Nerf Blaster Turret Project

Team Mecha25

Tyler Noxon

Nathaniel Furbeyre

Kair Turubayev

03/19/2018

Table of Contents

1 Introduction

The purpose of this project was to design, build, program and test a device that aims and fires foam darts at specified locations on the ME 405 whiteboard. The device must be able to fire upon a 40”x40” grid consisting of 25 8”x8” squares while rotating about two different axes (pan and tilt). The device was designed so that it would take in a row and column and then move to the corresponding location on the board to fire the dart automatically. To make our automated design possible we used our knowledge of mechanical design, MicroPython programming, controls, and basic electric circuits. Our design development process contained software design and hardware design, which consisted of mechanical and electrical components.

2 Project Specifications

We were required to meet a variety of specifications so that the project would reflect a culmination of the information that we have learned throughout the quarter. The basic requirements outlined on the assignment document can be seen in Table 1. As shown, we did not utilize an encoder in our design because we opted to use a BNO055 IMU to record both angle measurements for our closed loop control systems.

Table 1. Initial Design Requirements from Assignment Sheet.

Initial Design Requirements	Pass/Fail
ME405 board or equivalent running MicroPython	Pass
Two DC Motors to move the device	Pass
An encoder for measuring movement on one axis	Fail
An IMU for measuring movement on one axis	Pass
Remotely Operated User Interface for entering coordinates	Pass
Closed-loop control systems for motor control	Pass

Once we had chosen our blaster and begun the design process, we found that there were additional design specifications necessary to ensure device functionality and safety. These requirements can be seen in Table 2. For our motor requirements, the calculations can be seen in Attachment A.

Table 2. Project Design Specifications

Project Design Specifications	Pass/Fail
Automatic Trigger Firing Mechanism	Pass
Only use one DC Power Supply (plugged in)	Pass
Calibration sequence for zeroing device	Pass
Device must be safe	Pass
Device must weight approximately 1 kg or less	Pass
Device must be able to tilt at least 45 degrees	Pass
Motors must be able to produce 15.6 mN-m of torque	Pass
Time to Aim < 1 second	Pass
Device must be able to rotate at least 120 degrees	Pass

3 Design Development

Over the course of the project, we considered a few different designs for building our device. A couple of the major design decisions were choosing between using gears or pulleys, and using an assembled gun or a gun deconstructed from existing products.

3.1 Design Decisions

When we first began brainstorming ideas for design, we thought that we might use 3D printed gears for transmitting the torque between the motors and the turret assembly. However, as we progressed in our design, we decided to use belts and pulleys for two main reasons. The first reason was that we could simply laser cut large pulleys for the belts rather than 3D print full gear sets. Additionally, after looking at the Johnson Electric HC615G motors we were going to use, we found that the belt pulleys attached would be difficult to remove without modification. Since these motors were not ours to modify, we determined we would just source a belt with the same tooth size as the motor pulleys.

We had some trouble deciding between using a deconstructed gun or an assembled gun. The benefits to using the deconstructed gun were that we could see a greater reduction in total weight—we would only have the parts essential for the firing mechanism to operate. However, we found that the drawbacks for this approach included having to design and print mechanisms that mimicked the functions built in to a preexisting gun. We ultimately decided to modify a commercial gun (Nerf Doomlands Desolator) as we could use the same linkages and magazine to hold the darts while feeding them into the flywheels without the need to redesign and test a similar system. We were considering purchasing the Nerf Stryfe, but we opted for the Doomlands Desolator because the rails across the top were along the same plane, making it easier to mount

the gun upside down. We thought that mounting the gun upside down made the most sense as it would allow us to have access to the magazine without having to disassemble the device. The Doomlands Desolator can be seen in Figure 1.

Figure 1. Nerf Doomlands Desolator.

Another design decision that we needed to make was how to calibrate our gun with respect to the board. We needed a robust method that would account for changes in relative positions to the board, including varying projectile angles for the nerf darts. To do this we opted for a simple approach of targeting 9 squares and recording the IMU Euler angles for pitch and heading at each square. We picked the following squares spread out evenly and symmetrically across the board: A1, A3, A5, C1, C3, C5, E1, E3 and E5. Once we had generated all the angles for these squares, we made a simple, yet effective assumption that all unrecorded squares were along midpoints of the squares that we had just recorded angles for. A simple Python function was able fill in the coordinates for empty spots. Once all the board coordinates were generated, our machine was considered "calibrated." As can be seen on the state/transition diagrams in Appendix E, this was a pivotal point that switched many of our tasks from State 1 to State 2 during the execution of our program.

3.2 Final Hardware Design (mechanical and electrical)

3.2.1 Mechanical Design

We began the mechanical design by estimating the weight of our device and calculating the moment of inertia about the two axes of rotations. We assumed that our system had uniform mass distribution, therefore a center of gravity is the same point as the center of mass (Platform 1 at point O1 – center of xyz system, Platform 2 at point O2 – center of XYZ system). Moreover, we neglected all friction forces in the system. We then decided on gear ratios that would fit our mechanical system and found that the minimum torque requirement for our system was 15.6 mN-m. These calculations can be seen in Attachment A. Since the minimum torque required by each platform was 15.6 mN-m, a gear ratio between motor 1 and platform 1 was made to be 24:1, and the gear ratio between motor 2 and platform 2 was made to be 12:1. These ratios allowed us to have high factors of safety (FS1 = 11.5, FS2 = 6.5) in reference to

the torque needed to move the system (Appendix A). Also, the high gear ratios allowed us to have a higher angular acceleration rate, which was crucial for a responsive system.

3.2.1.1 Motor and Belt Choice

Once we figured out our minimum expectable torque, we found motors that would be strong enough for rotating our design. The motors we chose were the Johnson Electric HC615G DC Motors that were available in class. These motors were already available and didn't require us to purchase them, which was a big benefit to us. Additionally, we made sure that the motors fit our design specifications by looking at the data sheets found in Appendix D. Having the motors immediately available was advantageous as we could then begin building our device with motor mounts incorporated. The motors that we used from class already had pulleys attached to them, so we just sourced toothed belts with the same tooth pitch to be compatible.

Additionally, we decided to use a high torque servo with a linkage system to actuate the trigger firing mechanism in the gun. An image of our linkage system can be seen in Figure 2. This mechanism allowed us to automatically operate our trigger when the device was pointed at the correct location.

Figure 2. Servo Trigger Linkage Mechanism.

3.2.1.2 Bearings

For our design, we utilized a Lazy Susan bearing and 2 skateboard bearings to allow for smooth rotation of our parts. We used the Lazy Susan to rotate the entire assembly whereas the skateboard bearings were used to rotate the top platform that held the gun. We used the skateboard bearings because we had them on hand, and we mounted them using an interference fit between the plywood and the bearings. We had a fixed 5/16" threaded shaft than ran through the bearings and attached our top platform to the rest of the assembly.

3.2.1.3 Main Mechanical Design

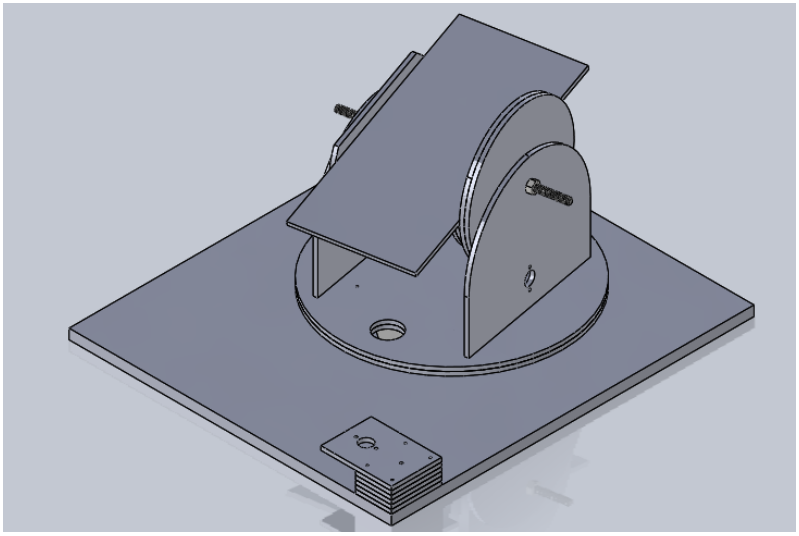


Figure 3. Isometric view of Solidworks Model.

Most of the components of our assembly were made from laser cut plywood. We decided to use plywood as we wanted to keep the weight of the system down and figured that it would be strong enough for our needs. All of the required parts were made on campus, utilizing the laser cutter in the Mustang 60 shop. A representation of the main assembly can be seen above in Figure 3. An exploded view indicating how to assemble the device can be found in Appendix C. Most of our assembly was completed using woodscrews, however, we also 3D printed brackets to hold our system together. This bracket can be seen in Figure 4. We found these brackets on the website thingiverse.com, and they were designed by Azdle.

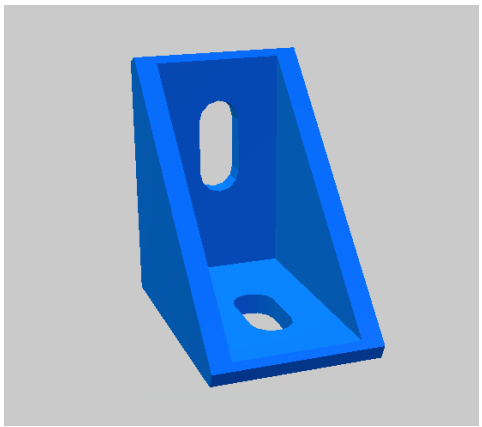


Figure 4. 3D printed bracket designed by Azdle.

3.2.2 Electrical Design

For our electrical design, we decided that we only use one DC power supply plugged into the wall. Because of this, we needed to use additional ways to step down voltages so that everything could be powered using the correct voltages. The electrical design includes everything that we plugged into both the ME405 board and the nearby breadboard. These items include all motors, the IMU, and the laser diode. Refer to Appendix E for connection diagram.

3.2.2.1 Varying Voltages

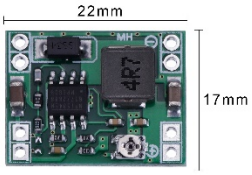


Figure 5. Eboot Mini MP1584EN Buck Converter

In order to combat the need for multiple voltage sources, we had to use different ways to supply the needed voltages and currents. From our DC power supply, we pulled 18V to supply our motors, but we also needed to use a buck converter to step down the voltage to 6V for our servo motor. To do this, we used the Eboot Mini MP1584EN Buck converter as seen in Figure 5. For the servo, we also added a 1000 μF capacitor to smooth out the voltage seen by the servo. For our IMU and laser diode, we were able to power them using the onboard voltage supplies of our microprocessor. The flywheel motors inside the gun were ran on their own system at 7.4V utilizing 2 18650 Lithium batteries connected in series.

3.2.2.2 Sensor

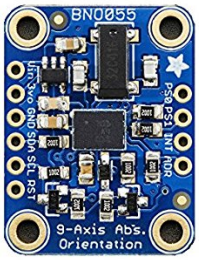


Figure 6. Adafruit BNO055 Sensor

By deciding that we were going to use our IMU to receive both angular position measurements, we ended up purchasing the Adafruit BNO055 Absolute Orientation Sensor. We went with this sensor because it had the ability to directly measure Euler angles to give us the angular position of the board. Initially, we thought that we might run into the problem of gimbal lock with our sensor but because we were never going to have our device tilt 90 degrees, it was not

an issue. This allowed us to read the angular data easily without the need to convert to quaternions for angular measurements.

3.2.2.3 Laser Diode

Figure 7. Laser Diode

In order for us to calibrate our gun, we relied on the use of a laser diode, mounted directly beneath the barrel. The positioning of the laser allowed us to know where the barrel was pointing so that we could accurately locate our targets. It was also useful for tuning our controller because we could see how the system responded with the movement of the laser. We were able to directly connect the laser to the onboard power pins in order to send current through the diode.

3.3 Software Design

All the software used to make our project work was written in Python. Since we worked with a Nucleo board to control each component of the design, MicroPython libraries and modules were consulted for effective design of the project. Many files imported the module *pyb* which allowed for interfacing with the microcontroller pins and timers. Significant amounts of code involved in this project had been written by our team during lab sessions earlier in the quarter. For example, our ProjectClasses.py file contained functional class implementations of a motor driver and closed loop controller. These two classes were further modified during this project to be compatible with the design of our nerf turret. A Servo class that added that ability to control a servo rotation was written to allow actuation of a nerf dart into the firing motors.

One of the major components in our nerf turret design was the use of AdaFruit's BNO055 Absolute Orientation Sensor [1]. We opted out of using an encoder and decided to control both axes of motion with this IMU. To interface between our own written code and the IMU, we wrote a device driver that helps integrate the built-in architecture of the IMU to our code. Much of the code we wrote was based off the BNO055 product specification documents, with focus on writing operation modes to certain registers and reading sensor data from others. We specifically read fused Euler angle data from registers which had to be unpacked little-endian style for relaying to our controllers:

```
def get_heading (self):
    """ Get the heading angle from the BNO055 in degrees, assuming
    that the accelerometer was correctly calibrated at the factory.
    @return The measured heading angle in degrees """
    data = self.i2c.mem_read(2, DEVICE_ADDR, HEADING_LSB)
    unpacked = ustruct.unpack('<h', data)
    return unpacked[0] / 16
```

We must also give credit to a couple of sources that we borrowed material from. The mybno055.py file contains code taken from previously written work by author *deshipu* on github.com. In addition, the init function for the BNO055 class was modelled off Dr. Ridgely's init function on mma845x_shell.py on the ME 405 webpage.

Our motor controller classes implemented PID control to get the IMU positioned in the same direction as the specified set points for our heading a pitch axes. After significant dialing and tuning the gain constants for the controllers we found the most optimal values corresponding to our physical system to be a kp of 8.5, ki of 0.2 and kd of 0.5 for our heading motor. The pitch motor was optimized to a kp of 5.8, a ki of 0.25 and a kd of 0.25. These controller gains gave us the quickest response time and had our laser pointer zero in to the targeted squares. One problem that we had for traversing large heading angles was that our heading motor would sometimes reach an unsteady state, oscillating about the set point but never actually being able to reach it. Fortunately, this only happened for changes of angles larger than what we needed to account for with the specified game board of 40” by 40”.

Writing all the previous classes allowed for smooth incorporation of their functionality to our main.py file. This file contained several different functions acting as individual tasks in our multitasking scheme. These tasks, in conjunction with the intertask communication variables written by Dr. Ridgely in task_share.py, allowed our code to simultaneously run each component of the design for a smooth-performing system. Each function was passed into a Task object initializer (code also written by Dr. Ridgely) to allow the functions to run in a priority scheduler. The State/Transition Diagrams for each of our tasks can be viewed in Appendix D. The following Task Diagram (with influence from Dr. Ridgely [2]) shown in Figure 8 is used to visualize the organization of our machine:

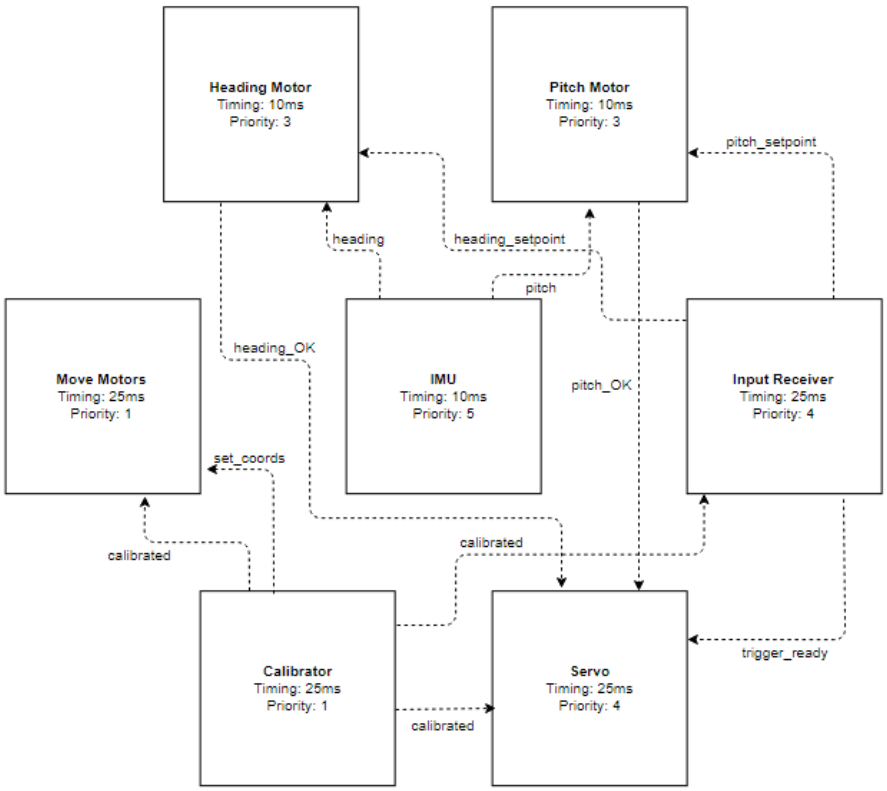


Figure 8. Task Diagram for Software Design

As we discovered by experimenting with changing the priority and timing of our tasks, they did not matter too much in the effectiveness of our turret. Our design did not contain enough tasks to bring us near the upper limit of the processing capabilities of our board. For the priorities, we deemed the Calibrator and Move Motor tasks to be least important. It didn't matter if our Calibrator put in the coordinates of the 9 targeted squares several seconds after they were targeted and selected, as long as it happened before we were ready to select a square for firing on. In addition, the Move Motor function used to move the laser pointer to one of the 9 squares could be delayed significantly, as long as we were able to target as close to the middle of a square as possible. Both motors were placed at a priority below the Input Receiver and Servo Tasks. The Servo and Input Receiver tasks were helpful in getting the gun to target the square as quickly as possible. Finally, the IMU had the highest priority as we wanted to have the most accurate angles to lock in our motors as we got closer to the set points. From a timing perspective, the IMU was set at 10ms, which is the sample rate for the BNO055 as listed in its documentation. Both motors were also scheduled at the same rate of 10ms. The other 4 tasks all had timings specified to every 25ms.

Finally, we wrote a script to be run on a PC, allowing user input to specify which square on the game board to target. This script begins by initializing a Serial object from the imported serial module and reset the board:

```
with serial.Serial('/dev/ttyACM0',115200, 5) as ser:
    ser.write([0x04])
```

From here the script allows the user to point the nerf gun (using w, a, s, d keys) with help of a laser pointer to all 9 spots needed for calibration. After calibration is completed, a while True loop is used for continually asking the user which row and column for the turret to target.

4 Results

Our system was successful with regard to the closed loop motor controllers that we implemented. During the lab competition, we were able to accurately target 8 out of 10 squares (based on visual inspection of our laser pointer) that our professor had called out. In addition, one of the two targeting failures was due to user error of inputting a wrong square for the PC-microcontroller communication. Our system also had a very quick response time, probably due to our controllers using such high proportional constant values. Unfortunately the servo/firing mechanism of our turret was not as successful as the targeting capabilities. Using limited torque servos that we had found in the lab ended up causing a significant number of dart jams while trying to fire. With some design reiterations, we could address this problem either by using a more powerful servo to actuate the darts, or to extend the linkage pushing the darts in to place by about a half inch.

Overall, this project was immensely fun, and all of our team members agree that this has been one of the most rewarding and engaging projects in our undergrad career. The ability to design and implement a successful turret is not something that we would have been able to do 10 weeks ago.

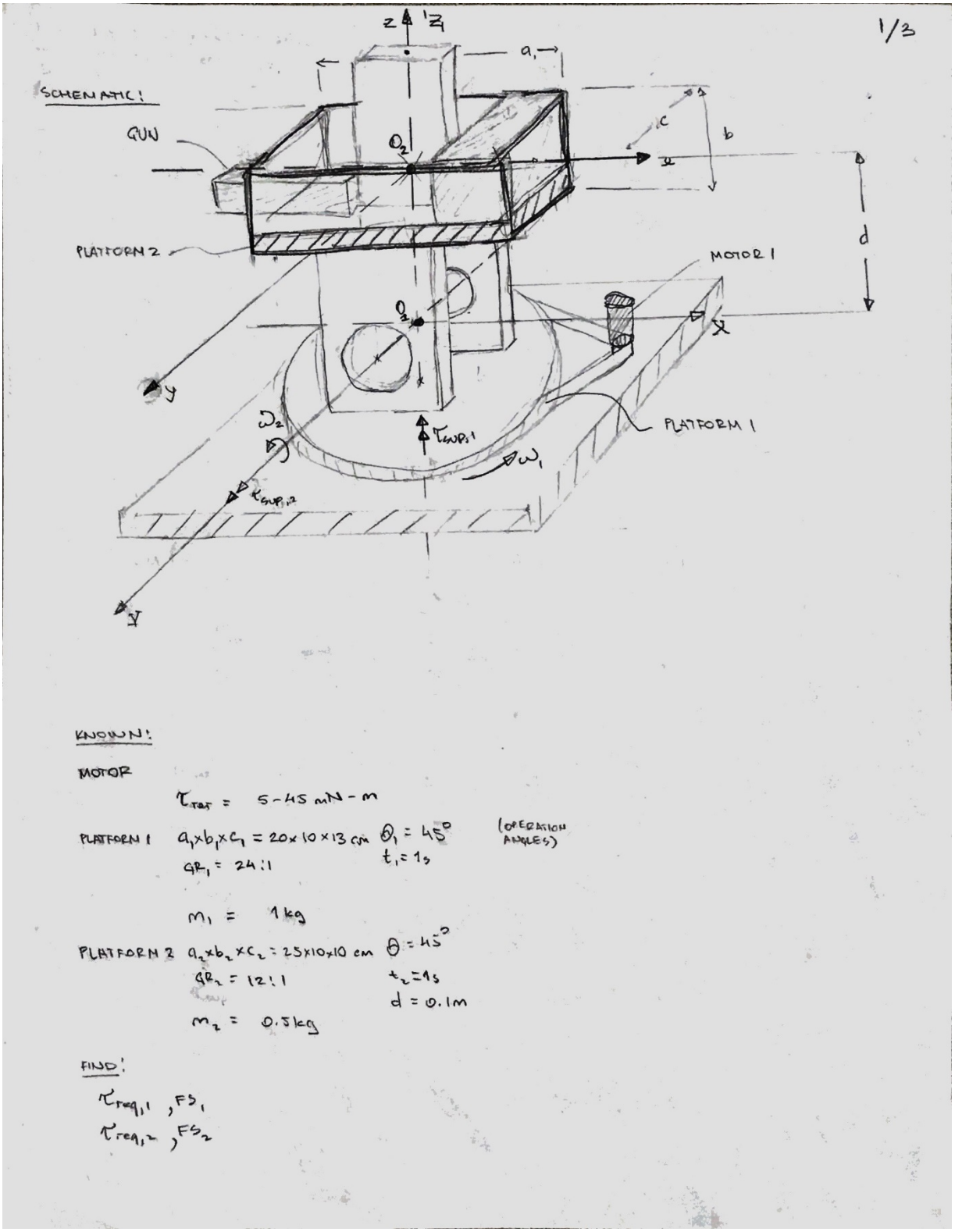
References

[1] Bosch Sensortec, “BNO055 Intelligent 9-axis absolute orientation sensor data sheet”, 11/14
[2] Ridgely, J.R. “Using Tasks and Finite State Machines in Python”, *Cal Poly Mechanical Engineering*.
[3] Source code at http://wind.calpoly.edu/hg/mecha25

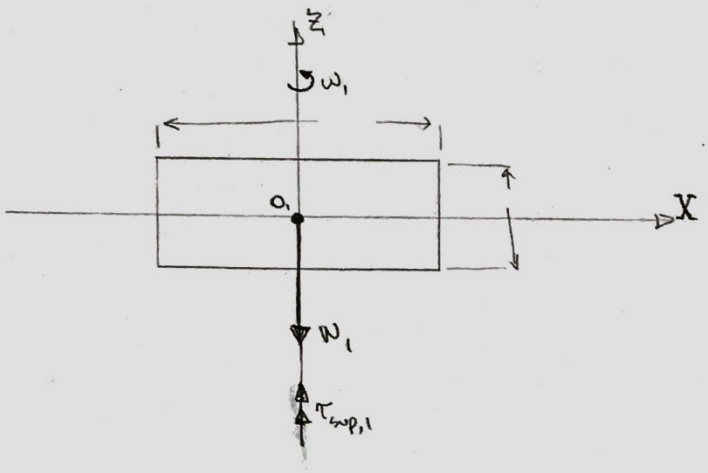
Appendices

[A] Design Sample Hand Calculations
[B] State/Transition Diagrams
[C] Solidworks drawings
[D] Motor Torque Curves
[E] Electric Circuit Diagram
[F] Doxygen Documentation

Appendix A – Design Sample Hand Calculations



PLATFORM 1 (XYZ)



ASSUMPTIONS:
1. UNIFORM MASS DISTRIBUTION
2. $f_{rr} \sim 0$
3. CENTER OF MASS AT POINT O_1

ANALYSIS:

EULER'S EQUATION:

$$\Sigma M_z = \left[I_{zz}\dot{\omega}_z - (I_{xx} - I_{yy})\,\omega_x \omega_y \right]$$

$$T_{req,1} = I_{zz}\dot{\omega}_z$$

$$I_{zz} = \frac{1}{12} M_1 (a_1^2 + c_1^2)$$

$$\theta = \omega_{z_0} t_1 + \frac{\dot{\omega}_z t_1^2}{2}$$

(ω_{z_0} = 0)

(INITIAL SPEED = 0 rad/s)

$$\dot{\omega}_z = 2\theta / t_1^2$$

$$\boxed{T_{req,1} = \left[\frac{1}{12} M_1 (a_1^2 + c_1^2) \right] \left[2\theta / t_1^2 \right]}$$

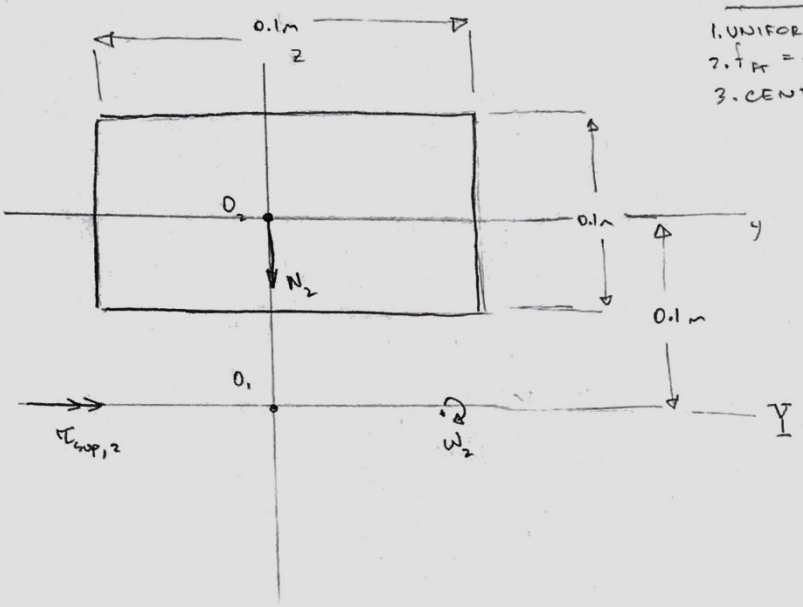
$$= \left[\frac{1}{12} (1\,kg)(0.25^2 + 0.15^2)\,m^2 \right] \left[2\,\frac{\pi}{4}\,rad/s^2 \right]$$

$$\boxed{T_{req,1} = 10.4\ mN \cdot m}$$

WITH $GR_1 = 24$
AND $T_{MOT} = 5\ mN \cdot m$

$FS_1 = 11.5$

PLATFORM 2 (xyz)



ASSUMPTIONS:

1. UNIFORM MASS DISTRIBUTION
2. $\uparrow_{PT} = 0$
3. CENTER OF MASS AT O_2

ANALYSIS:

EULER'S EQUATION:

$$\Sigma M_y = \left[I_{yy}\dot{\omega}_y - (I_{zz} - I_{xx})\omega_z \omega_x \right]$$

$$\tau_{req,2} = I_{yy}\dot{\omega}_y$$

$$I_{yy} = I'_{yy} + m_2 d^2$$

$$= \frac{1}{12} m_2 (b_2^2 + c_2^2) + m_2 d^2$$

THEN:

$$\tau_{req,2} = \left[\frac{1}{12} m_2 (b_2^2 + c_2^2) + m_2 d^2 \right] \left[\frac{2\theta_2}{t_2^2} \right]$$

$$= \left[\frac{1}{12}(0.5kg)(0.1^2 + 0.1^2)m^2 + 0.5kg(0.1^2)m \right] \left[\frac{2 \cdot \frac{\pi}{4}}{1s^2} \right]$$

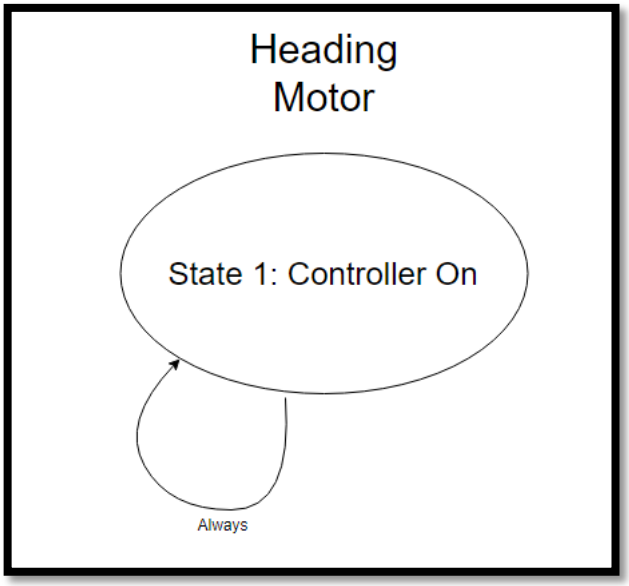
$$\tau_{req,2} = 9.16\ mN \cdot m$$

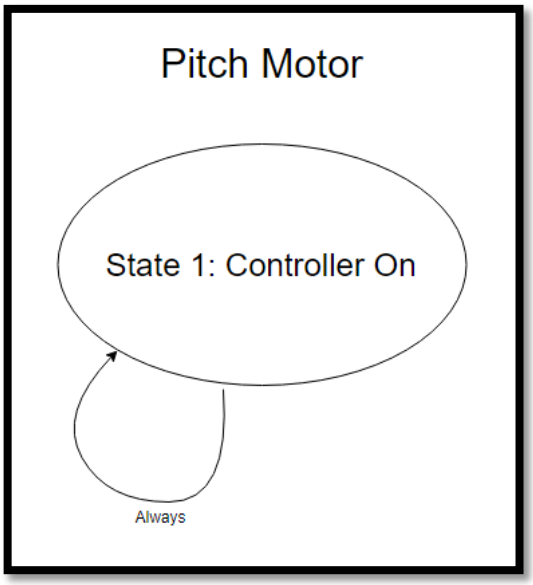
WITH $GR_2 = 12$

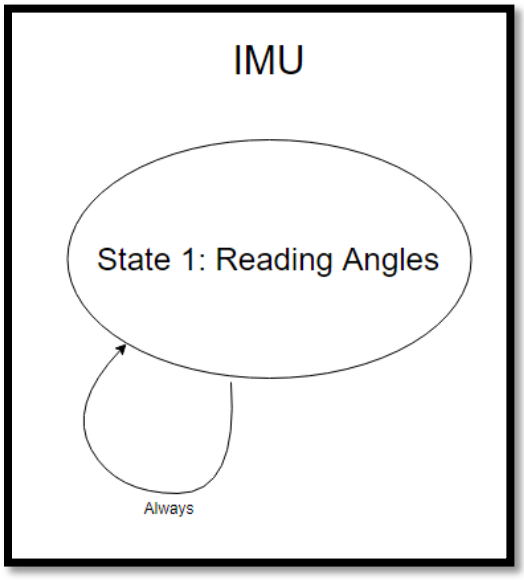
AND $\tau_{mot} = 5\ mN \cdot m$

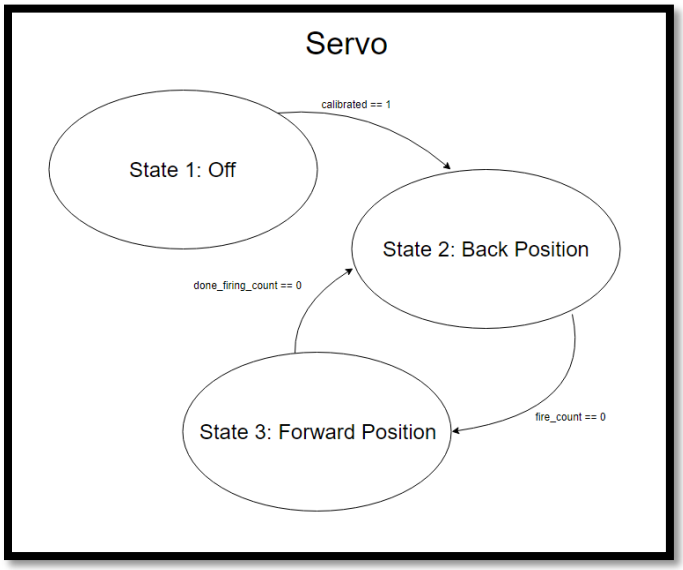
$FS_2 = 6.5$

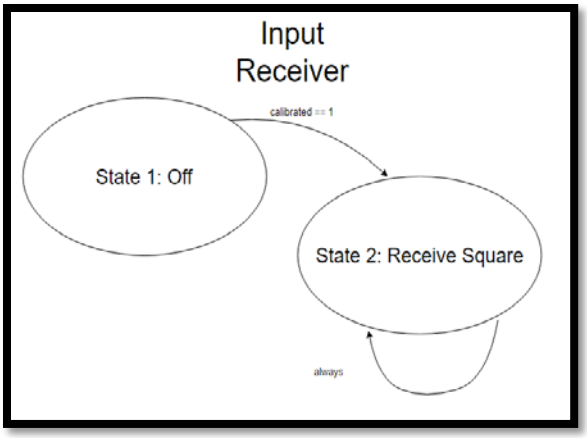
Appendix B – State/Transition Diagrams

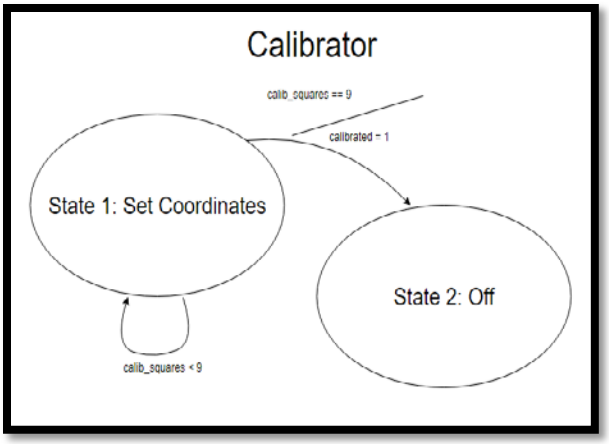


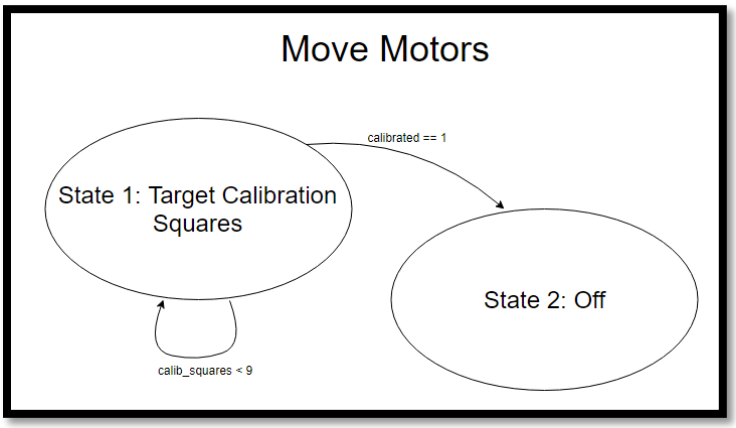




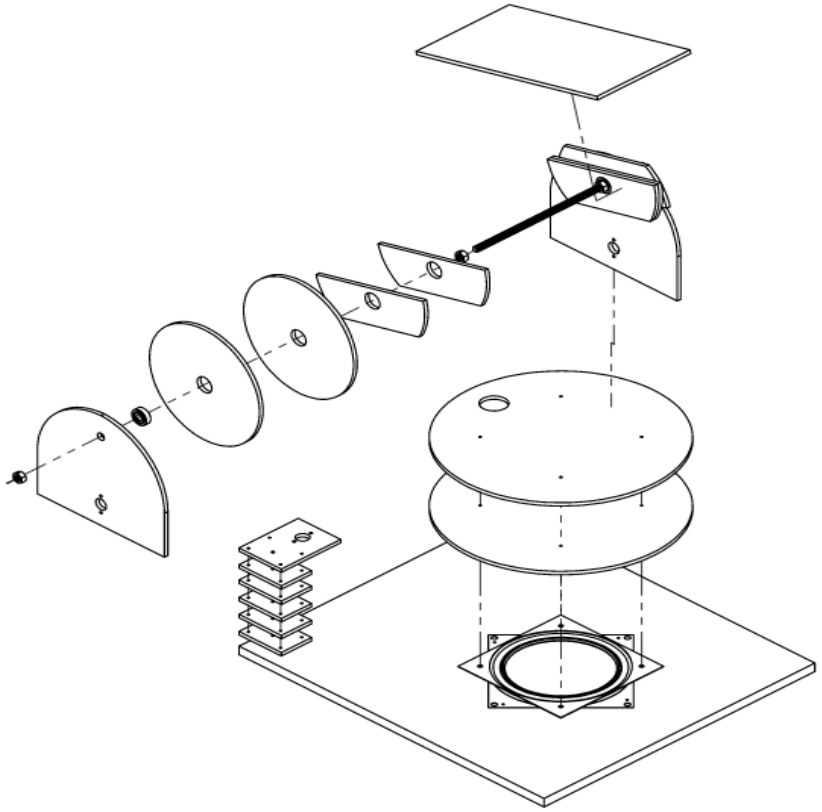


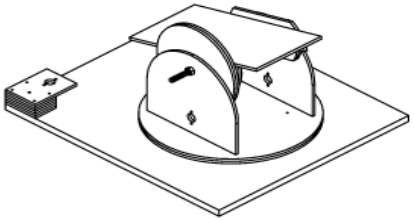

Input
Receiver
calibrated == 1
State 1: Off
State 2: Receive Square
always

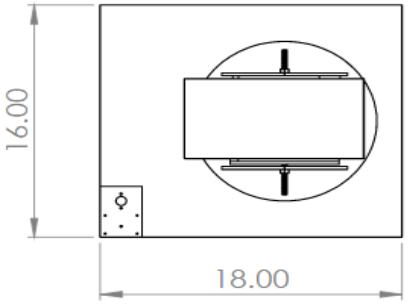

Calibrator
calib_squares == 9
calibrated = 1
State 1: Set Coordinates
State 2: Off
calib_squares < 9

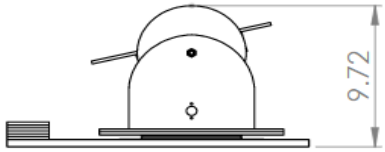

Move Motors
calibrated == 1
State 1: Target Calibration Squares
State 2: Off
calib_squares < 9

APPENDIX C: SOLIDWORKS DESIGN









Cal Poly Mechanical Engineering ME 405 - Winter 2018	Lab Section: Dwg. #:	Assignment # Nxt Asb:	Title:Turret Mechanical Assembly Date:3/19/2018	Scale: 1:7	Drwn. By: TYLER NOXON Chkd. By: ME STAFF

Appendix D – Motor Torque Curves

HC615G-001 PMDC Motor

PMDC

Market
Leisure & Fitness

Application:
Joystick

Motor for joystick application that has high power, low cogging, high stall torque, low starting voltage and longer life

Motor Characteristics:

Diameter	: 35.8 mm
Length	: 50 mm
Shaft Diameter	: 3.175 mm
Weight	: 145.5 g
Nominal Voltage	: 24 V
Torque Constant	: 94.45 m-Nm/A
Dynamic Resistance	: 50.59 Ohms
Motor Regulations	: 54.60 Rpm/m-Nm
Pole Number	: 5

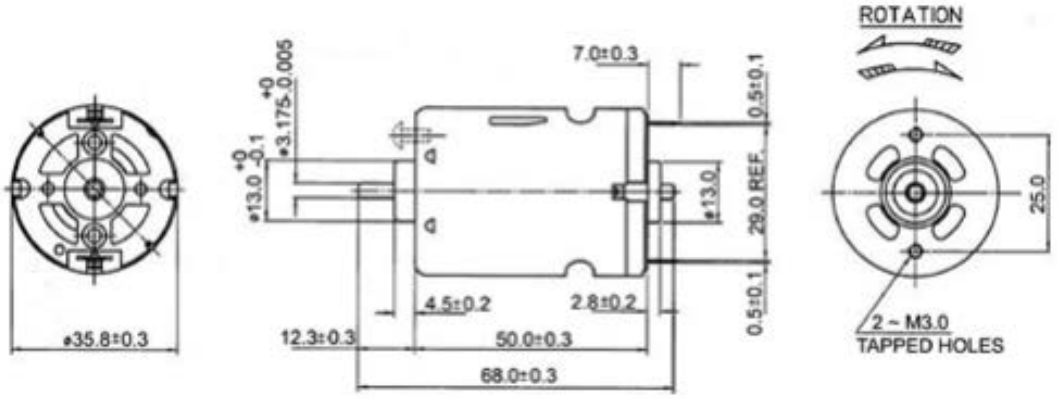
Standard Data:

Operation Temperature	: -10 — 50°C
Storage Temperature	: -20 — 100°C
Mounting	: Any position
Electrical Connection	: Terminal
Winding Temperature (Tmax)	: 200 °C
Direction of Rotation	: CCW
Standards	: RoHS

Electrical Performance:

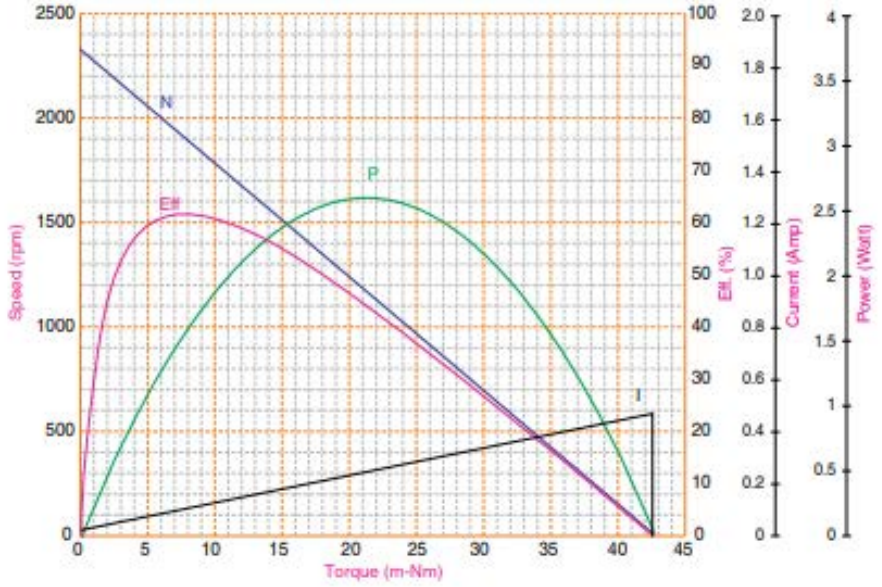
	No Load	Stall	Max Eff	Max Power
Speed (Rpm)	2300		1900	1200
Current (A)	0.023	0.47	0.10	0.25
Torque (mNm)		42.63	7.67	21.32
Efficiency (%)			61.27	
Power (W)			1.53	2.60

Drawing:

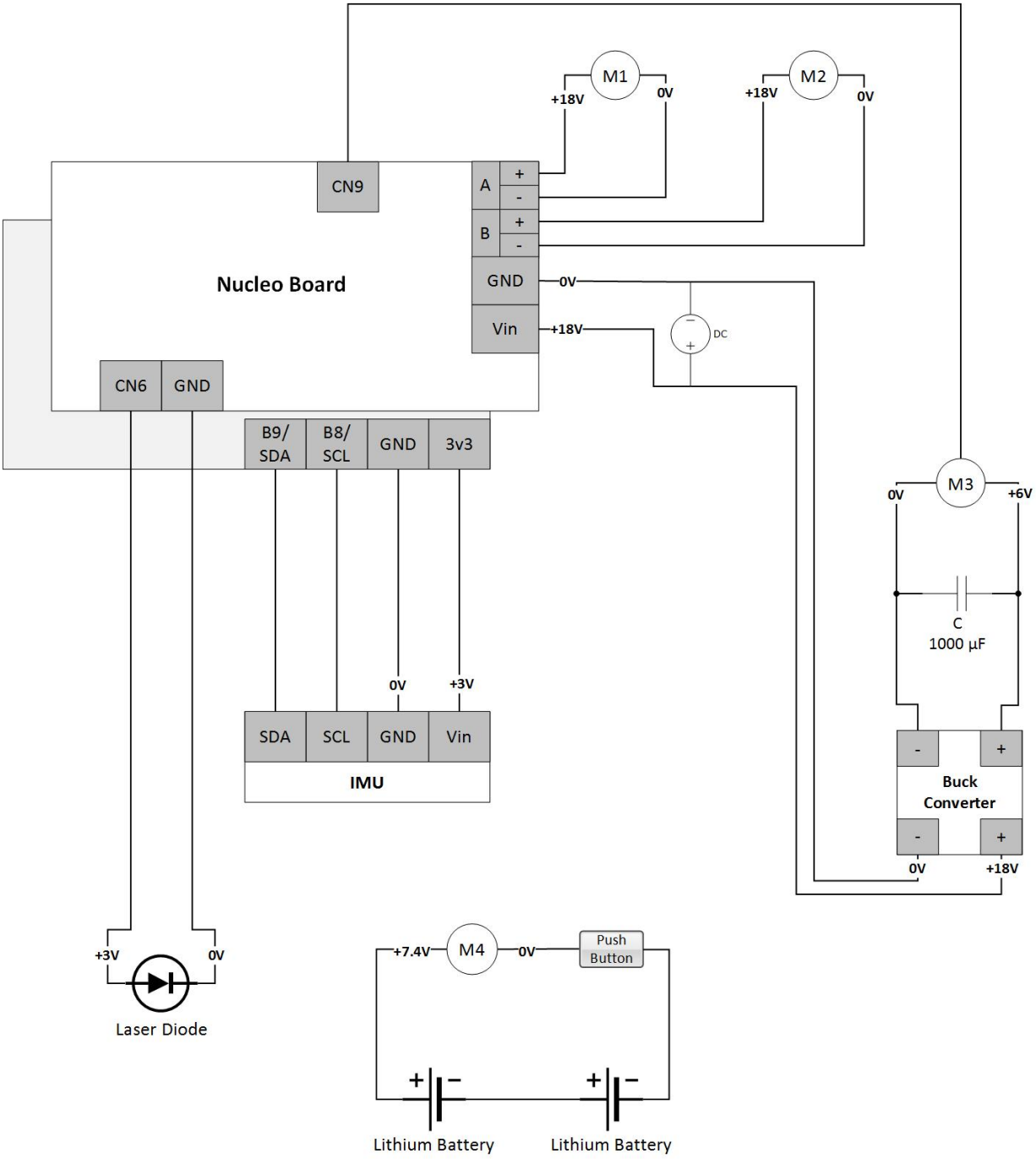


Unit in mm

Performance Data:



Appendix E – Electric Circuit Diagram



mybno055.BNO055 Class Reference

Public Member Functions

def **__init__** (self, i2c, address=DEVICE_ADDR)
Creates an object of the **BNO055** class that is used to read orientation angles. More...

def **calibrate** (self)
Calibrates the IMU by resetting the IMU, setting the heading angle back to 0.

def **set_degrees_units** (self)
Changes the output units to degrees for reading the IMU. More...

def **change_mode** (self, MODE)
Method that changes the mode of the chip. More...

def **get_roll** (self)
Get the roll angle from the BMO055 in degrees, assuming that the accelerometer was correctly calibrated at the factory. More...

def **get_pitch** (self)
Get the pitch angle from the BMO055 in degrees, assuming that the accelerometer was correctly calibrated at the factory. More...

def **get_heading** (self)
Get the heading angle from the BMO055 in degrees, assuming that the accelerometer was correctly calibrated at the factory. More...

Public Attributes

i2c

buffer

mode

Constructor & Destructor Documentation

def mybno055.BNO055.__init__ (self,
i2c,
address = DEVICE_ADDR
)

Creates an object of the **BNO055** class that is used to read orientation angles.

Parameters

i2c an i2c object that is used to communicate between the board and the IMU

address the address of the IMU device

Member Function Documentation

def mybno055.BNO055.change_mode (self,
MODE
)

Method that changes the mode of the chip.

Parameters

MODE the mode to set the chip in

def mybno055.BNO055.get_heading (self)

Get the heading angle from the BMO055 in degrees, assuming that the accelerometer was correctly calibrated at the factory.

Returns

The measured roll angle in heading

def mybno055.BNO055.get_pitch (self)

Get the pitch angle from the BMO055 in degrees, assuming that the accelerometer was correctly calibrated at the factory.

Returns

The measured roll angle in pitch

def mybno055.BNO055.get_roll (self)

Get the roll angle from the BMO055 in degrees, assuming that the accelerometer was correctly calibrated at the factory.

Returns

The measured roll angle in degrees

def mybno055.BNO055.set_degrees_units (self)

Changes the output units to degrees for reading the IMU.

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

- mybno055.py

ProjectClasses.Controller Class Reference

This class creates a PI **Controller** for general purposes. More...

Public Member Functions

def **__init__** (self, kp, ki, kd, setpoint, frq)
This constructor creates the neccesary variables for the controller to function. More...

def **set_setpoint** (self, setpoint)
This method allows a user to put a new setpoint. More...

def **set_kp** (self, kp)
This method allows a user to put a new proportional gain. More...

def **set_ki** (self, ki)
This method allows a user to put a new integral gain. More...

def **set_kd** (self, kd)
This method allows a user to put a new integral gain. More...

def **set_flag** (self, timer)
This method is not used when running through the task lists, but it sets a flag telling the control loop to execute if ran as a call back. More...

def **control_loop** (self, data)
This method computes closed loop controller output, using proportional and integral gain. More...

def **is_done** (self, data)
A method to return if the incoming data is close enough to the setpoint for the controller to be consided done. More...

Public Attributes

kp

ki

kd

setpoint

Detailed Description

This class creates a PI **Controller** for general purposes.

Constructor & Destructor Documentation

def ProjectClasses.Controller.__init__ (self,
kp,
ki,
kd,
setpoint,
frq
)

This constructor creates the neccesary variables for the controller to function.

Parameters

kp	the proportional gain given to the controller
ki	the integral gain given to the controller
setpoint	the setpoint that the controller is trying to reach
frq	the frequency that the control loop runs at

Member Function Documentation

def ProjectClasses.Controller.control_loop (self, data)

This method computes closed loop controller output, using proportional and integral gain.

Parameters

data the incoming data needed to compare with the setpoint

Returns

the output coming from the control loop

def ProjectClasses.Controller.is_done (self, data)

A method to return if the incoming data is close enough to the setpoint for the controller to be consided done.

Returns

a boolean value signifying if control loop is done

def ProjectClasses.Controller.set_flag (self, timer)

This method is not used when running through the task lists, but it sets a flag telling the control loop to execute if ran as a call back.

Parameters

timer is the timer channel needed to run callbacks

def ProjectClasses.Controller.set_kd (self,
kd
)

This method allows a user to put a new integral gain.

Parameters

ki the integral gain given to the controller

def ProjectClasses.Controller.set_ki (self,
ki
)

This method allows a user to put a new integral gain.

Parameters

ki the integral gain given to the controller

def ProjectClasses.Controller.set_kp (self,
kp
)

This method allows a user to put a new proportional gain.

Parameters

kp the proportional gain given to the controller

def ProjectClasses.Controller.set_setpoint (self, setpoint)

This method allows a user to put a new setpoint.

Parameters

setpoint the new setpoint to give the controller

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

- ProjectClasses.py

ProjectClasses.MotorDriver Class Reference

Creates a motor driver for use with the ME405 nucleo board. More...

Public Member Functions

def **__init__** (self, timer, pinENA, pinINP, pinINN, frequ=20000)
Initializes a motor driver object with the specified pins and channels. More...

def **set_duty_cycle** (self, level)
A method to set the speed for the motor using different pulse width percents on the motor timer channels. More...

Public Attributes

ch1

ch2

Detailed Description

Creates a motor driver for use with the ME405 nucleo board.

Constructor & Destructor Documentation

```
def ProjectClasses.MotorDriver.__init__ ( self,
                                          timer,
                                          pinENA,
                                          pinINP,
                                          pinINN,
                                          frequ = 20000
                                        )
```

Initializes a motor driver object with the specified pins and channels.

Parameters

- **timer** a timer number passed in to specifiy which board timer to use for the driver
- **pinENA** the high pin for the motor driver
- **pinINP** pin used for channel 1 of the timer
- **pinINN** pin used for channel t of the timer
- **frequ** the frequency at which the motor driver will run

Member Function Documentation

```
def ProjectClasses.MotorDriver.set_duty_cycle ( self,
                                                level
                                              )
```

A method to set the speed for the motor using different pulse width percents on the motor timer channels.

Parameters

- **level** the speed to set the motor to run at (comes as an output of a control loop

---

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

- ProjectClasses.py

ProjectClasses.Servo Class Reference

This class implements the control of a servo used for actuating the nerf dart into a motor for firing. More...

Public Member Functions

def **__init__** (self, timer, pin, chan)
This method initializes a servo object with the specified pins and channels. More...

def **set_pulse_width** (self, level)
This method allows the user to control the position of the servo. More...

Public Attributes

ch

Detailed Description

This class implements the control of a servo used for actuating the nerf dart into a motor for firing.

Constructor & Destructor Documentation

def ProjectClasses.Servo.__init__ (**self,**
timer,
pin,
chan
)

This method initializes a servo object with the specified pins and channels.

Parameters

timer the number of the timer used on the nucleo board

pin the pin used for the timer channel passed in

chan the channel used with the timer

Member Function Documentation

def ProjectClasses.Servo.set_pulse_width (**self,**
level
)

This method allows the user to control the position of the servo.

Parameters

level the time in us for the pulse width to be high

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

- ProjectClasses.py