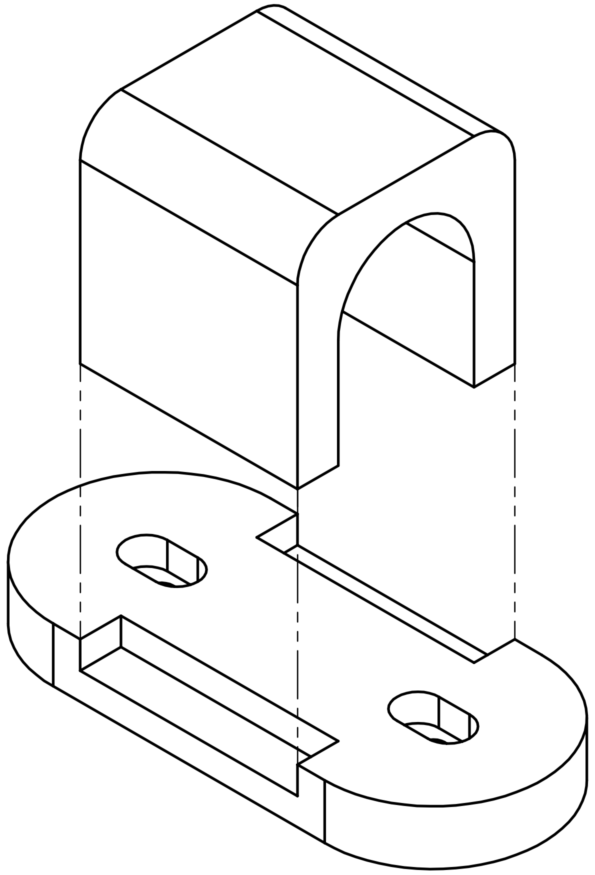


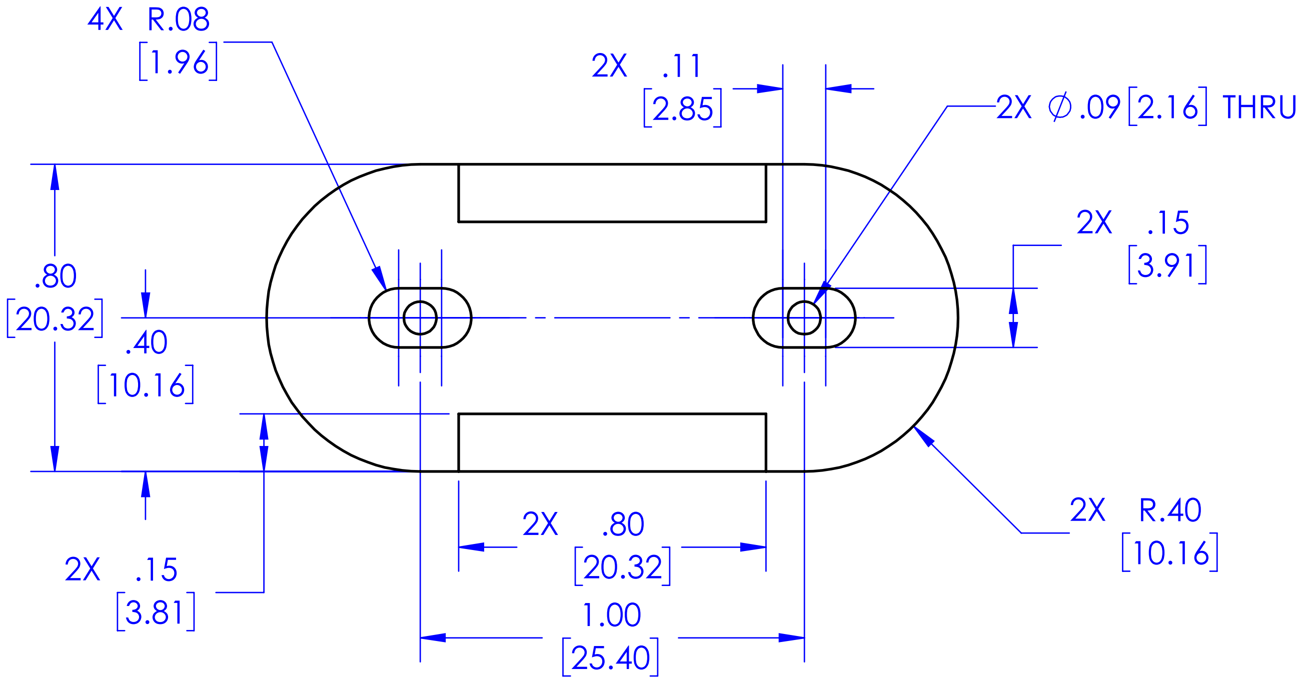
Cal Poly Mechanical Engineering ME 405 - SPR 2018	Lab Section: 04	Assignment #-1	Title: RODS	Drwn. By: DIMA KYLE & SAM LEE
	Dwg. #: 08	Nxt Asb: N/A	Date: 5/19/2018 · Scale: 1:1	Chkd. By: ME STAFF

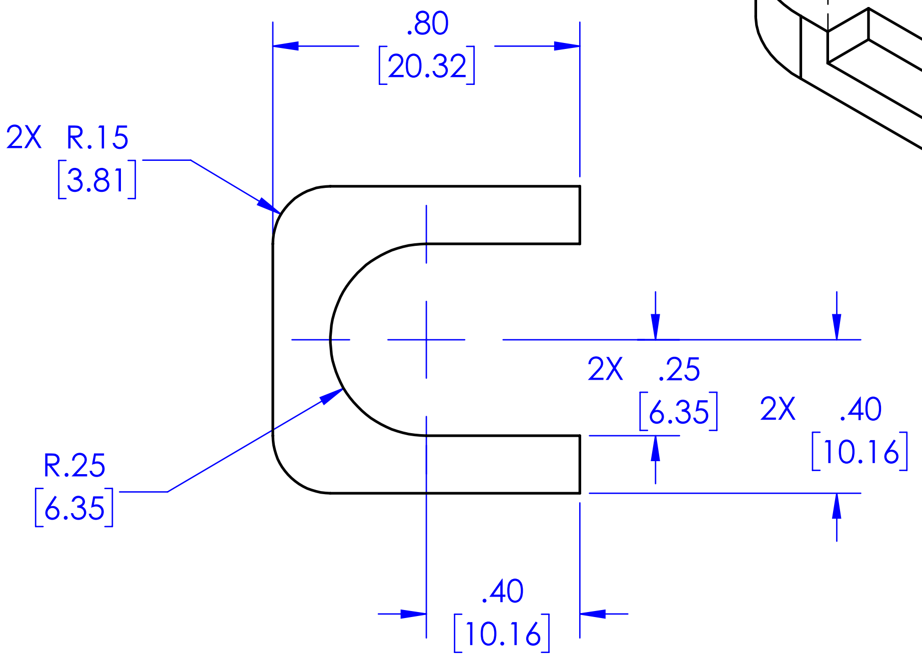
PEN HOLDER JIG 3D PRINT

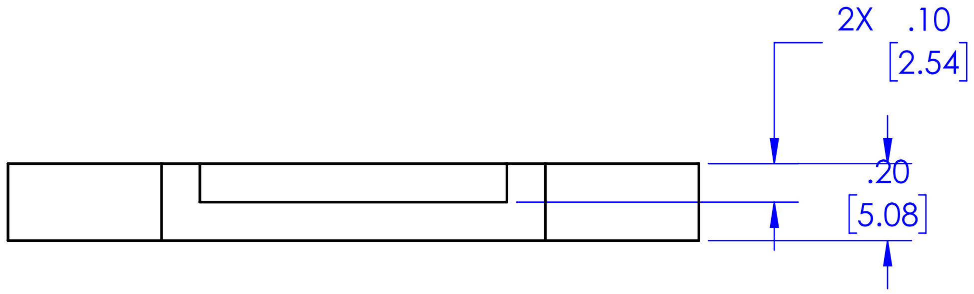
NOTES

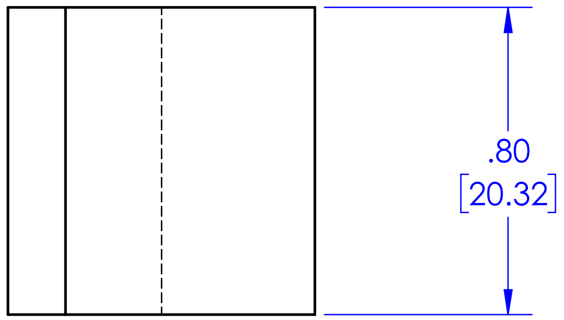
1. MATERIAL: ABS
2. TWO PARTS EPOXIED TOGETHER









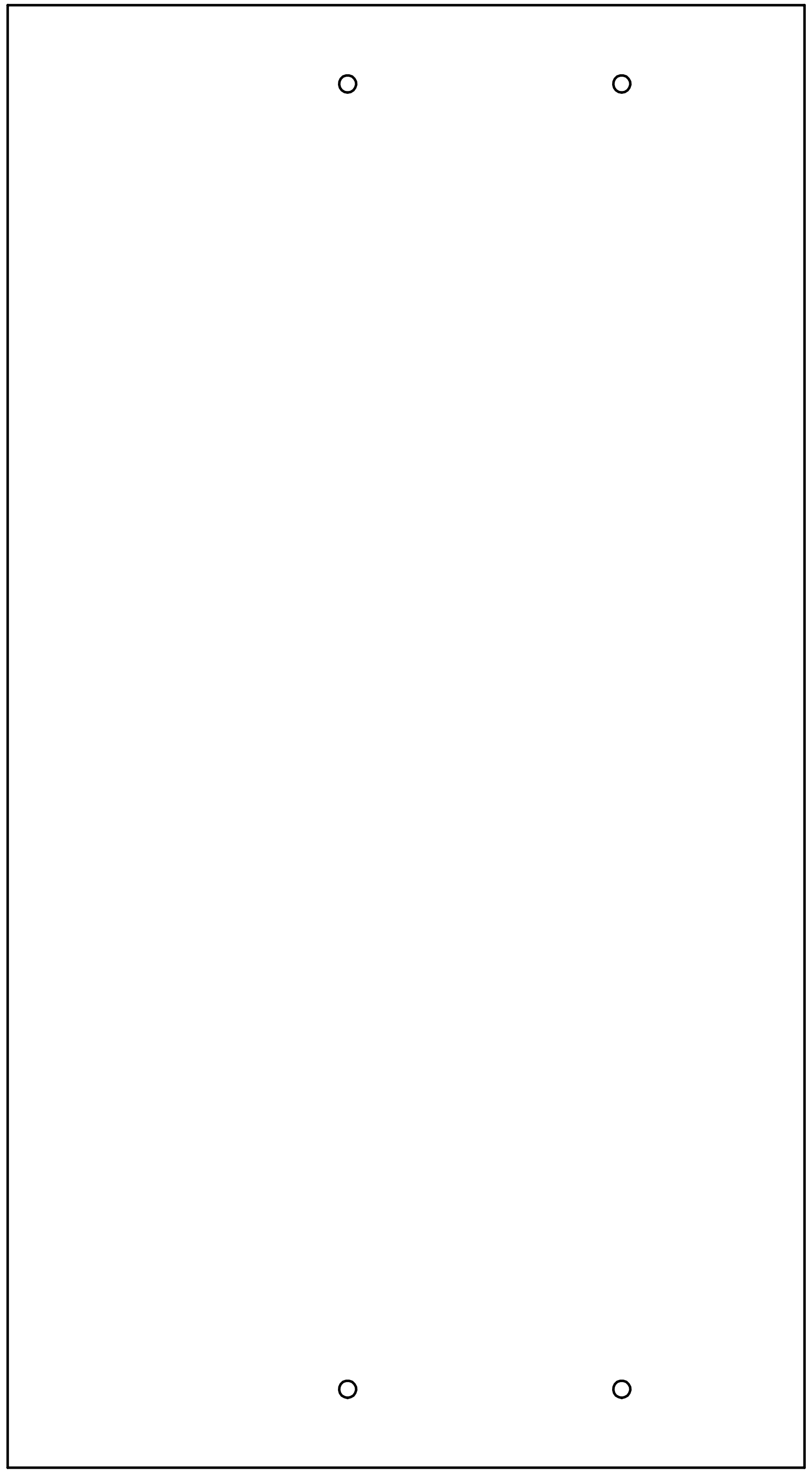


Cal Poly Mechanical Engineering ME 405 - SPR 2018	Lab Section: 04	Assignment #-1	Title: PEN HOLDER JIG		Drwn. By: DIMA KYLE & SAM LEE
	Dwg. #: 09	Nxt Asb: N/A	Date: 5/19/2018	Scale: 2:1	Chkd. By: ME STAFF

BASE PLATE LASER TEMPLATE

NOTES

1. MATERIAL: MDF
2. THICKNESS: 1/2 IN STK.

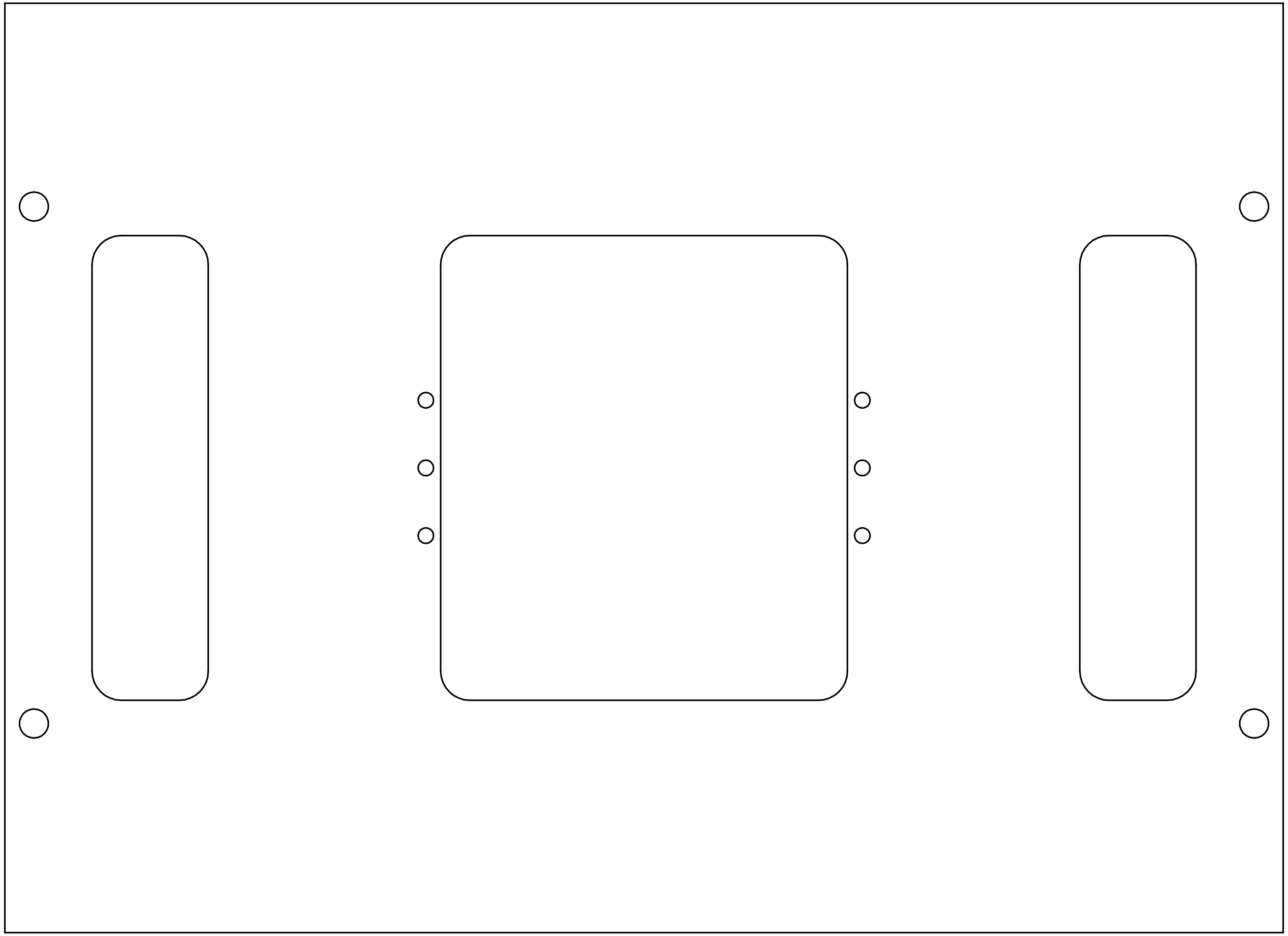


Cal Poly Mechanical Engineering ME 405 - SPR 2018	Lab Section: 04	Assignment #-1	Title: BASE PLATE	Drwn. By: DIMA KYLE & SAM LEE
	Dwg. #: 003	Nxt Asb: N/A	Date: 5/19/2018 — Scale: 1:1	Chkd. By: ME STAFF

MOTOR PLATE LASER TEMPLATE

NOTES

1. MATERIAL: MDF
2. THICKNESS: 1/2 IN STK.



Cal Poly Mechanical Engineering	Lab Section: 04	Assignment #-1	Title: MOTOR PLATE		Drwn. By: DIMA KYLE & SAM LEE
ME 405 - SPR 2018	Dwg. #: 04	Nxt Asb: N/A	Date: 5/19/2018	Scale: 1:1	Chkd. By: ME STAFF

ARM 1 LASER TEMPLATE

NOTES

1. MATERIAL: ACRYLIC
2. THICKNESS: 1/4 IN STK.

Cal Poly Mechanical Engineering ME 405 - SPR 2018	Lab Section: 04	Assignment #-1	Title: ARM 1	Drwn. By: DIMA KYLE & SAM LEE
	Dwg. #: 005	Nxt Asb: N/A	Date: 5/19/2018 | Scale: 1:1	Chkd. By: ME STAFF

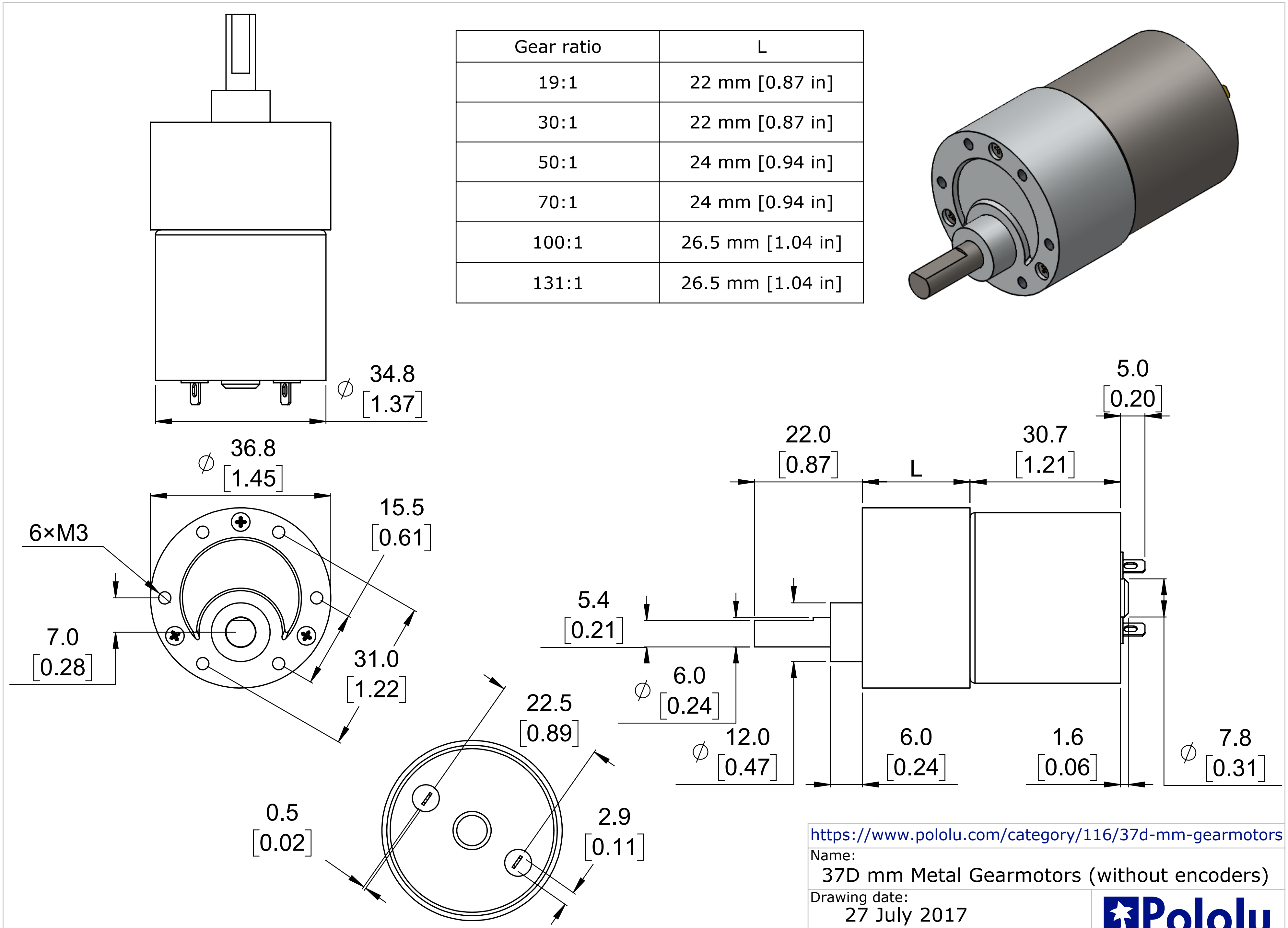
ARM 2 LASER TEMPLATE

NOTES

1. MATERIAL: ACRYLIC
2. THICKNESS: 1/4 IN STK.

Cal Poly Mechanical Engineering ME 405 - SPR 2018	Lab Section: 04	Assignment #-1	Title: ARM 2	Drwn. By: DIMA KYLE & SAM LEE
	Dwg. #: 06	Nxt Asb: N/A	Date: 5/19/2018 Scale: 1:1	Chkd. By: ME STAFF

Gear ratio	L
19:1	22 mm [0.87 in]
30:1	22 mm [0.87 in]
50:1	24 mm [0.94 in]
70:1	24 mm [0.94 in]
100:1	26.5 mm [1.04 in]
131:1	26.5 mm [1.04 in]



1. To get the specified scale, select 100% in print settings.

Scale: 1:1

https://www.pololu.com/category/116/37d-mm-gearmotors

Name:
37D mm Metal Gearmotors (without encoders)

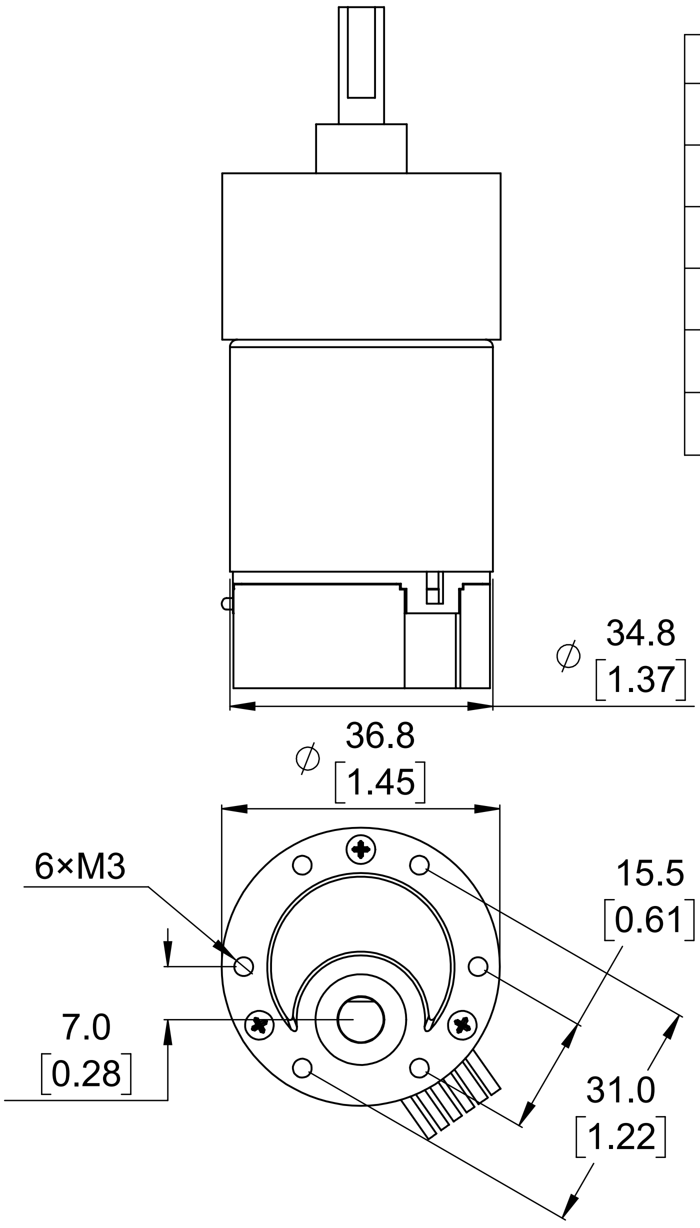
Drawing date:
27 July 2017

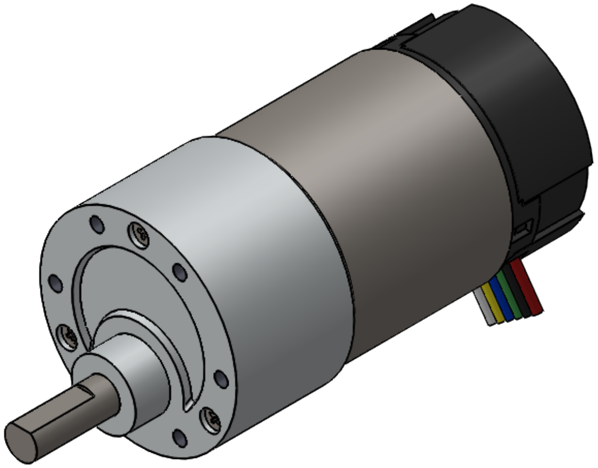
Units: mm [in]

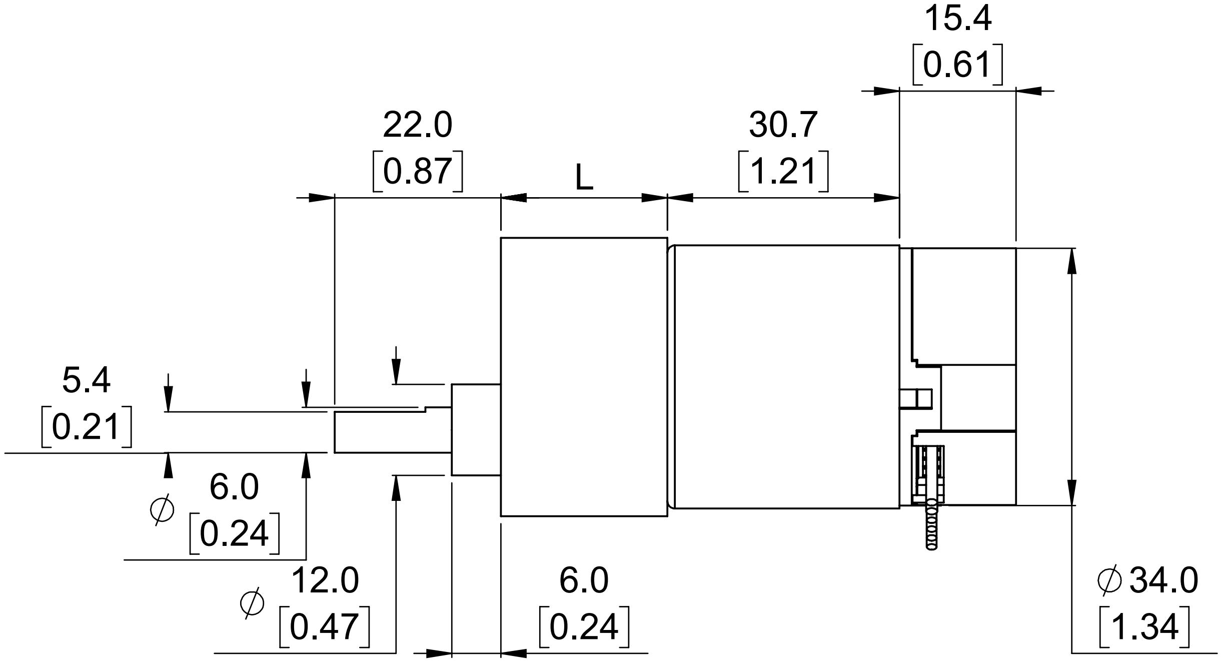
Material: mix

Pololu Robotics & Electronics

Gear ratio	L
19:1	22 mm [0.87 in]
30:1	22 mm [0.87 in]
50:1	24 mm [0.94 in]
70:1	24 mm [0.94 in]
100:1	26.5 mm [1.04 in]
131:1	26.5 mm [1.04 in]







https://www.pololu.com/category/116/37d-mm-gearmotors

Name:
37D mm Metal Gearmotors (with encoders)

Drawing date:
27 July 2017

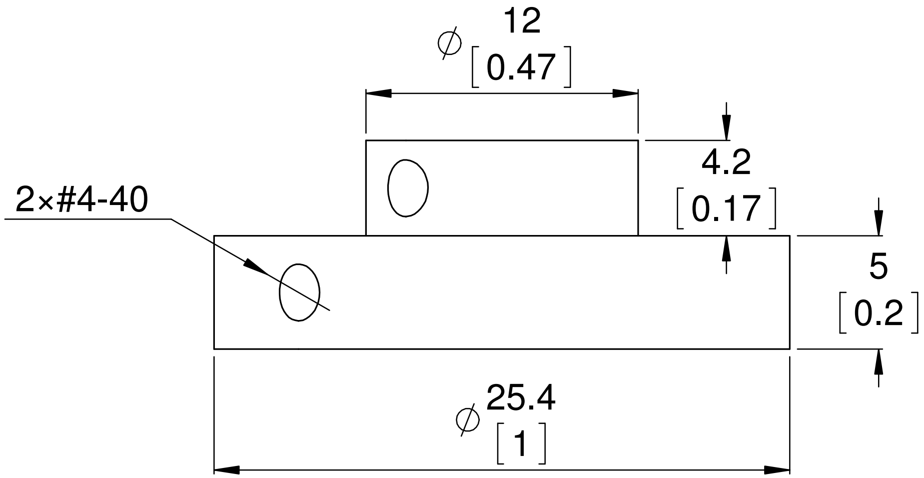
Units: mm [in]

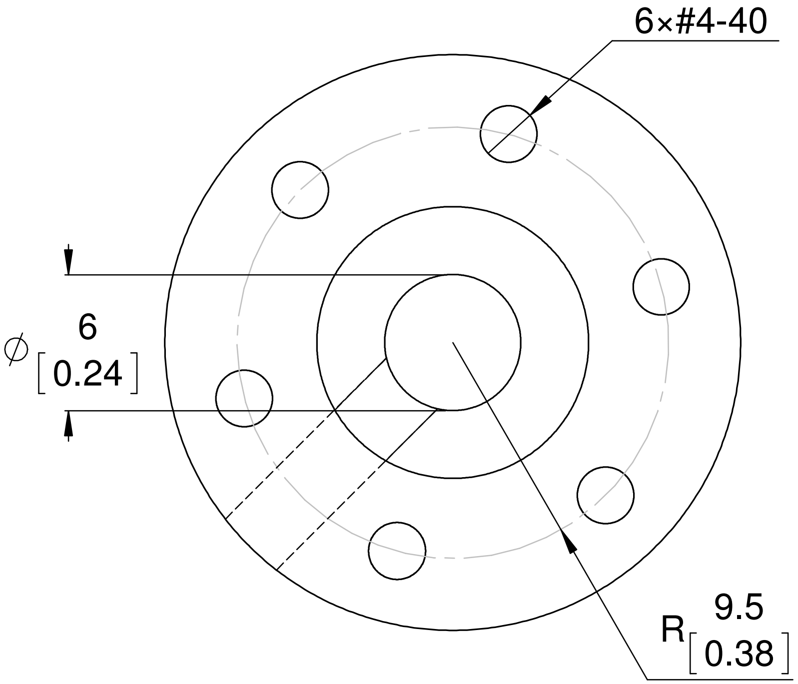
Material: mix

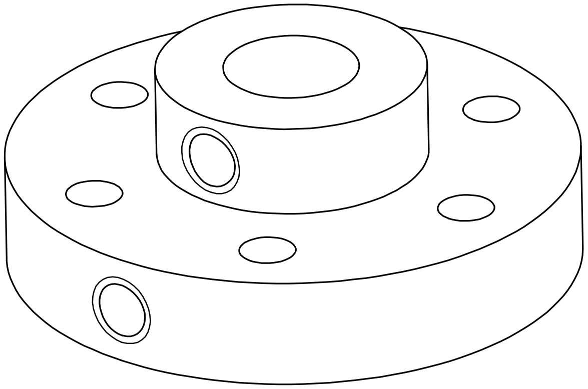
Pololu Robotics & Electronics

1. To get the specified scale, select 100% in print settings.

Scale: 1:1

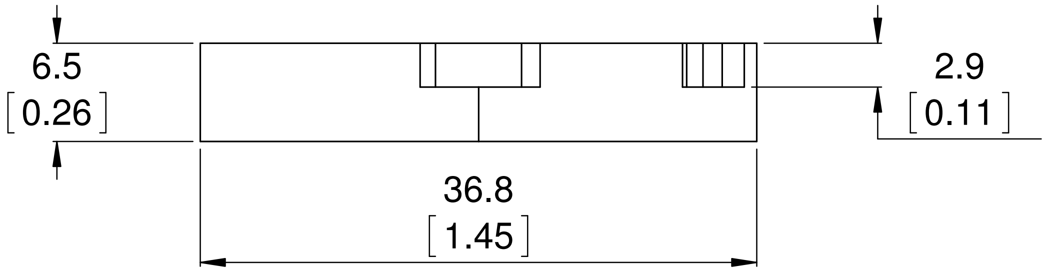


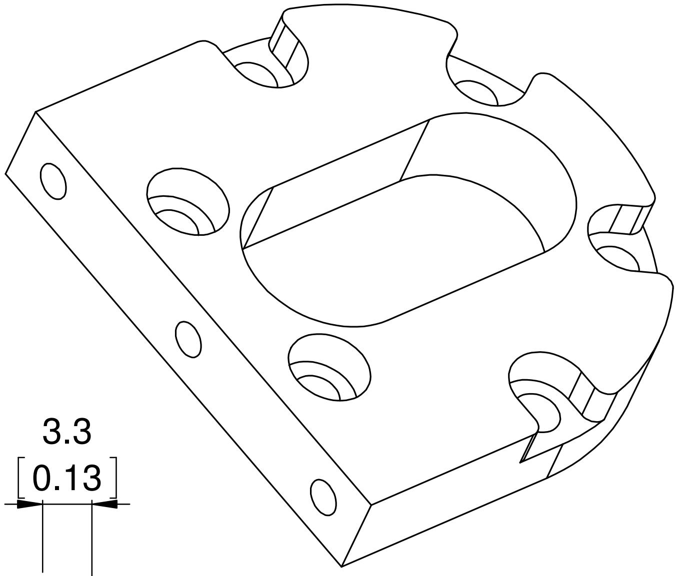


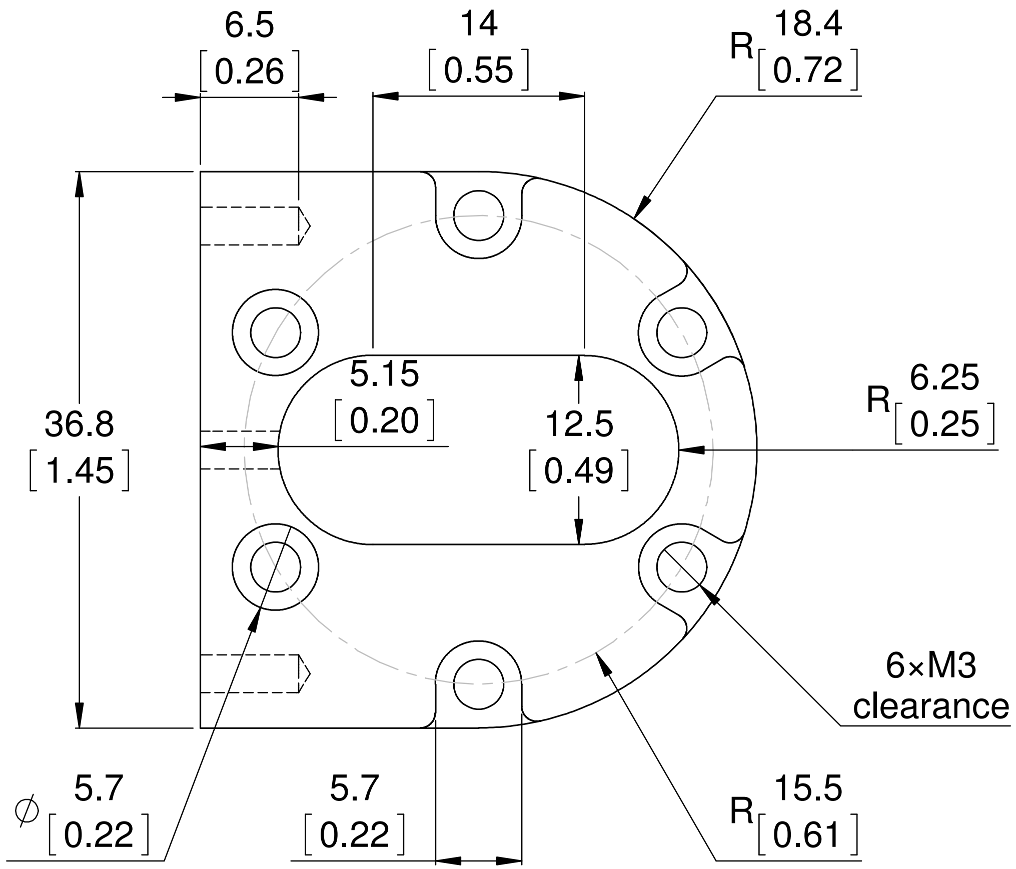


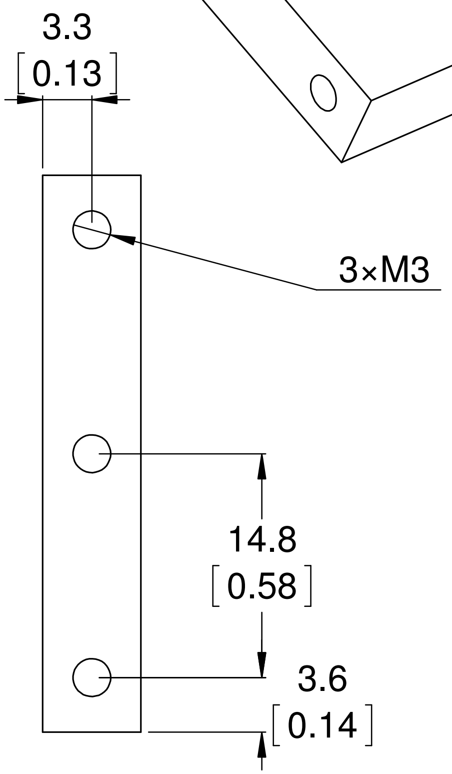
NOT TO SCALE

http://www.pololu.com/catalog/product/1083	
Name: Pololu Universal Aluminum Mounting Hub for 6mm Shaft Pair, 4-40 Holes	Item number: 1083
Drawing date: 5 August 2013	Pololu Robotics & Electronics
Units: mm [in] / Material: aluminum	









http://www.pololu.com/catalog/product/1995

Name:
Pololu Machined Aluminum Bracket
for 37D mm Metal Gearmotors

Item number:
1995

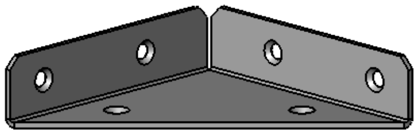
Drawing date:
2 August 2013

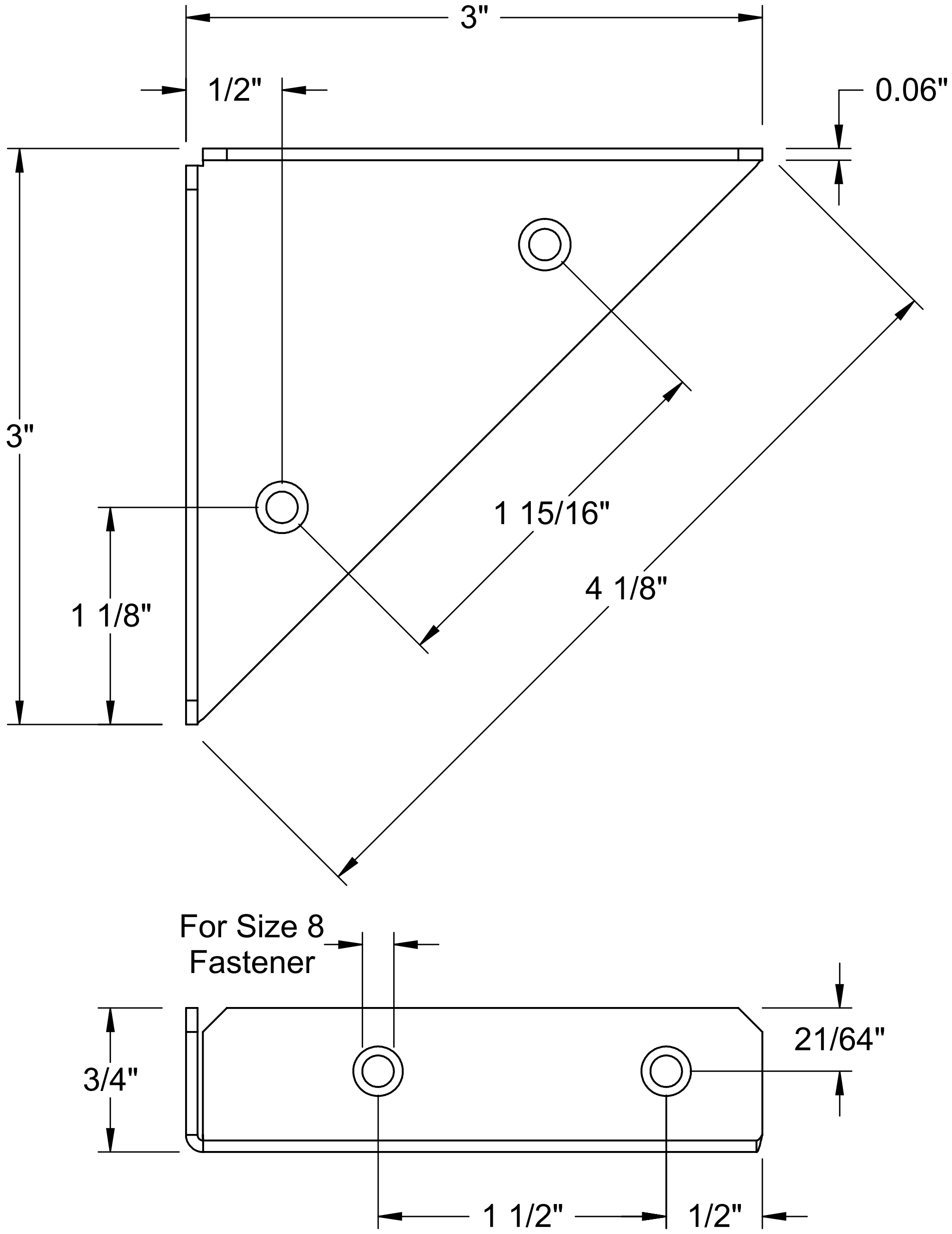
Units: mm
[in]

Material:
Aluminum

Pololu
Robotics & Electronics

NOT TO SCALE



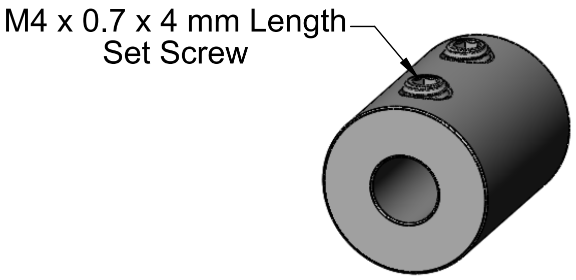


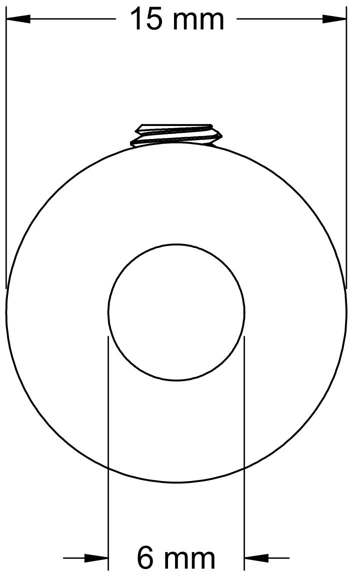
McMASTER-CARR® CAD

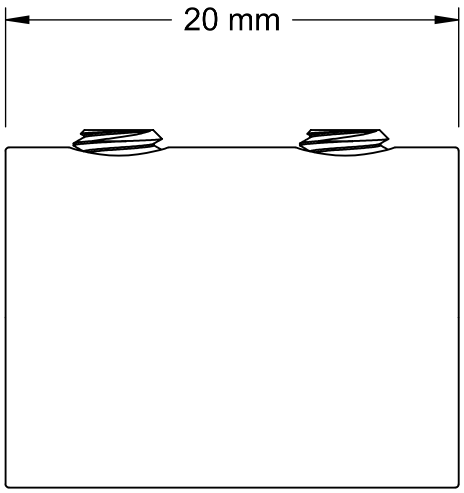
PART NUMBER **1088A32**

http://www.mcmaster.com

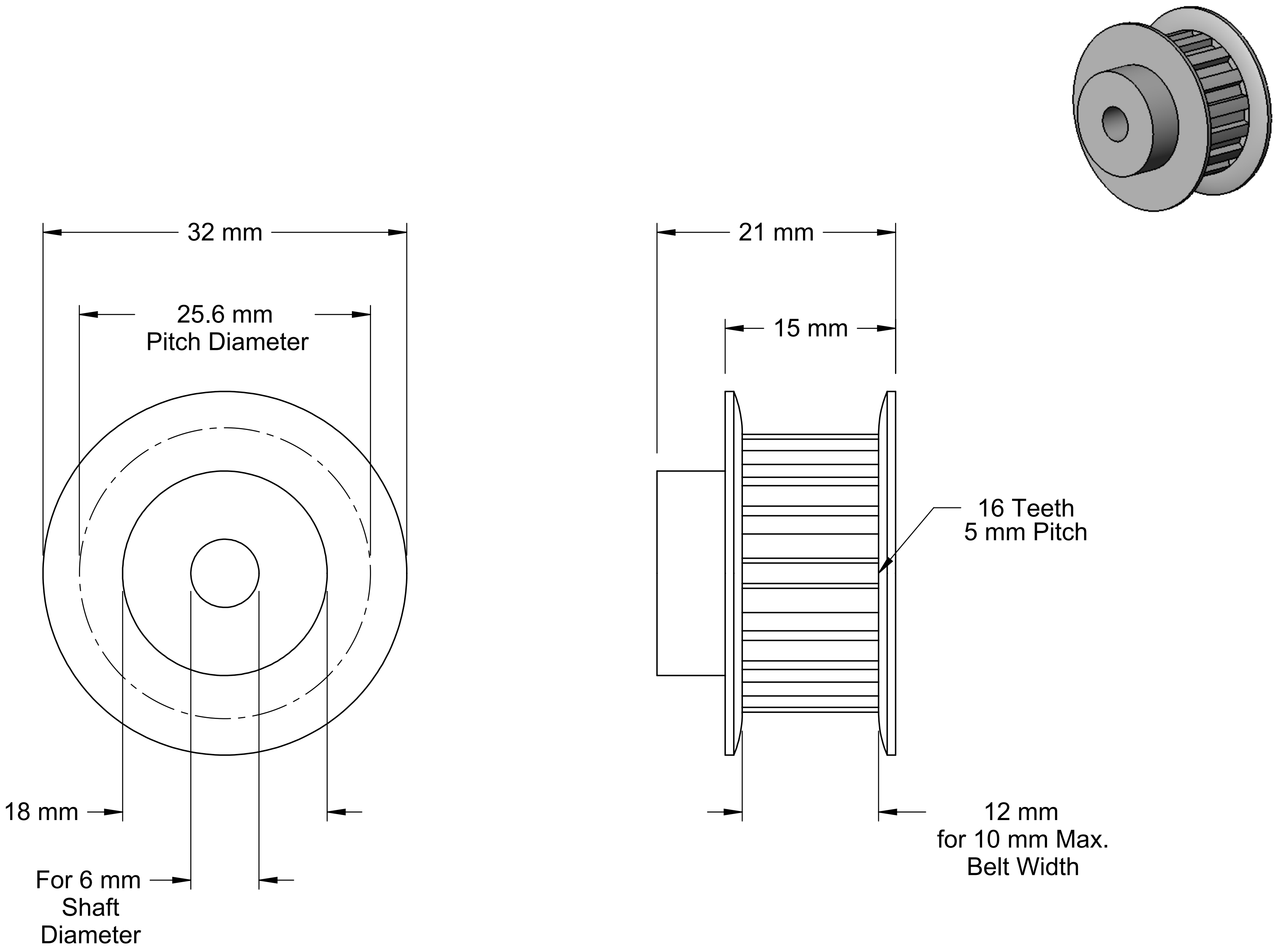
Inside-Corner Reinforcing Bracket



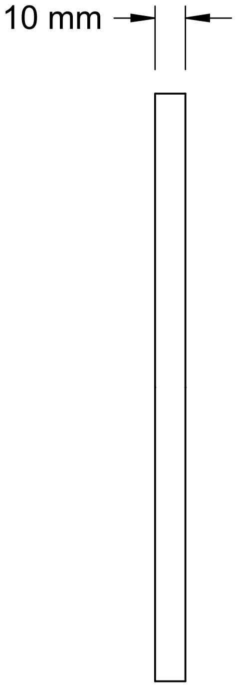


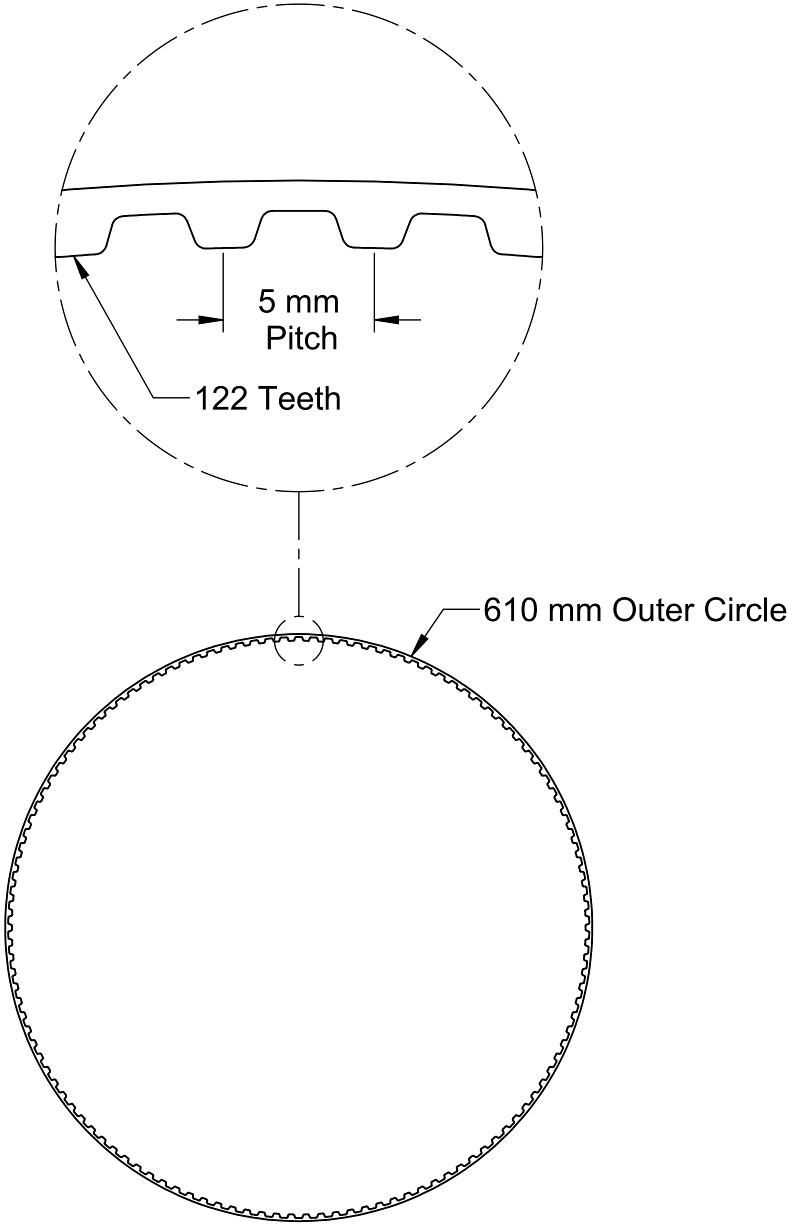


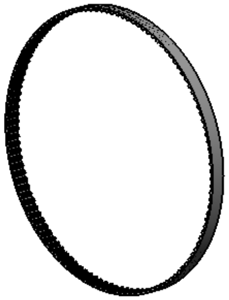
McMASTER-CARR CAD	PART NUMBER **5395T111**
http://www.mcmaster.com 	Set Screw Rigid Shaft Coupling



McMASTER-CARR® CAD	PART NUMBER	**1428N24**
http://www.mcmaster.com © 2016 McMaster-Carr Supply Company	T5 Series Timing Belt Pulley	







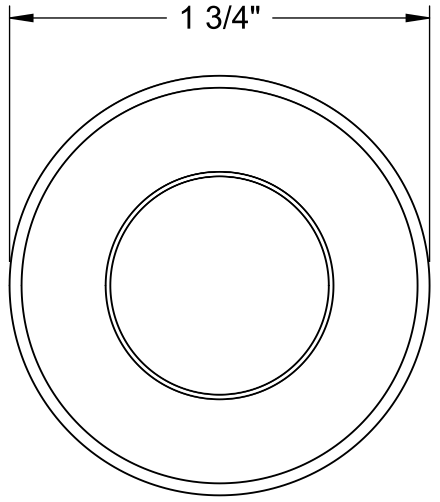
McMASTER-CARR CAD

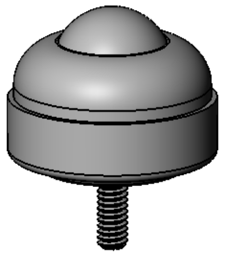
http://www.mcmaster.com

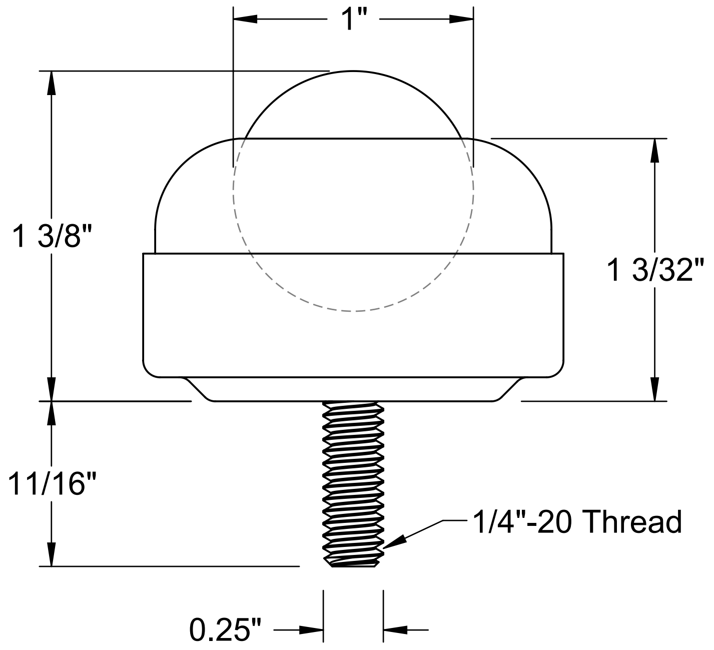
PART NUMBER **1679K544**

Trade Number T5-610-10

Dust-Free Timing Belt

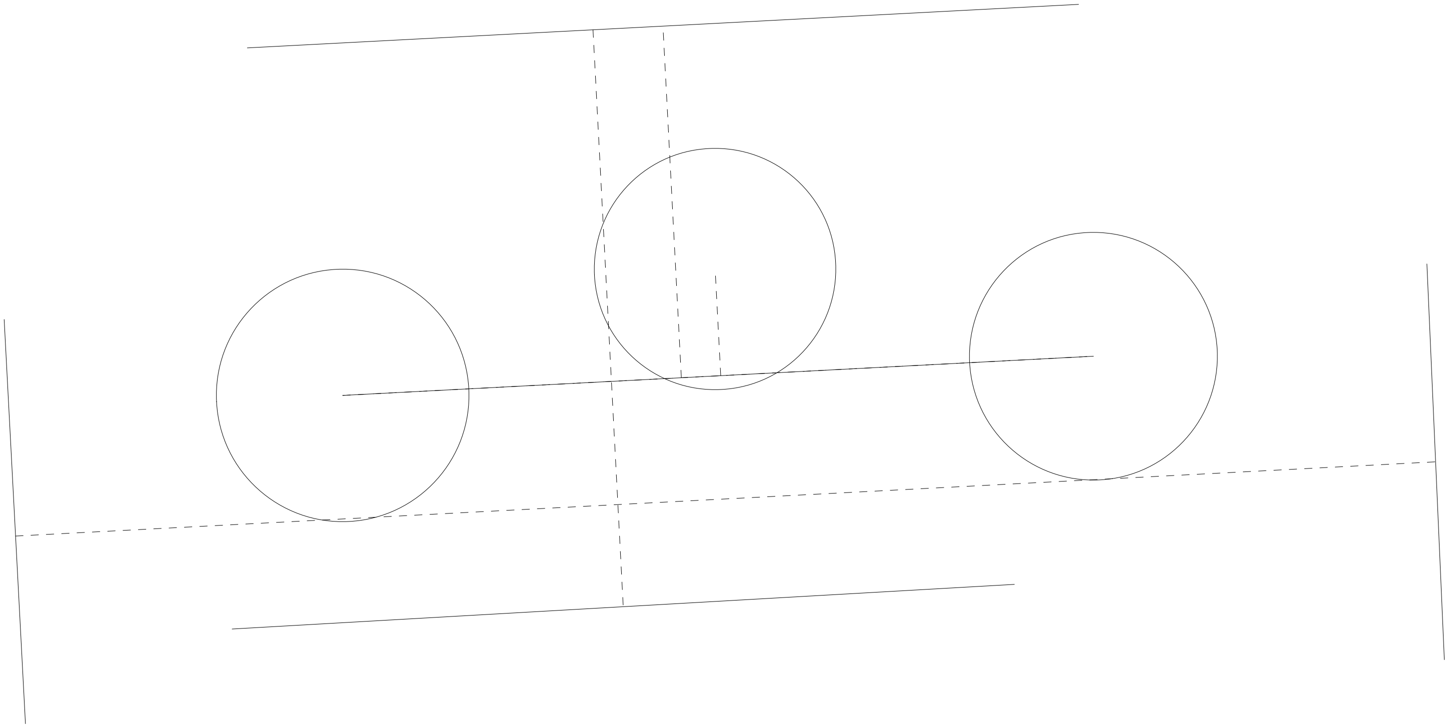






McMASTER-CARR CAD	PART NUMBER	**6460K31**
http://www.mcmaster.com © 2015 McMaster-Carr Supply Company Information in this drawing is provided for reference only.	Stud-Mount Ball Transfer	

```
Program: Untitled                                     Date: Sat May 19 2018     Time: 22:18:39
Units: in, dec deg
```



```
Feature                      Actual       Nominal        Upper        Lower      Dev/Nom      Out/Tol
-----------------------------------------------------------------------------------------------------
System 1                      [MCS]
  Origin X                   0.0000        0.0000                                 0.0000
  Origin Y                   0.0000        0.0000                                 0.0000
  Origin Z                   1.0512        1.0512                                 0.0000
  Skew                       2.8590        2.8590                                 0.0000
  Pitch                      0.0000        0.0000                                 0.0000
  Roll                       0.0000        0.0000                                 0.0000

Circle G                 [System 1]
  Center X                   6.9695        6.9695                                 0.0000
  Center Y                   6.2527        6.2527                                 0.0000
  Diameter                   0.0788        0.0788                                 0.0000

Circle H                 [System 1]
  Center X                   7.2040        7.2040                                 0.0000
  Center Y                   6.2532        6.2532                                 0.0000
  Diameter                   0.0773        0.0773                                 0.0000

Circle Center            [System 1]
  Center X                   7.0875        7.0875                                 0.0000
  Center Y                   6.2863        6.2863                                 0.0000
  Diameter                   0.0753        0.0753                                 0.0000

Distance G to H          [System 1]
  Distance X                 0.2345        0.2345                                 0.0000
  Distance Y                 0.0005        0.0005                                 0.0000
  Distance XY                0.2345        0.2345                                 0.0000
```

```
Program: Untitled                                                  Date: Sat May 19 2018     Time: 22:18:39
Units: in, dec deg

Feature                       Actual       Nominal          Upper         Lower       Dev/Nom       Out/Tol
------------------------------------------------------------------------------------------------------------

Line HG L                 [System 1]
  Direction                   0.1279        0.1279                                      0.0000

Distance HG L to Cente    [System 1]
  Distance X                  0.0001        0.0001                                      0.0000
  Distance Y                  0.0333        0.0333                                      0.0000
  Distance XY                 0.0333        0.0333                                      0.0000

Line edge                 [System 1]
  Direction                -179.8626     -179.8626                                      0.0000

Line Case Edge            [System 1]
  Direction                -179.5913     -179.5913                                      0.0000

Line G Edge               [System 1]
  Direction                 -89.8635      -89.8635                                      0.0000

Line H Edge               [System 1]
  Direction                  89.6648       89.6648                                      0.0000

Distance G to H Edge      [System 1]
  Distance X                  0.4436        0.4436                                      0.0000
  Distance Y                  0.0011        0.0011                                      0.0000
  Distance XY                 0.4436        0.4436                                      0.0000

Distance Edge to Case     [System 1]
  Distance X                  0.0004        0.0004                                      0.0000
  Distance Y                  0.1803        0.1803                                      0.0000
  Distance XY                 0.1803        0.1803                                      0.0000

Distance Edge to HG L     [System 1]
  Distance X                  0.0002        0.0002                                      0.0000
  Distance Y                  0.1099        0.1099                                      0.0000
  Distance XY                 0.1099        0.1099                                      0.0000
```

controller.Controller Class Reference

This class implements closed-loop proportional control to run as a generic controller for a Shoe of Brian purple MicroPython board that is pin connected on top with a white Nucleo L476RG board. More...

Public Member Functions

def **__init__** (self)
Constructor method which sets the proportional gain, inital setpoint, actual setpoint, error signal, actuation signal with a saturation limit. More...

def **algorithm** (self, **actual**)
Algorithm is a function that subtracts the measured parameter of the device from the desired setpoint to return an error signal, which is then multiplied by the proportional gain input value to solve for an actuation value. More...

def **set_gain** (self, gain)
This function sets the user inputed Kp value of the device to a variable named gain which represents the proportional gain of the device. More...

def **set_KI** (self, **K_I**)
This function sets the user inputed K_I value of the device to a variable named gain which represents the integral gain of the device. More...

def **set_KD** (self, **K_D**)
This function sets the user inputed K_D value of the device to a variable named gain which represents the derivative gain of the device. More...

def **set_KW** (self, **K_W**)
This function sets the user inputed K_W value of the device to a variable named gain which represents the anti_windup gain of the device. More...

def **set_setpoint** (self, point)
Method which creates lists for the actual value being measured, time, and error values to be used for plotting a step response of the device. More...

def **print_response** (self)
Method which runs step response tests by sending a signal through the USB serial port to the MicroPython board, reading the resulting actual and time data, and plotting the step response.

def **get_response** (self)

Method that takes the time values and actual measured motor position values, and puts them into a list for getting time and position response of the motor. More...

Public Attributes

K_P
Input for proportional gain of the motor.

setpoint
Desired position of the motor.

error
Difference in the setpoint of the motor from its measured position.

actuation
Signal sent to the motor to control the magnitude and direction of its torque.

actual
Measured position of the motor after the setpoint desired is inputed.

act_value

time

error_list

delta_time
Response time of motor run for one revolution (setpoint=4000) for the step response test.

delta
Last motor position measured from step response test.

accuracy
Percent difference from setpoint position and actual motor position.

error_sum
Error sum for integral control.

K_I
Integral control constant.

prev_error
Previous error used for derivative control.

d_error
Delta err for derivative control.

t

Time of control.

prev_t
Previous t of control.

dt
Delta time for derivative control.

K_D
Derivative control constant.

act_star
A* for K anti-windup.

K_W
Anti-windup control constant.

proportional

integral

derivative

prev_err

Detailed Description

This class implements closed-loop proportional control to run as a generic controller for a Shoe of Brian purple MicroPython board that is pin connected on top with a white Nucleo L476RG board.

This class has the following methods: *init*(), **algorithm()**, **set_gain()**, **set_KI()**,**set_KD()**,**set_KW()**, **set_setpoint()**, **print_response()**, **get_response()**. The constructor first sets all the necessary parameters for the controller to work. Algorithm returns an actuation value that can be generally set to anything as a generic controller. The algorithm method takes the subtraction of the setpoint parameter (motor position input for this project) and the actual measured parameter (measured motor position for this project). Set_setpoint creates arrays for specific parameters and sets the setpoint, which is the desired position for the DC motor in our case. Set_gain sets the proportional control gain for the device. Get_response and print_response are methods which run step response tests each time the enter key is pressed by the user, which reads the resulting data and prints a list of time, actual position, and error values in the serial port terminal.

This controller is setup for proportional, integral, derivative control. The anti-windup features are not completely developed.

The only anti-windup code simply puts a saturation limit of the error_sum to the duty cycle divided by the K_I.

```
The following are not actual parameters specific to the code, but are m
to describe important attributes for the Controller class.
@param K_P User input proportional gain of the device
@param setpoint Input parameter to device (desired position of motor)
@param error Error signal or the difference in setpoint from the
measured setpoint.This will be the measured motor location subtracted
from the initial setpoint location.
@param actuation Actuation signal to device as a result from the error
signal multiplied by control gain. This will be a signal sent
to the motor to control magnitude and direction of motor torque.
@param actual Measured parameter of device (motor position)
@param act_value List of measured motor positions
@param time List for how long motor has run for
@param error_list List of error values between each controller run
@param delta_time Total time elapsed for motor run at one revolution
@param delta Last actual motor position measured at the end of each
test conducted.
@param accuracy Percent difference from the setpoint and actual
values at the last motor position measured.
@param K_I Integral control constant
```

Constructor & Destructor Documentation

def controller.Controller.__init__ (self)

Constructor method which sets the proportional gain, inital setpoint, actual setpoint, error signal, actuation signal with a saturation limit.

Lists are initialized to extract actuation signal, time, and error data along with setting a change in time paramter for generating response plots of the motor over the period of time it is run for.

Member Function Documentation

def controller.Controller.algorithm (**self,**
actual
)

Algorithm is a function that subtracts the measured parameter of the device from the desired setpoint to return an error signal, which is then multiplied by the proportional gain input value to solve for an actuation value.

This actuation signal which controls the magnitude and direction of the device torque is limited to be within -100 and 100 before getting returned.

Parameters

actual The actual position of the object of interest

Returns

actuation The level to set the actuation for control.

def controller.Controller.get_response (**self)**

Method that takes the time values and actual measured motor position values, and puts them into a list for getting time and position response of the motor.

Returns

[time, act_value] Returns a list of a time and position

def controller.Controller.set_gain (**self,**
gain
)

This function sets the user inputed Kp value of the device to a variable named gain which represents the proportional gain of the device.

Parameters

gain The gain for the proportional control.

**def controller.Controller.set_KD (self,
K_D
)**

This function sets the user inputed K_D value of the device to a variable named gain which represents the derivative gain of the device.

Parameters

K_D The gain for the derivative control.

**def controller.Controller.set_KI (self,
K_I
)**

This function sets the user inputed K_I value of the device to a variable named gain which represents the integral gain of the device.

Parameters

K_I The gain for the integral control.

**def controller.Controller.set_KW (self,
K_W
)**

This function sets the user inputed K_W value of the device to a variable named gain which represents the anti_windup gain of the device.

Parameters

K_w The gain for the anti_windup control.

def controller.Controller.set_setpoint (self,
point
)

Method which creates lists for the actual value being measured, time, and error values to be used for plotting a step response of the device.

Commented lists to hold time, error, and actual position for response These are commented out for now to save memory.

Parameters

point Point to set as the setpoint.

The documentation for this class was generated from the following file:

- **controller.py**

cotask.Task Class Reference

This class implements behavior common to tasks in a cooperative multitasking system which runs in MicroPython. More...

Public Member Functions

def **__init__** (self, run_motor, **name**='NoName', **priority**=0, **period**=None, profile=False, trace=False)
Initializes a task object, saving copies of constructor parameters and preparing an empty dictionary for states. More...

def **schedule** (self)
This method is called by the scheduler; it attempts to run this task. More...

def **ready** (self)
This method checks if the task is ready to run. More...

def **reset_profile** (self)
This method resets the variables used for execution time profiling. More...

def **get_trace** (self)
This method returns a string containing the task's transition trace. More...

def **go** (self)
Method to set a flag so that this task indicates that it's ready to run. More...

def **__repr__** (self)
This method converts the task to a string for diagnostic use. More...

Public Attributes

name
The name of the task, hopefully a short and descriptive string. More...

priority
The task's priority, an integer with higher numbers meaning higher priority. More...

period
The period, in microseconds, between runs of the task's `run()` method. More...

go_flag
Flag which is set true when the task is ready to be run by the scheduler.

Detailed Description

This class implements behavior common to tasks in a cooperative multitasking system which runs in MicroPython.

The ability to be scheduled on the basis of time or an external software trigger or interrupt is implemented, state transitions can be recorded, and run times can be profiled. The user's task code must be implemented in a generator which yields the state (and the CPU) after it has run for a short and bounded period of time.

Example:

```
def task1_fun ():
    ''' Simple and silly task which just toggles its state '''
    state = 0
    while True:
        if state == 0:
            state = 1
        elif state == 1:
            state = 0
        yield (state)

# In main routine. This task runs twice per second
task1 = cotask.Task (task1_fun, name = 'Task 1', priority = 1,
                     period = 500, profile = True, trace = True)
cotask.task_list.append (task1)
while True:
    cotask.task_list.pri_sched ()
```

Constructor & Destructor Documentation

```
def cotask.Task.__init__ ( self,
                           run_motor,
                           name = 'NoName',
                           priority = 0,
                           period = None,
                           profile = False,
                           trace = False
                         )
```

Initializes a task object, saving copies of constructor parameters and preparing an empty dictionary for states.

Parameters

run_fun	The function which implements the task's code. It must be a generator which yields the current state
name	The name of the task, by default 'NoName.' This should **really** be overridden with a more descriptive name by the user
priority	The priority of the task, a positive integer with higher numbers meaning higher priority (default 0)
period	The time in milliseconds between runs of the task if it's run by a timer or `None` if the task is not run by a timer. The time can be given in a `float` or `int`; it will be converted to microseconds for internal use by the scheduler
profile	Set to `True` to enable run-time profiling
trace	Set to `True` to generate a list of transitions between states. **Note:** This slows things down and allocates memory.

Member Function Documentation

def cotask.Task.__repr__ (self)

This method converts the task to a string for diagnostic use.

It shows information about the task, including execution time profiling results if profiling has been done.

def cotask.Task.get_trace (self)

This method returns a string containing the task's transition trace.

The trace is a set of tuples, each of which contains a time and the states from and to which the system transitioned.

Returns
A possibly quite large string showing state transitions

def cotask.Task.go (self)

Method to set a flag so that this task indicates that it's ready to run.

This method may be called from an interrupt service routine or from another task which has data that this task needs to process soon.

def cotask.Task.ready (self,
bool
)

This method checks if the task is ready to run.

If the task runs on a timer, this method checks what time it is; if not, this method checks the flag which indicates that the task is ready to go. This method may be overridden in descendent classes to implement some other behavior.

def cotask.Task.reset_profile (self)

This method resets the variables used for execution time profiling.

It's also used by **`init`**() to create the variables.

def cotask.Task.schedule (self,
bool
)

This method is called by the scheduler; it attempts to run this task.

If the task is not yet ready to run, this method returns `False` immediately; if this task is ready to run, it runs the task's generator up to the next `yield()` and then returns `True`.

Returns

`True` if the task ran or `False` if it did not

Member Data Documentation

cotask.Task.name

The name of the task, hopefully a short and descriptive string.

cotask.Task.period

The period, in microseconds, between runs of the task's **`run()`** method.

If the period is None, the **`run()`** method won't be run on a time basis but will instead be run by the scheduler as soon as feasible after code such as an interrupt handler calls the **`go()`** method.

cotask.Task.priority

The task's priority, an integer with higher numbers meaning higher priority.

The documentation for this class was generated from the following file:

- **cotask.py**

cotask.TaskList Class Reference

This class holds the list of tasks which will be run by the task scheduler. More...

Public Member Functions

def	**__init__** (self) Initialize the task list. More...
def	**append** (self, task) Append a task to the task list. More...
def	**rr_sched** (self) This scheduling method runs tasks in a round-robin fashion. More...
def	**pri_sched** (self) This scheduler runs tasks in a priority based fashion. More...
def	**__repr__** (self) Create some diagnostic text showing the tasks in the task list.

Public Attributes

pri_list
The list of priority lists. More...

Detailed Description

This class holds the list of tasks which will be run by the task scheduler.

The task list is sorted by priority so that the scheduler can efficiently look through the list to find the highest priority task which is ready to run at any given time. Tasks can also be scheduled in a simpler "round-robin" fashion.

An example showing the use of the task list is given in the documentation for class **Task**.

Constructor & Destructor Documentation

def cotask.TaskList.__init__ (self)

Initialize the task list.

This creates the list of priorities in which tasks will be organized by priority.

Member Function Documentation

def cotask.TaskList.append (self,
task
)

Append a task to the task list.

The list will be sorted by task priorities so that the scheduler can quickly find the highest priority task which is ready to run at any given time.

Parameters

task The task to be appended to the list

def cotask.TaskList.pri_sched (self)

This scheduler runs tasks in a priority based fashion.

Each time it is called, it finds the next task which is ready to run and calls that task's **`run()`** method.

def cotask.TaskList.rr_sched (self)

This scheduling method runs tasks in a round-robin fashion.

Each time it is called, it goes through the list of tasks and gives each of them a chance to run. This scheduler runs the highest priority tasks first, but that's not important to a round-robin scheduler, as they are all given a chance to run each time through the list, and it takes about the same amount of time before each is given a chance to run again.

Member Data Documentation

cotask.TaskList.pri_list

The list of priority lists.

Each priority for which at least one task has been created has a list whose first element is a task priority and whose other elements are references to task objects at that priority.

The documentation for this class was generated from the following file:

- **cotask.py**

encoder.Encoder Class Reference

This class implements a quadrature encoder for a Shoe of Brian purple MicroPython board that is pin connected on top with a white Nucleo L476RG board. More...

Public Member Functions

def	**__init__** (self, timer, pin_1, pin_2) Creates a motor driver by initializing GPIO pins and gets first initial position. More...
def	**read** (self) Method for returning the correct current position of the encoder. More...
def	**zero** (self) Method for reseting the position of the ecoder to zero. More...

Public Attributes

tim
The Timer desired for the encoder with period=0xFFF, prescalar=0.

pinENa
Pin object to work with Channel 1 of quadrature encoder.

pinENb
Pin object to work with Channel 2 of quadrature encoder.

ch1

ch2

position
A class attribute for the encoder's current position.

last_pos
A class attribute for the encoder's last position.

delta
A class attribute for encoder's change in position.

read_value
A read attribute to hold the current value of encoder's position.

Detailed Description

This class implements a quadrature encoder for a Shoe of Brian purple MicroPython board that is pin connected on top with a white Nucleo L476RG board.

To create an instance of class **Encoder**, see the following example. Class methods are: read(self) Returns the motors current position zero(self) Zeros the motor's position

Limited to Channel 1 and 2.

Constructor & Destructor Documentation

def encoder.Encoder.__init__ (self,
timer,
pin_1,
pin_2
)

Creates a motor driver by initializing GPIO pins and gets first initial position.

Ensure that the timer and pins used correspond to 1, Where the encoder is connected to the board and 2. Timer works for those pins. See Table 17 To create an instance of class **Encoder**. See the following example.

EX:

```
Encoder_1 = Encoder(8,'PC6','PC7')
```

Creating an instance of **Encoder** called Encoder_1 on Timer 8 and connected to the board in pins C6 and C7.

Parameters

timer The timer wanted to be used.
pin_1 The first pin on the board for encoder Ch A.
pin_2 The second pin on the board for encoder Ch B.

Member Function Documentation

def encoder.Encoder.read (self)

Method for returning the correct current position of the encoder.

Returns

position The current position of the encoder

def encoder.Encoder.zero (self)

Method for reseting the position of the ecoder to zero.

Zeros position

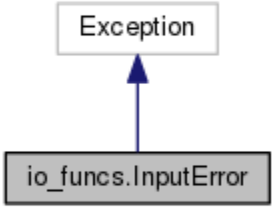
The documentation for this class was generated from the following file:

- encoder.py

io_funcs.InputError Class Reference

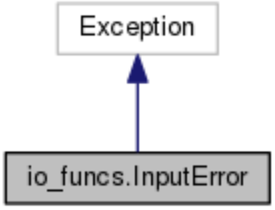
This is a custom exception error for incorrect user input. More...

Inheritance diagram for io_funcs.InputError:



[legend]

Collaboration diagram for io_funcs.InputError:



[legend]

Public Member Functions

def **__init__** (self, message, **errors**)
This method initializes the input error. More...

Public Attributes

errors
Call the base class constructor with the parameters it needs. More...

Detailed Description

This is a custom exception error for incorrect user input.

Constructor & Destructor Documentation

```
def io_funcs.InputError.__init__ ( self,
                                   message,
                                   errors
                                 )
```

This method initializes the input error.

Parameters

message	The message you want to display to get input
errors	The errors for a particular input error

Member Data Documentation

io_funcs.InputError.errors

Call the base class constructor with the parameters it needs.

Errors for a particular **InputError**

The documentation for this class was generated from the following file:

- **io_funcs.py**

main.py File Reference

This is the main file for the Lab 1 that contains the code to create a class Encoder. More...

Functions

def	**main.servo_func** () Servo task function. More...
def	**main.command_func** () This is the main command function. More...

Variables

int	**main.lift** = 30
int	**main.tolerance** = 20
	main.pen_servo = **servo.Servo**('PA5',prescaler=4.5, freq=25, min_us=665, max_us=2360, angle=190)
bool	**main.pen_cal** = True
int	**main.n** = 0
	main.answer = io_funcs.get_input(str,'Calibrated? [y/n] ')
	main.angle = io_funcs.get_input(int,'Angle? [degrees] ')
	main.pen_angle = angle
	main.down_angle = angle
	main.up_angle = down_angle+lift
bool	**main.file_search** = True
	main.file_name = io_funcs.get_input(str,'File name? [file.txt] ')
	main.file = open(file_name,'r')
	main.motor_1_task = **motor_task.Motor_control_task**(0)
string	**main.mname1** = 'Motor_'
	main.motor_2_task = **motor_task.Motor_control_task**(1)
string	**main.mname2** = 'Motor_'
bool	**main.cal** = True
	main.position
	main.actual

	main.setpoint
bool	**main.run_wait** = True
	main.servo_task
	main.command_task
string	**main.servo_state** = ''
	main.vcp = pyb.USB_VCP()
bool	**main.end** = False

Detailed Description

This is the main file for the Lab 1 that contains the code to create a class Encoder.

This is the main file that runs the pen plotter.

The class Encoder can read the current position and also zero the position.

Authors

Sam Lee and Dima Kyle

Function Documentation

def main.command_func ( )

This is the main command function.

It takes in a two motor task instances and a file, a servo_state to trigger servo state changes, and an end variable (not used).

It takes in a file and reads it line by line. Parses it by the command. The commands are the main states of this task. There are 5 main states: NEXT, IN, PU, PD, SP. IN and SP are neglected. In NEXT, the next line of the file is read and parsed to get the next command and maybe points. PU brings the pen up after reaching a setpoint PD brings the motor to a point, brings the pen down, and then traces the following points.

def main.servo_func ( )

Servo task function.

This function has 2 (3 including a done state). The three states are Up, Down, and Done. servo_state is what causes the state to change This function receives a class instance of a servo and the angles that correspond with down and up.

Variable Documentation

main.command_task

Initial value:

```
= cotask.Task(command_func, name = 'Command Task', priority=2,
                             period = 50, profile = True)
```

main.servo_task

Initial value:

```
= cotask.Task(servo_func, name = 'Servo Task', priority=1,
                             period = 50, profile = True)
```

motor_sam_dima.MotorDriver Class Reference

This implements a DC motor at a frequency of 2000 Hz for the Shoe of Brian purple MicroPython board that is pin-connected on top with a white Nucleo L476RG board. More...

Public Member Functions

def **__init__** (self, timer, pin_1, pin_2, pin_3)
Creates a motor driver by initializing GPIO pins and turning the motor off for safety. More...

def **get_duty_cycle** (self)
This function simply returns the duty cycle of the motor. More...

def **set_duty_cycle** (self, level)
This method sets the duty cycle to be sent to the motor to the given level. More...

Public Attributes

pinEN
Open-drain output pin set high to enable the DC motor.

pinIN1
Regular push-pull output pin set to low and configured with af=2 to control the direction of PWM signal sent to the motor. More...

pinIN2
Regular push-pull output pin set to high and configured with af=2 to power the motor in one direction.

Hz
The desired frequency of the pulse in the pulse width modulation.

tim
The Timer wanted for the motor at a specified frequency.

ch1

ch2

duty_cycle

Detailed Description

This implements a DC motor at a frequency of 2000 Hz for the Shoe of Brian purple MicroPython board that is pin-connected on top with a white Nucleo L476RG board.

Class methods are: set_duty_cycle(level) **get_duty_cycle()** ==> returns the duty_cycle of the motor

Limited to Timers 3 and 5

Constructor & Destructor Documentation

def motor_sam_dima.MotorDriver.__init__ (self,
timer,
pin_1,
pin_2,
pin_3
)

Creates a motor driver by initializing GPIO pins and turning the motor off for safety.

We will be using DC motors that will be powered with 12 volts and a 0.5 amp current limit by connecting power from a benchtop supply to the motor driver board with the Gnd and Vin screw terminals. To program a **MotorDriver** class, a USB cable is connected to the bottom MicroPython board and a DC motor is connected to the Motor A or B screw terminals in the driver board. The ST Microelectronics L6206 dual H-bridge motor driver chip datasheet was referenced. The link to the data sheet can be found on page 2, Figure 2 from the following link.

L6206 Datasheet: https://www.google.com/search?q=ST+Microelectronics+L6206+dual+H-bridge+motor+driver+chip&oq=st+micro&aqs=chrome.1.69i57j69i59j0l4.3671j0j7&sourceid=chrome&ie=UTF-8

From the diagram, the motor is connected to pins OUT1A and OUT2A. The microcontroller controls pins ENA, IN1A, and IN2A

To properly initialize an instance of **MotorDriver**, refer to the example below. EX:

```
motor_1 = MotorDriver(3,'PA10','PB4','PB5')
```

This makes an instance of **MotorDriver** using Timer 3, PA10 enables the motor and PB4 and PB5 are used control the motor in one particular direction.

Parameters

timer	Timer to be used for the motor
pin_1	First pin to enable the motor. PinEn is to be the output pin at pin_1.
pin_2	Second pin for IN1 direction 1. PinIN1 is to be the output pin at pin_1.
pin_3	Third pin for IN2 direction 2. PinIN2 is to be the output pin at pin_2.

Member Function Documentation

def motor_sam_dima.MotorDriver.get_duty_cycle (self)

This function simply returns the duty cycle of the motor.

Returns

duty_cycle The duty cycle of the motor as a percentage

def motor_sam_dima.MotorDriver.set_duty_cycle (self,
level
)

This method sets the duty cycle to be sent to the motor to the given level.

Positive values cause torque in one direction, negative values in the opposite direction.

Parameters

level A signed integer holding the duty cycle of the motor (%)

Member Data Documentation

motor_sam_dima.MotorDriver.pinIN1

Regular push-pull output pin set to low and configured with af=2 to control the direction of PWM signal sent to the motor.

The documentation for this class was generated from the following file:

- **motor_sam_dima.py**

motor_task.Motor_control_task Class Reference

Class which contains a motor task function. More...

Public Member Functions

def **__init__** (self, motor_num)
This constructor method initializes two instances of DC motors and two quadruture encoders. More...

def **run_motor** (self)
Motor task function consisting of two states. More...

Public Attributes

control

motor

encoder

motor_number
Motor number which specifies which motor task is being run.

state
Motor control task to start in state 0 to run the motors with data.

position
Position of the motor initially.

iterate
Initial setpoint. More...

limit
Limit on the amount of iterations the motor is outputting position data for. More...

actuation

Detailed Description

Class which contains a motor task function.

This task initializes two instances of DC motors and quadruture encoders to be used and has two states to run the motor with the necessary data for finding the motor's position for one state, and another state to run the motor without any data. In the **main.py** file, a for loop is run for each motor number, where the period is set for each motor. From this for loop, both motors have an instance of the same motor controller task.

There is a method **run_motor()** to run the motor's in a scheduler.

Constructor & Destructor Documentation

def motor_task.Motor_control_task.__init__ (self, motor_num)

This constructor method initializes two instances of DC motors and two quadruture encoders.

Additionally, the optimal proportional gain of Kp is set for each motor. Both encoder positions are then zeroed, and the setpoint is set for the encoder ticks. Lastly, all variables used for the motor task function run_motor are initialized, including motor_num, state, position, iterate, and limit.

Parameters

motor_number	Motor number parmater that specifies which motor and encoder is being initialized for each task.

These are values that can be changed in the code itself.

Parameters

state	The state for which the motor control task is in.
position	Initializing the motor position to start at 0.
iterate	Initializing the iterate variable to start at 0.
limit	Limit on the amount of iterations the motor is running position data for.

The KP, KI, and KD can be changed for the situation required.

Member Function Documentation

def motor_task.Motor_control_task.run_motor (self)

Motor task function consisting of two states.

The first state runs the motors with data for a specific amount of iterations. Once the amount of iterations have reached a specific limit, then the task will go into the next state 1 which runs the motors without any data. The motor_number, position and actuation values are then printed before the state is yielded.

Member Data Documentation

motor_task.Motor_control_task.iterate

Initial setpoint.

Iteration limit for outputting data

motor_task.Motor_control_task.limit

Limit on the amount of iterations the motor is outputting position data for.

The documentation for this class was generated from the following file:

- **motor_task.py**

parse_hpgl.py File Reference

This program takes in hpgl file and parses it into a list of commands and parses it into a list of commands and positions. More...

Functions

def	**parse_hpgl.parse_file** (file_name, res, state=0, CPR=0, L1=0, L2=0, x_0=0, y_0=0) Takes in a file name for a hpgl file and parses it into a list. More...
def	**parse_hpgl.pair_split** (iterable) A quick function to split a list and pair up elements in a list. More...
def	**parse_hpgl.output_text** (hpgl, file_name) A function to output to a text file. More...
def	**parse_hpgl.coord_to_ticks** (coords, CPR, L1, L2, x_0, y_0, pre_tick, pre_angle) Converts coordinates into ticks for an encoder, particularly for a coaxial 2DOF pen plotter. More...

Variables

parse_hpgl.file = sys.argv[1]

parse_hpgl.output = sys.argv[2]

parse_hpgl.res = int(sys.argv[3])

parse_hpgl.x = parse_file(file,res)

parse_hpgl.CPR = int(sys.argv[4])

parse_hpgl.L1 = float(sys.argv[5])

parse_hpgl.L2 = float(sys.argv[6])

parse_hpgl.x_0 = float(sys.argv[7])

parse_hpgl.y_0 = float(sys.argv[8])

Detailed Description

This program takes in hpgl file and parses it into a list of commands and parses it into a list of commands and positions.

It takes only hpgl files with paths in them. It returns a list with the commands and coordinates for relevant commands. The more high resolution of the x,y coordinates the better.

There is also a command for writing all the commands to a txt file as well as converting the coordinates into units to inches.

This python file can be run with the system args of the input file, output file and the resolution of the hpgl file.

The file can be run like this:

```
python parse_hpgl.py drawing.hpgl print.txt 1016
```

where the first argument is the input file, second the output, and the last the resolution.

If the coordinates need to be parsed in encoder ticks. Use like this:

```
python parse_hpgl.py a.hpgl a.txt 5 3200 8.11 10.08 0.5 14
```

The arugments for this are the hpgl file to be parsed, the output text file, the resolution, the CPR of the motors, the length of arm 1, length of arm 2 the x_0 of the paper space, and lastly the y_0 origin of the paper space.

There may be an error in the coord to ticks function

Author

Samuel Lee

Copyright

Samuel Lee

Function Documentation

def parse_hpgl.coord_to_ticks (**coords,**
CPR,
L1,
L2,
x_0,
y_0,
pre_tick,
pre_angle
)

Converts coordinates into ticks for an encoder, particularly for a coaxial 2DOF pen plotter.

Parameters

CPR	Counts of ticks per one revolution of the output shaft
L1	Length of arm 1 [in]
L2	Length of arm 2 [in]
x_0	x orign of the paper space in respect to global fram [in]
y_0	y orign of the paper space in respect to global fram [in]
pre_angle	the inital angle the plotter begins [degrees]

Returns

tick_list List of tick pairs, nested list with ticks

def parse_hpgl.output_text (**hpgl,**
file_name
)

A function to output to a text file.

Parameters

hpgl	A list of commands from parsed_list
file_name	The output file name, extension '.txt' file must be included.

def parse_hpgl.pair_split (iterable)

A quick function to split a list and pair up elements in a list.

This is particular to a list of floats and returns the coordinates as tuples

Parameters

iterable A list of floats of paired x,y coordinates (x1,y1,x2,y2)

Returns

list_of_pairs Returns a list of list of the coordinates

```
def parse_hpgl.parse_file ( file_name,
                            res,
                            state = 0,
                            CPR = 0,
                            L1 = 0,
                            L2 = 0,
                            x_0 = 0,
                            y_0 = 0
                          )
```

Takes in a file name for a hpgl file and parses it into a list.

A raw hpgl file has text that looks as follows:

IN;SP1;PU0,0;PD0,90;PU487,751;PD492,749

the first two letters determines the code command. Each command has a certain amount of parameters thereafter that represent a particular setting or position.

Each command is separated by a ';'

For more information on hpgl code refer to **http://www.isoplotec.co.jp/HPGL/eHPGL.htm**

It returns a list with elements that look like this:

Units are in inches.

MAKE SURE THE RESOLUTION IS THE CORRECT. Default resolution in most hpgl code is 1016.

The first entry is tne command, second is the number of points for that command, and the rest are the position coordinates.

['IN;1; 0x0'] ['SP;1; 0x0'] ['PU;1;', '1183x327'] ['PD;1;', '1183x710'] ['PU;1;', '1175x701'] ['PD;14;', '1175x709',... ['PU;1;', '1183x238'] ['SP;1; 0x0'] ['IN;1; 0x0']

If state = 1 and all the correct arguments are supplied, it will convert the hpgl into encoder ticks needed to draw the picture.

Parameters

file_name	The hpgl file name 'names.hpgl'
resolution	The resolution of the hpgl file in dpi.
state	0 is for convert to inches, 1 to change to encoder ticks
CPR	Counts of ticks per one revolution of the output shaft
L1	Length of arm 1 [in]
L2	Length of arm 2 [in]
x_0	x orign of the paper space in respect to global fram [in]

Note

Origin is top left corner of paper

Parameters

y_0	y orign of the paper space in respect to global fram [in]

Returns

parsed_list List of command, nested list with command & parameters

plot.py File Reference

This file is Homework 0. More...

Variables

string	**plot.file_name** = 'plot.csv'
	plot.file = open(file_name,'r')
list	**plot.eliminate** = ['', ' ','\t']
list	**plot.x** = []
list	**plot.y** = []
	plot.lines = file.readlines()
list	**plot.final_points** = []
string	**plot.line_string** = ''
	plot.points = line_string.split(',')
list	**plot.list_of_points** = []

Detailed Description

This file is Homework 0.

It takes in a csv file called eric.csv. It will take only the two first columns of data and plot it. Other things will be ignored or deleted.

The order it works is:

1. Open file
2. Read all the lines
3. Close the file
4. For every line If the line has no numbers, ignore For character in line If it is a digit, comma, or period, it stays, else skip Split the left over string with the commas If there are less than 2 points skip If each element is a number, add to data points temp list Another check to make sure there are more than two data points If less than two, skip Else, add the first two numbers to the x and y data list

This opens to read a file called 'plot.csv' for plotting.

servo.Servo Class Reference

A class for controlling the position of a servo. More...

Public Member Functions

def	**__init__** (self, **pin**, prescaler=4.5, freq=50, min_us=665, max_us=2360, angle=190)
def	**write_us** (self, us) setting the duty cycle for the servo to control its position. More...
def	**write_angle** (self, degrees=None, radians=None) Move to the specified angle in `degrees` or `radians`. More...
def	**read_servo** (self) This function is not yet developed but is aiming to be able to read the timer.

Public Attributes

tim

The Timer desired for the servo at a specified frequency.

pin

Output pin used to control the servo.

ch2

Channel used to initialize the servo for PWM.

min_us

max_us

us

freq

angle

prescaler

t_freq

read

conversion

servo_pos

Detailed Description

A class for controlling the position of a servo.

This code was referenced from the following link below.

Reference: https://bitbucket.org/thesheep/micropython-servo/src

Class methods are: write_us(us) ==> sets the servo duty cycle write_angle(degrees) ==> solves for servo signal in microseconds from user input angle.

To Properly initialize an instance of **Servo**, refer to the example below Ex:

```
servo_1 = Servo('PA5')
```

This makes an instance of **Servo** using Timer 2 on pin A5 of the white Nucleo L476RG board that is pin connected on top of the Shoe of Brian purple MicroPython board.

```
Parameters:
@param pin (machine.Pin): The pin where servo is connected. Must suppor
@param prescaler: allow the timer to be clocked at the rate a user desi
@param freq (int): The frequency of the signal, in hertz.
@param min_us (int): The minimum signal length supported by the servo.
@param max_us (int): The maximum signal length supported by the servo.
@param angle (int): The angle between the minimum and maximum positions

All of these parameters can be found on the servo's datasheet linked be
HS-65MG Servo Datasheet: https://www.servocity.com/hs-65mg-servo
```

Member Function Documentation

def servo.Servo.write_angle (**self,**
degrees = None,
radians = None
)

Move to the specified angle in degrees or radians.

Solves for and returns the signal length of the servo

def servo.Servo.write_us (self,
us
)

setting the duty cycle for the servo to control its position.

Returns the signal length of the servo in microseconds, frequency (Hz). period (Sec), and the percent duty cycle being sent

Parameters

us The current signal length of the servo.

The documentation for this class was generated from the following file:

- **servo.py**

task_share.Queue Class Reference

This class implements a queue which is used to transfer data from one task to another. More...

Public Member Functions

def	**__init__** (self, type_code, size, thread_protect=True, overwrite=False, name=None) Initialize a queue by allocating memory for the contents and setting up the components in an empty configuration. More...
def	**put** (self, item, in_ISR=False) Put an item into the queue. More...
def	**get** (self, in_ISR=False) Read an item from the queue. More...
def	**any** (self) Returns `True` if there are any items in the queue and `False` if the queue is empty. More...
def	**empty** (self) Returns `True` if there are no items in the queue and `False` if there are any items therein. More...
def	**full** (self) This method returns `True` if the queue is already full and there is no room for more data without overwriting existing data. More...
def	**num_in** (self) This method returns the number of items which are currently in the queue. More...
def	**__repr__** (self) This method puts diagnostic information about the queue into a string. More...

Static Public Attributes

int	**ser_num** = 0 A counter used to give serial numbers to queues for diagnostic use. More...

Detailed Description

This class implements a queue which is used to transfer data from one task to another.

If parameter 'thread_protect' is `True`, the transfer will be protected from corruption in the case that one thread might interrupt another due to threading or due to one thread being run as an interrupt service routine.

Constructor & Destructor Documentation

def task_share.Queue.__init__ (**self,**
type_code,
size,
thread_protect = `True`,
overwrite = `False`,
name = `None`
)

Initialize a queue by allocating memory for the contents and setting up the components in an empty configuration.

The data type code is given as for the Python 'array' type, which can be any of

- b (signed char), B (unsigned char)
- h (signed short), H (unsigned short)
- i (signed int), I (unsigned int)
- l (signed long), L (unsigned long)
- q (signed long long), Q (unsigned long long)
- f (float), or d (double-precision float)

Parameters

type_code	The type of data items which the queue can hold
size	The maximum number of items which the queue can hold
thread_protect	`True` if mutual exclusion protection is used
overwrite	If `True`, oldest data will be overwritten with new data if the queue becomes full
name	A short name for the queue, default `QueueN` where `N` is a serial number for the queue

Member Function Documentation

def task_share.Queue.__repr__ (**self)**

This method puts diagnostic information about the queue into a string.

def task_share.Queue.any (self)

Returns `True` if there are any items in the queue and `False` if the queue is empty.

Returns

`True` if items are in the queue, `False` if not

def task_share.Queue.empty (self)

Returns `True` if there are no items in the queue and `False` if there are any items therein.

Returns

`True` if queue is empty, `False` if it's not empty

def task_share.Queue.full (self)

This method returns `True` if the queue is already full and there is no room for more data without overwriting existing data.

Returns

`True` if the queue is full

def task_share.Queue.get (self,
in_ISR = False
)

Read an item from the queue.

If there isn't anything in there, wait (blocking the calling process) until something becomes available. If non-blocking reads are needed, one should call **`any()`** to check for items before attempting to read any items.

Parameters

in_ISR Set this to `True` if calling from within an ISR

def task_share.Queue.num_in (self)

This method returns the number of items which are currently in the queue.

Returns

The number of items in the queue

def task_share.Queue.put (self, item, in_ISR = False)

Put an item into the queue.

If there isn't room for the item, wait (blocking the calling process) until room becomes available, unless the `overwrite` constructor parameter was set to `True` to allow old data to be clobbered. If non-blocking behavior without overwriting is needed, one should call **`full()`** to ensure that the queue is not full before putting data into it.

Parameters

item	The item to be placed into the queue
in_ISR	Set this to `True` if calling from within an ISR

Member Data Documentation

int task_share.Queue.ser_num = 0 static

A counter used to give serial numbers to queues for diagnostic use.

The documentation for this class was generated from the following file:

- **task_share.py**

task_share.Share Class Reference

This class implements a shared data item which can be protected against data corruption by pre-emptive multithreading. More...

Public Member Functions

def **__init__** (self, type_code, thread_protect=True, name=None)
Allocate memory in which the shared data will be buffered. More...

def **put** (self, data, in_ISR=False)
Write an item of data into the share. More...

def **get** (self, in_ISR=False)
Read an item of data from the share. More...

def **__repr__** (self)
This method puts diagnostic information about the share into a string. More...

Static Public Attributes

int **ser_num** = 0
A counter used to give serial numbers to shares for diagnostic use. More...

Detailed Description

This class implements a shared data item which can be protected against data corruption by pre-emptive multithreading.

Multithreading which can corrupt shared data includes the use of ordinary interrupts as well as the use of a Real-Time Operating System (RTOS).

Constructor & Destructor Documentation

def task_share.Share.__init__ (self,
type_code,
thread_protect = True,
name = None
)

Allocate memory in which the shared data will be buffered.

The data type code is given as for the Python 'array' type, which can be any of

- b (signed char), B (unsigned char)
- h (signed short), H (unsigned short)
- i (signed int), I (unsigned int)
- l (signed long), L (unsigned long)
- q (signed long long), Q (unsigned long long)
- f (float), or d (double-precision float)

Parameters

type_code	The type of data items which the share can hold
thread_protect	True if mutual exclusion protection is used
name	A short name for the share, default `ShareN` where `N` is a serial number for the share

Member Function Documentation

def task_share.Share.__repr__ (self)

This method puts diagnostic information about the share into a string.

def task_share.Share.get (**self,**
in_ISR = False
)

Read an item of data from the share.

Interrupts are disabled as the data is read so as to prevent data corruption by changes in the data as it is being read.

Parameters

in_ISR	Set this to True if calling from within an ISR

def task_share.Share.put (**self,**
data,
in_ISR = False
)

Write an item of data into the share.

Any old data is overwritten. This code disables interrupts during the writing so as to prevent data corrupting by an interrupt service routine which might access the same data.

Parameters

data	The data to be put into this share
in_ISR	Set this to True if calling from within an ISR

Member Data Documentation

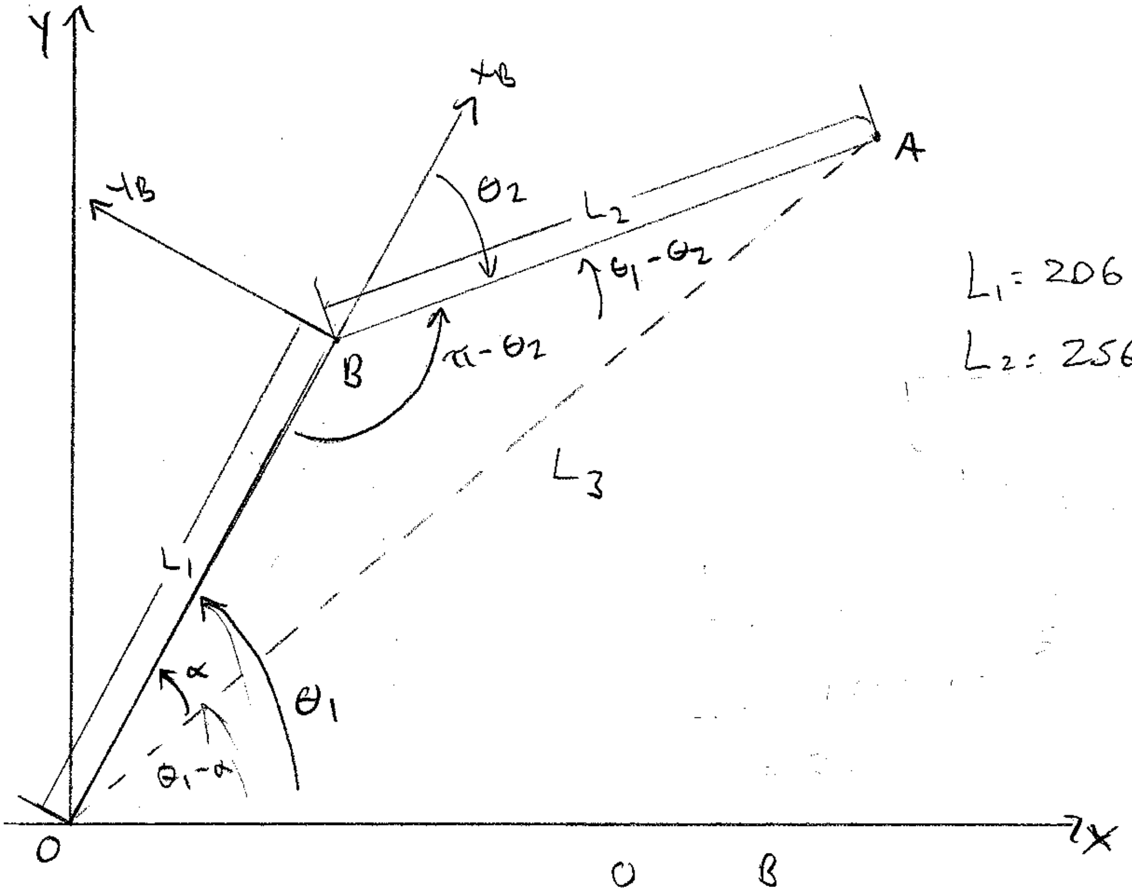
int task_share.Share.ser_num = 0 static

A counter used to give serial numbers to shares for diagnostic use.

The documentation for this class was generated from the following file:

- **task_share.py**

5/15/18 LAB-1 KINEMATICS ME-405-04 S. LEE



$L_1 = 206$ mm $= 8.1102$ in

$L_2 = 256$ mm $= 10.0787$ in

$L_1 \equiv$ LENGTH OF ARM 1 ($\overset{O}{\text{CENTER}}$ − $\overset{B}{\text{CENTER}}$)

$L_2 \equiv$ LENGTH OF ARM 2 ($\overset{B}{\text{CENTER}}$ − $\overset{A}{\text{PEN POINT}}$)

$L_3 \equiv$ DISTANCE FROM $\overset{O}{\text{ORIGIN}}$ TO $\overset{A}{\text{PEN POINT}}$

Y, X_B ALIGNED WITH ARM 1

FROM
LAW OF COSINES

$$\boxed{L_3^2 = L_1^2 + L_2^2 - 2L_1L_2\cos(\pi-\theta_2)}$$

$$x_p = L_3\cos(\theta_1-\alpha)$$

$$y_p = L_3\sin(\theta_1-\alpha)$$

$$L_3 = \sqrt{x_p^2 + y_p^2}$$

$$x_p^2 + y_p^2 = L_1^2 + L_2^2 - 2L_1L_2\cos(\pi-\theta_2)$$

$$\frac{L_3^2 - L_1^2 - L_2^2}{-2L_1L_2} = \cos(\pi - \theta_2)$$

$$\cos(\pi - \theta_2) = \frac{L_3^2 - L_1^2 - L_2^2}{-2L_1L_2}$$

$$\pi - \theta_2 = \cos^{-1}\left(\frac{L_3^2 - L_1^2 - L_2^2}{-2L_1L_2}\right)$$

$$\boxed{\theta_2 = \pi - \cos^{-1}\left(\frac{L_3^2 - L_1^2 - L_2^2}{-2L_1L_2}\right)} \Rightarrow \boxed{\theta_2 = \pi - \cos^{-1}\left(\frac{L_1^2 + L_2^2 - L_3^2}{2L_1L_2}\right)}$$

$$X_p = L_1\cos\theta_1 + L_2\cos(\theta_1 - \theta_2)$$
$$Y_p = L_1\sin\theta_1 + L_2\sin(\alpha_1 - \theta_2)$$

$$L_3 = \sqrt{x_p^2 + y_p^2}$$

$$\theta_1 - \alpha = \tan^{-1}\frac{y_p}{x_p}$$

Or $\theta_1 = \tan^{-1}\left(\frac{y_p}{x_p}\right) + \alpha$

LAW OF COSINES

$$L_2^2 = L_1^2 + L_3^2 - 2L_1L_3\cos(\alpha)$$

$$\frac{L_2^2 - L_1^2 - L_3^2}{-2L_1L_3} = \cos(\alpha)$$

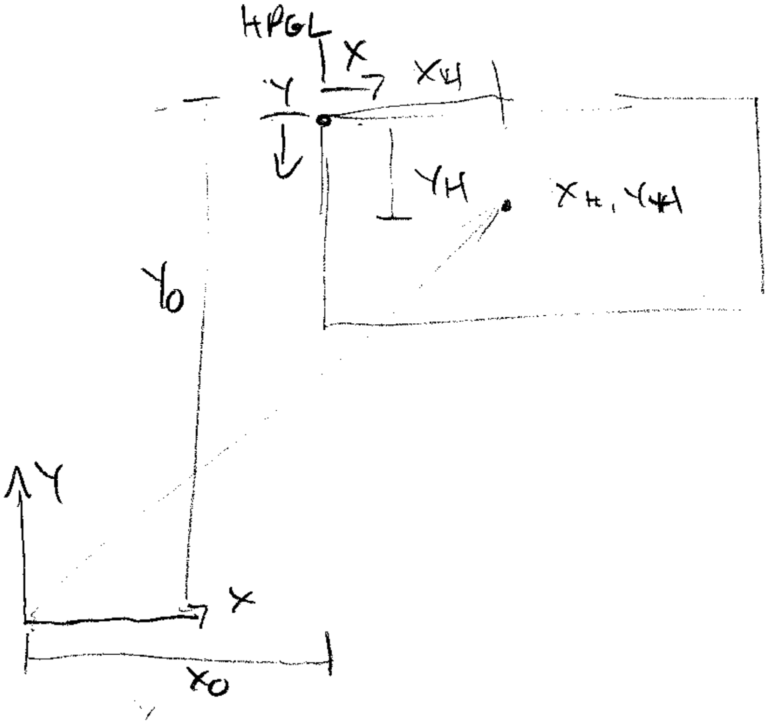
$$\alpha = \cos^{-1}\left(\frac{L_2^2 - L_1^2 - L_3^2}{-2L_1L_3}\right)$$

$$\theta_1 = \tan^{-1}\left(\frac{y_p}{x_p}\right) + \cos^{-1}\left(\frac{L_2^2 - L_1^2 - L_3^2}{-2L_1L_3}\right)$$

$$\boxed{\theta_1 = \tan^{-1}\left(\frac{y_p}{x_p}\right) + \cos^{-1}\left(\frac{L_1^2 + L_3^2 - L_2^2}{2L_1L_3}\right)}$$

ORDER OF CALCULATION IN CODE

GIVEN AN x_p, y_p IN ~~A~~ PAPER SPACE



x_H, y_H IS GIVEN IN HPGL COORDINATES

ACTUAL x_H, y_H IS $\boxed{\frac{x_{HPGL}}{DPI}}$ AND $\boxed{\frac{y_{HPGL}}{DPI}}$

DEFAULT DPI IS 1016

THUS x_p, y_p IS FOUND BY

$$\boxed{\begin{aligned} x_p &= x_0 + x_H \\ y_p &= y_0 - y_H \end{aligned}}$$

WITH $x_p, y_p \Rightarrow \boxed{L_3 = \sqrt{x_p^2 + y_p^2}}$

AND

$$\boxed{\begin{aligned} \theta_1 &= \tan^{-1}\left(\frac{y_p}{x_p}\right) + \cos^{-1}\left(\frac{L_1^2 + L_3^2 - L_2^2}{2 L_1 L_3}\right) \\ \theta_2 &= \underset{\text{OR } 180^\circ}{\pi} - \cos^{-1}\left(\frac{L_1^2 + L_2^2 - L_3^2}{2 L_1 L_2}\right) \end{aligned}}$$

4/9

θ's ⇒ TICKS

CPR ≡ COUNTS PER REV OF OUTPUT SHAFT

COUNTS = TICKS

$$\boxed{\text{TICKS} = \frac{\text{CPR}}{360} \cdot \theta}$$

$$\frac{\text{TICKS}}{1\,\text{REV}} \cdot \frac{\cancel{1\,\text{REV}}}{\cancel{360}} \cdot \cancel{\deg} \Rightarrow \underline{\underline{\text{TICKS}}}$$

FROM CAD AND MEASUREMENTS

L_1 = 8.1102 IN

L_2 = 10.078 IN

X_c = 0.5 IN

Y_c = 14.5 IN

```
                                  INPUT        OUTPUT      RES/DPI
                                    ↓            ↓            ↓
⇒ python   parse_hpgl.py     hello.hpgl   output.txt   1016

               3200     8.11     10.08     0.5     14.5
                 ↑        ↑        ↑        ↑        ↑
                CPR       L1       L2       Xc       Yc
```

CPR = 3200 BECAUSE

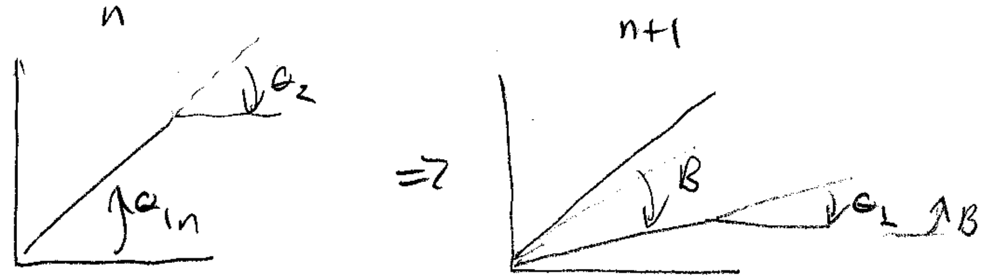
64 TICKS PER 1 REV OF MOTOR, BUT

50:1 GEAR RATIO

THUS

CPR ON OUTPUT ⇒ 64·50 = $\underline{3200}$

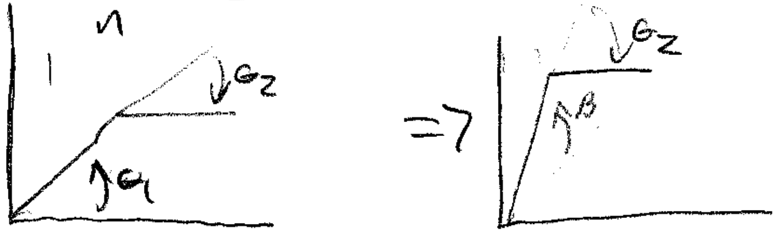
ACCOUNTING FOR θ_2 DEPENDENCE ON θ_1



$$B = \theta_n - \theta_{n+1}$$

$$\theta_{2n+1} = \theta_{2n} - B$$
$$= \theta_2$$

$$\boxed{\theta_{2_{n+1}} = \theta_{2n} + \theta_{n+1} - \theta_n}$$



$$B = \theta_{n+1} - \theta_n$$

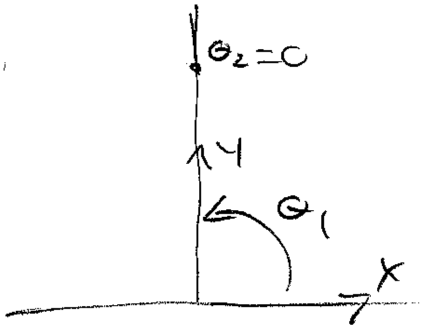
$$\theta_{2_{n+1}} = \theta_{2n} + B$$

$$\boxed{\theta_{2n+1} = \theta_{2n} + \theta_{n+1} - \theta_n}$$

6/4

CALIBRATION

SET TO $\theta_1 = 90$, $\theta_2 = 0$
AKA $\theta_1 = 800$ TICKS, $\theta_2 = 0$ TICKS

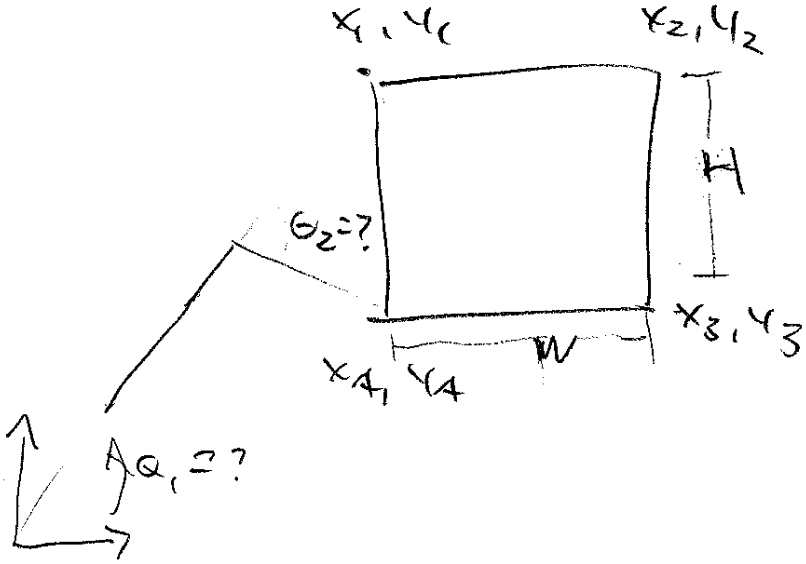


ALTERNATIVE METHOD

WOULD THERE BE A WAY TO HAVE SELF CALIBRATION?

GIVEN ARBITRAY BUT KNOWN x's, y's SAY; FOUR POINTS OF A PIECE OF A PIECE OF A PAPER, THE LENGTH AND THE WIDTH OF THE PAPER, CAN THE GLOBAL COORDINATE SYSTEM AND ABSOLUTE θ's BE FOUND?

SEE



TICKS

KEY $T_{\underline{\text{MOTOR}},\ \underline{\text{point}}}$

$x_1, y_1 \Rightarrow T_{1,1}, T_{2,1}$

$\downarrow \delta T_{1,1-2}, \delta T_{2,1-2}$

$x_2, y_2 \Rightarrow T_{1,2}, T_{2,2}$

$\downarrow \delta T_{1,2-3}, \delta T_{2,2-3}$

$x_3, y_3 \Rightarrow T_{1,3}, T_{2,3}$

$\downarrow \delta T_{1,3-4}, \delta T_{2,3-4}$

$x_4, y_4 \Rightarrow T_{1,4}, T_{2,4}$

$\downarrow \delta T_{1,4-1}, \delta_{2,4-1}$

SINCE WE KNOW

FROM $x_1, y_1 \Rightarrow x_2, y_2$

$\Delta x = W$

$\Delta y = 0$

$x_2, y_2 \Rightarrow x_3, y_3$

$\Delta x = 0$

$\Delta y = -H$

$x_3, y_3 \Rightarrow x_4, y_4$

$\Delta x = -W$

$\Delta y = 0$

$x_4, y_4 \Rightarrow x_1, y_1$

$\Delta x = 0$

$\Delta y = H$

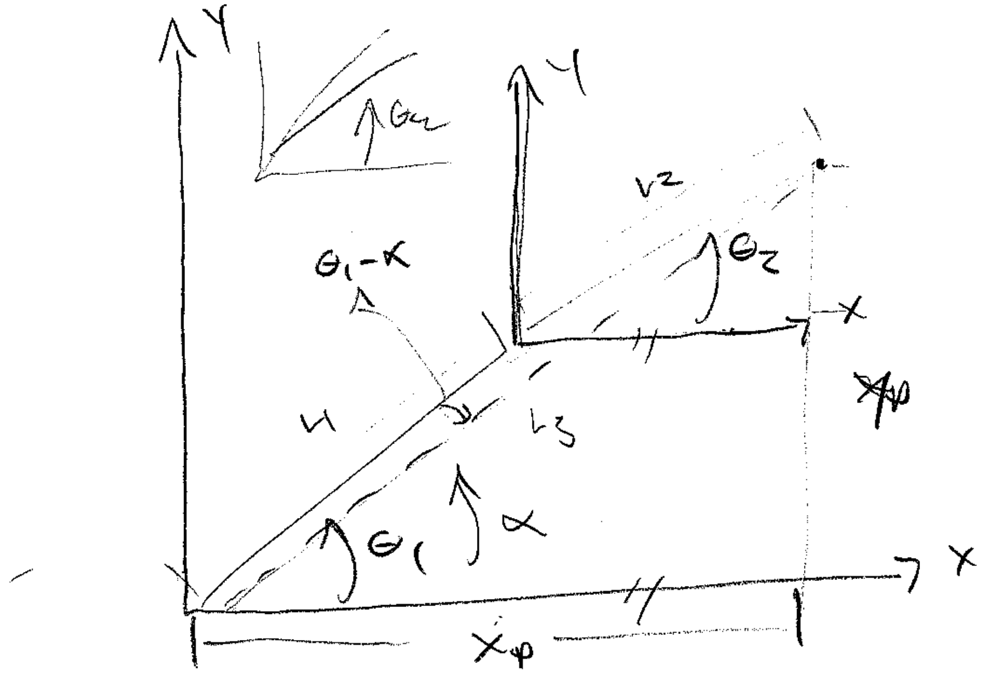
COULD YOU FIND ABSOLUTE θ'S ?

$\Delta x \Leftrightarrow \Delta\theta$'s?

Δy

NOT SURE... THIS WOULD BE COOL THOUGH.

ALTERNATIVE COORDINATE SYSTEM SETUP



$$x_p = L_1 \cos\theta_1 + L_2 \cos\theta_2$$

$$y_p = L_1 \sin\theta_1 + L_2 \sin\theta_2$$

$$L_3 = \sqrt{x_p^2 + y_p^2}$$

$$\alpha = \sin^{-1}\frac{y_p}{L_3} = \tan^{-1}\left(\frac{y_p}{x_p}\right)$$

$$L_2^2 = L_1^2 + L_3^2 - 2L_1L_3\cos(\theta_1 - \alpha)$$

$$\frac{L_1^2 + L_3^2 - L_2^2}{2L_1L_3} = \cos(\theta_1 - \alpha)$$

$$\theta_1 = \cos^{-1}\left(\frac{L_1^2 + L_3^2 - L_2^2}{2L_1L_3}\right) + \alpha$$

$$\boxed{\theta_1 = \cos^{-1}\left(\frac{L_1^2 + L_3^2 - L_2^2}{2L_1L_3}\right) + \sin^{-1}\left(\frac{y_p}{L_3}\right)}$$

$$+ \tan^{-1}\left(\frac{y_p}{x_p}\right)$$

$$x_p - L_1 \cos\theta_1 = L_2 \cos\theta_2$$

$$\boxed{\theta_2 = \cos^{-1}\left(\frac{x_p - L_1 \cos\theta_1}{L_2}\right)}$$

or

$$\theta_2 = \sin^{-1}\left(\frac{y_p - L_1 \sin\theta_1}{L_2}\right)$$

$$\boxed{\begin{aligned} L_3 &= \sqrt{x_p^2 + y_p^2} \\ \theta_1 &= \cos^{-1}\left(\frac{L_1^2 + L_3^2 - L_2^2}{2L_1L_3}\right) + \tan^{-1}\left(\frac{y_p}{x_p}\right) \\ \theta_2 &= \cos^{-1}\left(\frac{x_p - L_1 \cos\theta_1}{L_2}\right) \end{aligned}}$$

5/15/18 PLANNING ME-405-04 D. KYLE 1/4

GENERAL SOFTWARE STEPS FOR PLOTTING A DRAWING

1.) Create vector drawing on HPGL and save as a file that a PC program can accept.

2) Create PC program that reads HPGL format file and transmits the file to the Micropython board.

3) The x and y vector coordinates in HPGL file that are read from PC program are converted to target positions from our derived kinematic equations for Xp and Yp.

4) Setpoint in motor-Sam.dima controls how many ticks motor will spin.

- Specify one arm to be the X position and other arm to be the y position

↓ Each motor will be assigned its own Setpoint value that it gets from the vector position read from the HPGL format file.

For every increment in X and Y target position. from HPGL vector, update new increments for length of L_3, L_1, L_2 inside θ_2 and θ_1 equations.

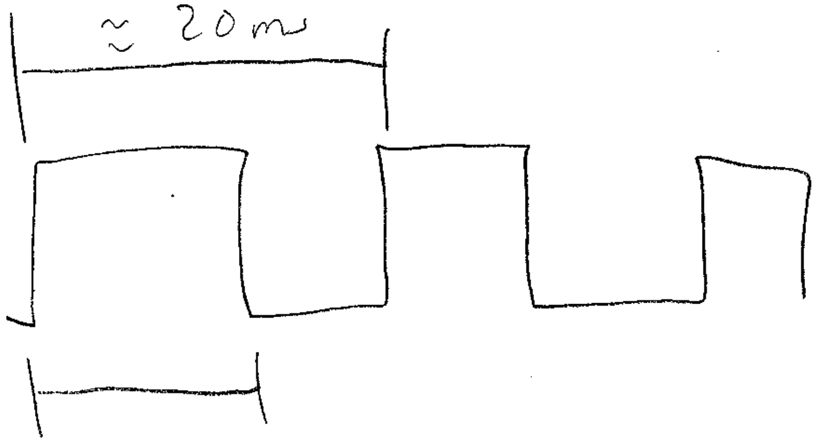
- Then solve for θ_1 and θ_2 and convert from degrees to ticks using $\frac{5.33\ \cancel{\text{ticks}}}{\cancel{X}\text{ deg.}}$ $\frac{8.\cancel{88}\text{ ticks}}{\text{deg}}$

↓ this θ_1 and θ_2 in terms of ticks becomes the set point for the X pos motor and Y pos motor.

SERVO

50 Hz .02 sec.

$f_{pwm} = 50$ Hz

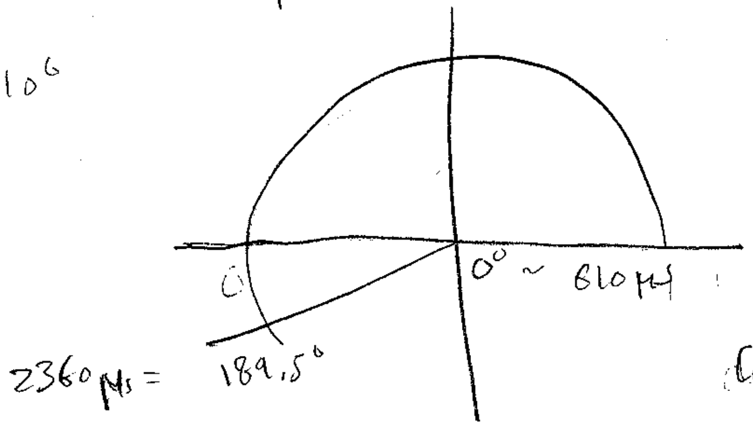


$610 < T_{width} < 2360$ μs

610×10^6

$$\frac{2360\ \mu s - 610\ \mu s}{189.5\ \text{deg}} = 9.235\ \mu s/\text{deg}$$

0.108 deg/μs



Conversion = $\frac{189.5\ \text{deg}}{1750\ \mu s}$