WORCESTER POLYTECHNIC INSTITUTE

ROBOTICS ENGINEERING PROGRAM

Developing Multi-Fingered Hand for Dexterous Manipulation

Major Qualifying Project Report

Submitted by:

Duong Nguyen

Hoang Nguyen

Thai Dinh

Date Submitted: 04/25/2018

Project Advisor: Professor Jane Li

Project Co-Advisor: Professor Jie Fu

ABSTRACT

The goal of this project is to develop a multi-fingered robotics hand that is capable of dexterous manipulation at a low cost that is easy to replicate and study. The robotics hand will be used in TRINA system to perform nursing task such as grasping and manipulating deformable objects. A simple and intuitive controlling method for the robot hand is also studied and developed. The robot hand can be controlled using multiple user inputs such as command lines, scripting, GUI and data glove. The software required for the robotics hand and the controlling method will both be available and supported on Ubuntu operating system running the Robotics Operating System (ROS).

Table of Contents

Table of Figures

1. Introduction

1.1. Background

Robots have been created to aid humans in tasks which fall under the three “D” labels - dull, dirty and dangerous. That is not the only use case for robot, however, as technology and science progress, we gradually have more needs for robots to fill in. For instance, robots used in surgical procedure has been emerging over the past 15 years[1]. However, these needs are not completely filled in, as many of our devices are engineered based on the human’s hands. The human’s hand is the result of millions of years of evolution, and thus very complex to fully replicate and translate to a robotics hand. Therefore, there are two generalized schools of thoughts when it comes to robot hand design. You have robot hands that are simple and straightforward and get the job done. And then you have very complex hands with four fingers and a thumb that are designed to closely mimic human hands[2]. The problems, however, with these hands are their complexity in design and cost to manufacture, as the human hand is very complicated to closely mimic. For instance, the Shadow Dexterous Hand is made with 20 actuated degrees of freedom and multiple sensors to perform the closest approximation possible to the human hand[3]. At this level of complexity, the hand will require a lot of computational power to properly control, as well as a very high cost for manufacture. Thus, this verified the second school of thoughts on robot hands: simple and straightforward and get the job done.

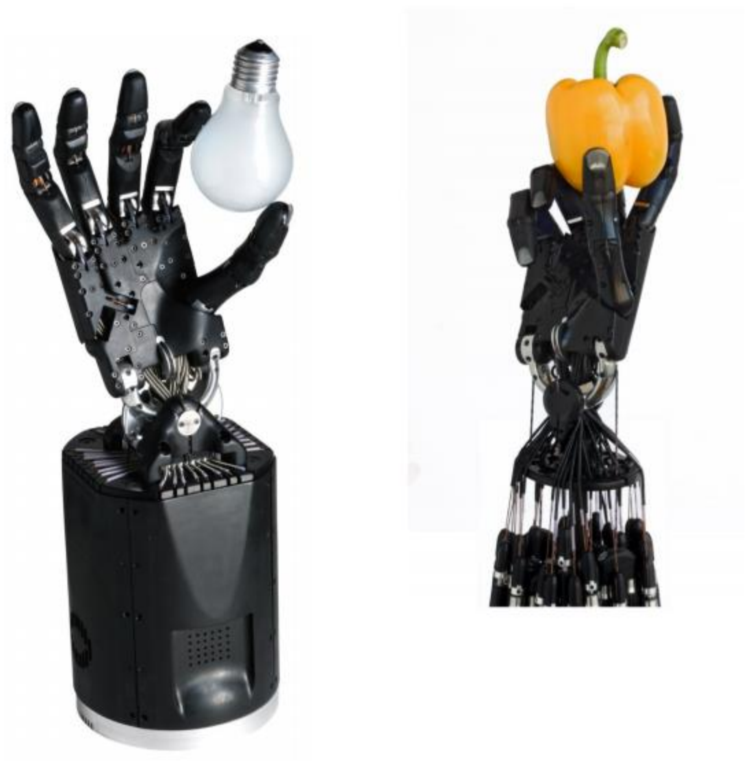


Figure 1-1 Shadow Dexterous Hand[4]

The different types of designs will be discussed further in the upcoming chapter of this report. However, as the school of thought implies, the simple and straightforward robotics hands are mostly gripper type designs, designed for a specific functionality in mind, thus reduce the complexity and cost in comparison to the closely human hand mimic hands.

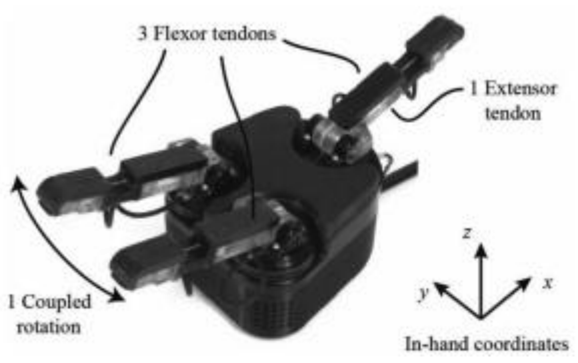


Figure 1-2 iHY Hand[5]

1.2. Motivation

Traditionally, mechanical system has not been able to iterate as fast as software. The reason behind this was the manufacturing down time, as well as the cost of production for mechanical systems. Software, on the other hand, mostly suffer from compiling down time. However, with the help of additive manufacturing, namely 3D-printing, rapid prototyping of mechanical systems has become available. In additional to that, the growth of hobbyist with some professional communities has allowed many open-source project, opening up many opportunities for people to get involved and contribute to such projects.

Of the open-source project, research team at Yale University has been working on a universal hand for grasping and manipulating multiple objects, called the Yale's Open Hand project. The hand is mechanically built with 3D-printed molds, thus can be rapidly modified during production. Since the project is open-source, other organizations are able to contribute to it as well. One of which is Right Hand Robotics Corporation. One of the project's designs, the Model O, is an open-source derivative of the iHY and RightHand Robotics' Reflex Hand design[6].

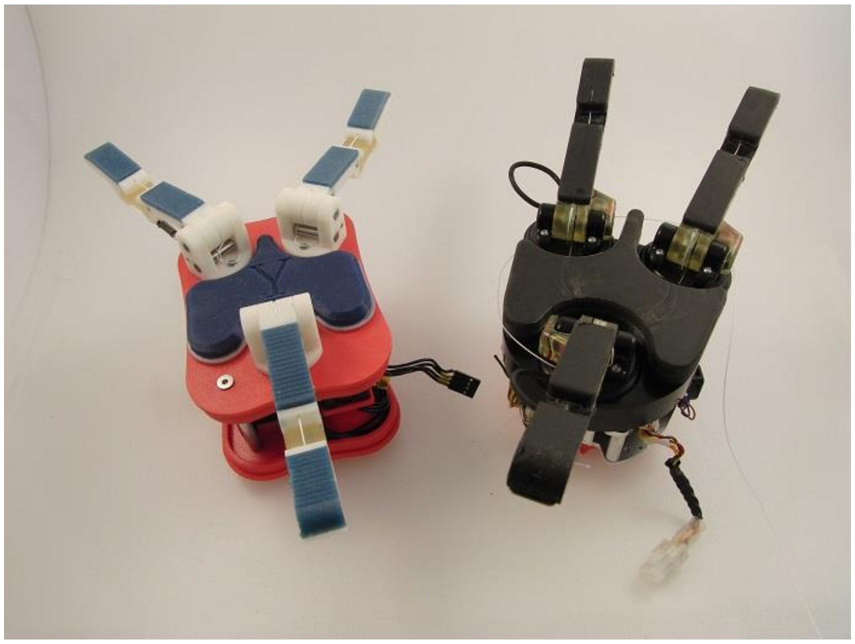


Figure 1-3 The Model O (left) and the RightHand' Reflex (right)

Initially, our team was planning to prototype and improve the design of the Yale's Open Hand Model O for in-hand manipulation. Based on information and guides already available from Yale University, modify and prototype the original design with 3D printing is completely possible. In additional to mechanical improvements, we were going to build and integrate a control interface for the hand as well, such that anyone without an engineering background can pick-up and use the hand. Lastly, our team also wants to lower the cost of the hand. When school began, our project was sponsored by Righthand Robotics, Inc. to develop multi-fingered hand for dexterous manipulation. We based the developments on a newer prototype of RightHand Robotics. The source code and RightHand Robotics design can be found on their company's website.

1.3. Work significance

With the help of 3D printing and of the shelf parts, we would like to create a modular hand that can be easily manufactured, replicated and modified for the purpose of further research required in the future. Our parts will be either directly 3D printed, or molded for elasticity, with molds from the 3D printer of course. The hand's motors are from Dynamixels, which support variety of hardware to directly communicate with the motors and implement into ROS environment. Thus, the hand should be easily replicated and studied at a low cost.

In term of software, the hand is planning to be integrated together with the Duke's Tele-Robotic Intelligent Nursing Assistant (TRINA) system. Therefore, the controlling software interface must be intuitive toward the end-user, and scalable, compatible with another system. In additional to that, we want the software to be scalable and dependent only to the Dynamixel motor type, not the mechanical setup, constraints or number of motors, so that our project can be kept modularity.

Lastly, our team will be looking into a way to intuitively control our final design. As a piece of software would be nothing but an expensive and fancy looking paperweight if not properly controlled. However, the approach to control is an important aspect to consider as well. For instance, a fully autonomous grasp planning system can use computer vision to thoroughly plan out the most optimal approach. However, such a system would require a very long time to set up and study in order to function properly. In the other hand, a traditional control system focusing on the motion of each individual motors would be very hard for a non-technical user to approach, as well as the system can soon be very complicated as the number of motors increase. Therefore, along with the hand, a data glove for capturing the human's hand motion will be developed as mean to directly and intuitively control the motion of the robotics hand in real time.

1.4. Objectives

1.4.1. Mechanical design

Previously, Reflex SF Hand only has four Dynamixel Motors. Therefore, Reflex SF Hand limits motions to only flexion, extension, coupled adduction and abduction of fore two fingers. However, the new prototype has five Dynamixel Motors so we changed the objectives from developing robot hand with four motors for dexterous manipulation to developing robot hand with five motors for dexterous manipulation. We determined the dexterous manipulation tasks for testing based on daily activities, nursing tasks, and tool manipulation.

1.4.2. Control software

The Dynamixel Motors directly implemented to ROS environment like we mentioned before, therefore all the Control Software for the hand would also be developed directly into ROS. The end goal of the project for the Control Software would be a ROS package that able to perform advanced hand manipulation with an easy to use graphical user interface and a glove interface for real-time control.

The current control for the Hand is very simple, open and close using a controller or a button press. Therefore, we would like to develop ways to control the hand for more complicated tasks in-hand manipulation, for example: picking up coins, small needles using recorded data from trial and error, waypoints input or in-hand force manipulation.The user could also see how the hand manipulation performed in RViz simulation before running the hand in real life.

Collecting these data using traditional method, input series of position for the hand to move, can be difficult and time consuming. Therefore, we would like a better method to collect these data by controlling the input of the hand in real-time.These Control and Tuning method would be tested and performed by human hand using special glove.

In addition to glove interface, we would also want to build an easy to use graphical user interface (GUI) for end-user. Controlling the hand by position, velocity control with setting waypoints and calibration all in one GUI.

Furthermore, calibration for these hands are currently non intuitive and time consuming, therefore another objective for this would be Auto Calibration function that works for multiple hands version.

1.4.3. Modularity and scalability

We want to hand designs and control software of the hand can easily be developed more in the future and performed as a foundation of further research. Therefore modularity and scalability are very important objectives that we have.

For Mechanical Design, we want the hand to perform a variety of tasks ranging from daily activities to industrial tasks. Therefore, the design of each modular finger needs to take into account the purpose of each finger and its contribution to the overall performance of the robot hand. Scalability of Mechanical Design aspect lies in the various combination of three modular fingers to achieve many difficult tasks.

For Control Software aspect, we want the software to not only run on our hand but also other model of hand using Dynamixel motors only depend on the Motor type and not mechanical setup and number of motor constraints. Therefore the user interface and glove interface would also changed and compatible with different setup of Dynamixel motors Daisy Chain with minimal user specification. Not only running the software with different hand, we want the software to be compatible to control multiple hand at once, which is the modularity and scalability of the software aspect.

1.4.4. Data glove

Another goal to pursue is to develop a data glove that is capable of intuitively control the robotics hand in real time. Like the robotics hand itself, the data glove is aimed to be manufactured using off the shelf parts with open source design and integration, such that other scholars can replicate and continue to study the work based easily. Moreover, the system must be compatible with Linux and ROS, as these are some of the most popular and supported platform for robotics system development. Not to mention the TRINA system that was originally intended to be tested on is developed with ROS on Ubuntu.

2. Related Work

2.1. Robotics Hand

2.1.1. Rigid robot hand

As the name implies, this category of robotics hand consisted mostly of rigid links with every joint driven by a motor. These robot hands within this category can range widely from a simple gripper to a complex hand like the Shadow showed above. The disadvantage of this system, however, is that it required some extensive planning in order to grasp an object properly. Since each joints of this type of hand is controlled by a motor for a complicated design, the motors must be controlled to operate in sequence in order to generate the desirable grasping motion. For the case of a simpler gripper, the gripper needs to position itself properly before being able to grasp the object.

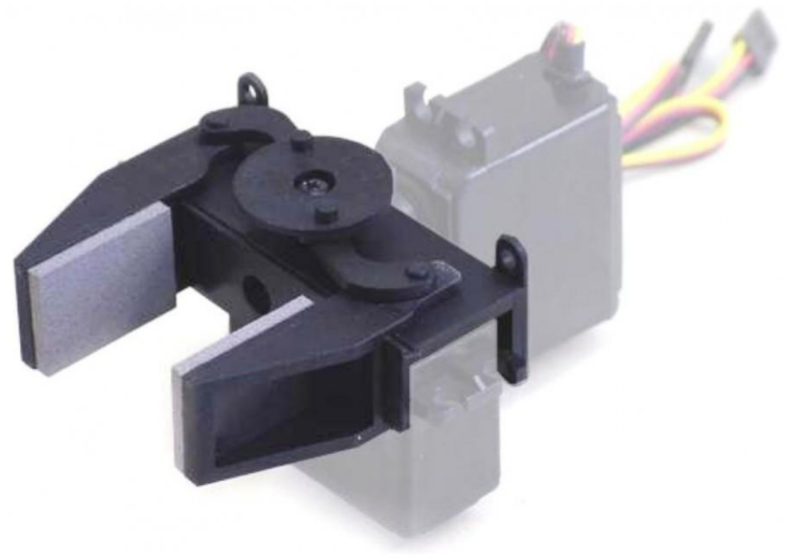


Figure 2-1 Single servo driven gripper

Another drawback is that since the links are all rigid, it is hard for the controller to receive feedback on the grasping motion, whether the object has been grasp successfully or not. This leads to the issue of the inability to pick up soft objects, or the manipulator itself would require additional force sensors or a visual system that can detect the deformation of the object in order to proceed. With all of these adding up, the rigid robot hand can be either a relatively cheap and simple system that can do certain tasks, or a very expensive and complicated system that would require an extensive processing power for proper control.

2.1.2. Compliant robot hand

Looking to solve the soft contact issue and stiff grasp approach of the rigid hand, engineers and scientists alike started developing more flexible, soft robot hand to solve the issue. Some popular result would be a soft robotics actuator, mostly tentacle shape, powered by a pneumatic system picking up some soft objects, such as an egg or tomato.

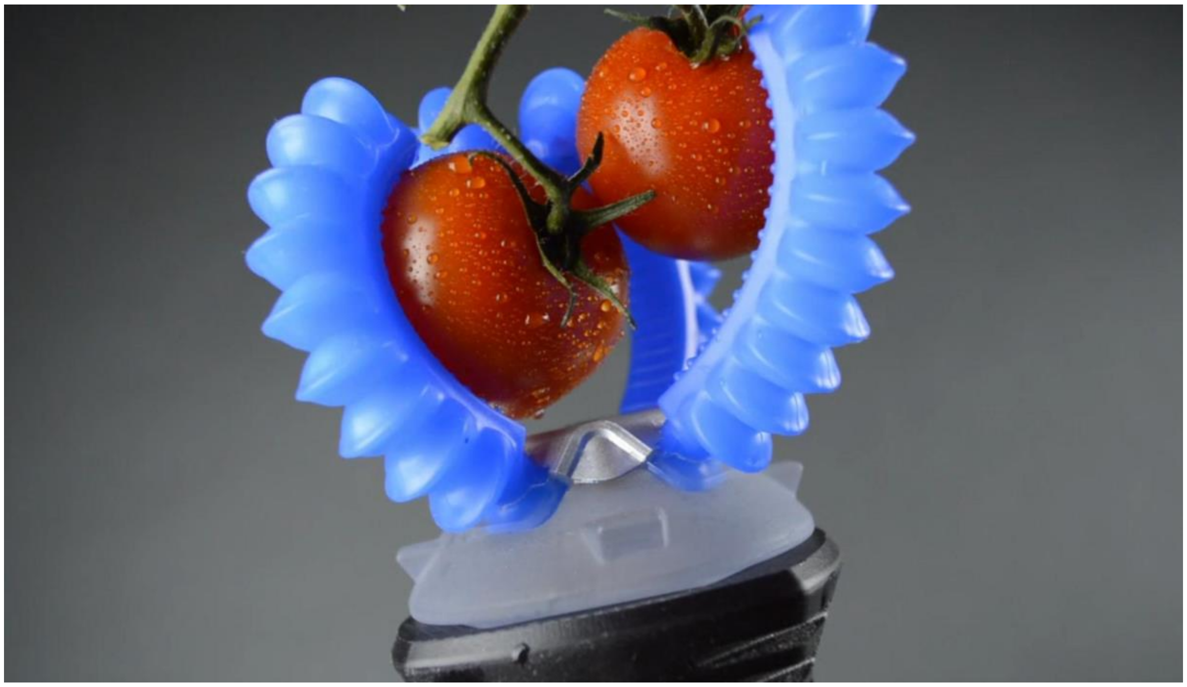


Figure 2-2 Soft Robotics Inc.'s gripper holding some tomatoes[7]

Some advantages for this compliant robotics gripper including gentle grasping and the planning does not require extensive processing power. Since the joints are flexible, they can deflect themselves to fit the object grasping, thus allowing higher tolerance and more flexibility to grasp planning to their rigid joints alternatives.

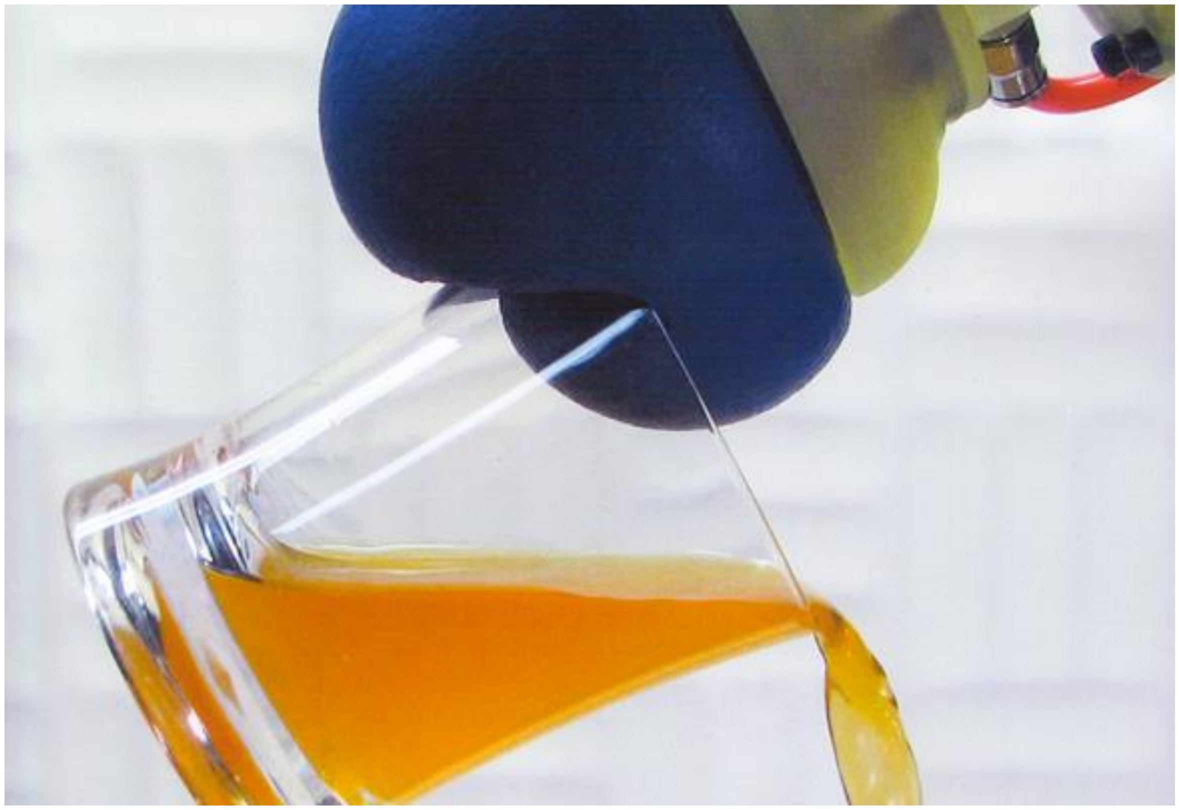


Figure 2-3 Universal Jamming Gripper picking up a glass of juice[8]

Despite the advantages, there are certainly drawbacks within these systems. Each joints still require a certain level of control in order to work properly, moreover those are pneumatics controllers, which is much more complicated than a motor to control properly.

2.1.3. Under-actuated compliant robot hand

As an attempt to combine between the soft compliant and the rigid joint together, the development for under-actuated compliant robotics hand started. The term under-actuated means that the robotic hand will have less number of motors than the number of joints, then compensated for that with compliant joints. This design makes room for many unique behavior combinations depending on the orientation and setup of the compliant joints with the driven motors. This approach of lesser number of motors helps reduce the complexity required for controlling system. Some popular researches focusing on this category of robotics hand include the previously mentioned Yale's GrabLab, RightHand Robotics.

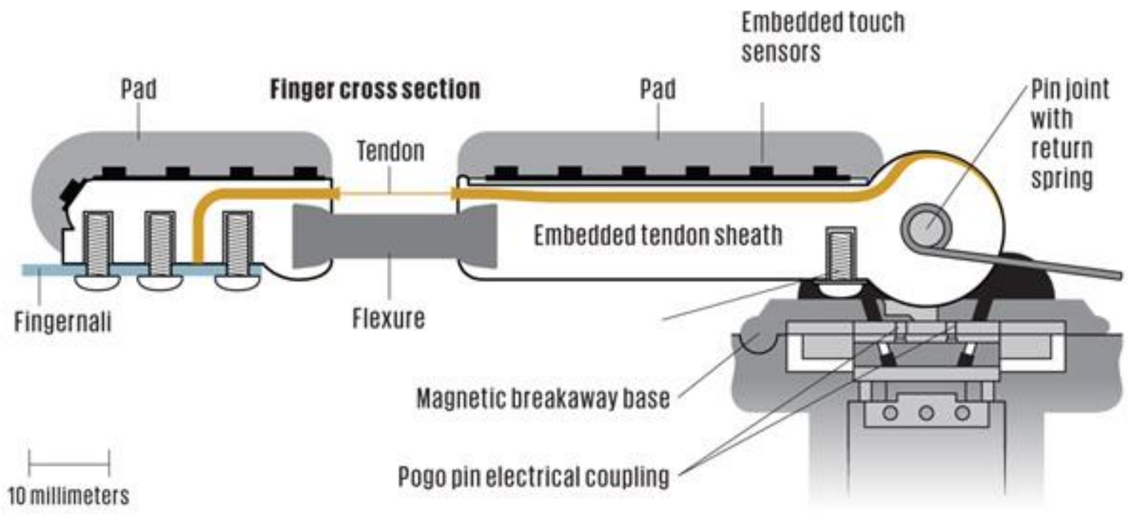


Figure 2-4 A compliance finger design

Above is the design diagram for a compliance finger design from RightHand Robotics. Each finger will be driven by a single motor only. However, due to the design and the stiffness different in the pin and the flexure joint, the finger can be controlled to different location depending on the end condition. For example, the finger can bend more to grab an off centered object from the palm of the robotics hand.

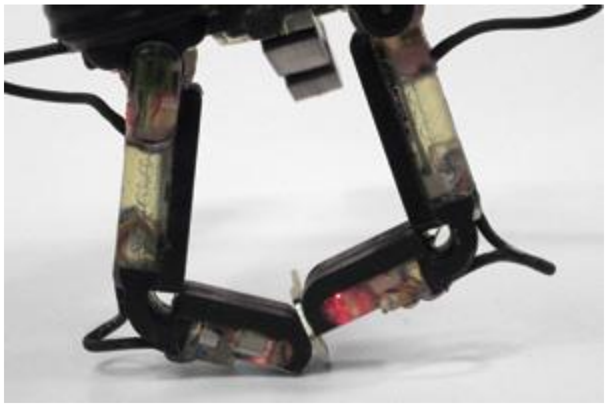


Figure 2-5 The same fingers would deform differently to grasp different objects[9]

This design allows a more flexible grasping approach, less control complexity due to smaller number of motors thus ultimately reduce the cost required to make this hand. However, there are certain drawbacks for this design approach. The behavior of the finger would actually depend on the stiffness different between the two joints. Therefore, it would be hard to simulate the correct behavior of the finger if the stiffness is not studied properly. This leads to another issue of stiffness variance during the manufacturing process of the flexure joint that will be discussed further in later chapter. In short, various manufacturing conditions can lead to a big variance in stiffness within the fingers, making it hard to correctly simulate the behavior of a finger on a software.

Another drawback is since the joints are flexible, it needs to retain its shape in order to function correctly. Therefore, the storing and operating conditions are very important to assure the functionality of the robotics hand, as once a finger is permanently deformed, the hand will be unusable. Furthermore, the finger assembly overall is very complicated as the motors need to be setup properly to drive the fingers. This process will be discussed further in the following chapters.

2.2. Data glove

Due to the advancement of virtual reality technology, the development of data glove is also on the rise. The purpose of these gloves are to capture the motion of the human hand,

and transfer that information to a processor to simulate those motions within the virtual reality realm. As a glove allows higher precision capture and more intuitive than a remote controller, the VR gloves are becoming more and more popular. However, since VR technology is still currently in its early stage, this is also the case for most data glove developer companies.

Figure 2-6 Virtual reality smart glove offered by CaptoGlove

The early stage for these VR gloves means that they provide limited support for developers who would want to use the glove for the purposes other than gaming and virtual reality. Moreover, virtual reality technology is currently centralized around Microsoft Windows platforms, so it is unsurprisingly that Linux support for these gloves are close to non-existent. Not to mention the lack of a community to help develop an open source, off the shelf glove design that can be replicated and study. This proves to be a somewhat challenging task for the team.

3. Methodology

3.1. Mechanical design and manufacturing

In Reflex SF design, we noticed Righthand Robotics used a single motor to drive a gear mechanism for coupled adduction and abduction motion of fore fingers. In the new prototype, the coupled motion is replaced with a motor for each fore finger so fore fingers can individually perform adduction and abduction motion as well as flexion and extension. However, the thumb still can only perform flexion and extension. According to [citation], thumb accounts for approximately 50 to 60 percent in performance of hand. Therefore, we decided to move a motor from fore fingers to the thumb and design a gear mechanism for coupled motion of fore fingers as in Reflex SF. As a result, now fore fingers have one and a half degrees of freedom and thumb have two degrees of freedom.

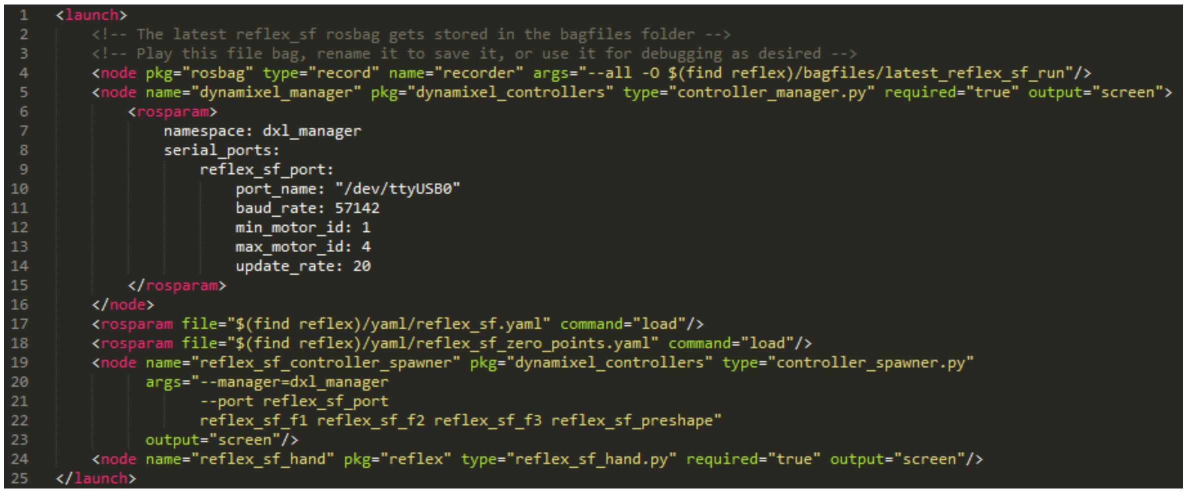
Since the motor used in the newer prototype is XL-320 instead of MX-28T, the torque is decreased. Therefore, we decided that the ratio of velocity reduction in the new gear mechanism should be less than or equal to the ratio of one in Reflex SF. The equations that used to determine gear parameters are listed under the GitHub repository under the Appendix.

To manufacture and test our designs, we chose 3D printing so we could check errors and interference in the designs then modify the designs quickly. We planned to have our 0th prototype by the end of first term to have basic knowledge of how well the 3D-printer printed our design. From the inspections of our 0th prototype, we planned to have 3 iterations during our second term and the 3rd iteration would be the final one. However, we could accomplish 2 iterations during the second term and push the finalization to our third term. Details of iterations are discussed from section 4.2 to section 4.5.

3.2. Control software

The control software for the Reflex SF ran entirely in ROS environment in a Linux machine, as mentioned earlier. The software consists of two different parts:

- Dynamixel controller: a lower level package to directly request data from the Dynamixel servo, or write data to the servo for operation. This package took care of scanning for servos, establishing UART channel and handling communication protocol.
- Reflex controller: a higher level package ran on top of the Dynamixel controller that handle basic functions of the Reflex robotics hand. The package provides a high level control over multiple Dynamixel controllers to provide the user with a single controller for the whole hand.



```
<launch>
    <!-- The latest reflex_sf rosbag gets stored in the bagfiles folder -->
    <!-- Play this file bag, rename it to save it, or use it for debugging as desired -->
    <node pkg="rosbag" type="record" name="recorder" args="--all -O $(find reflex)/bagfiles/latest_reflex_sf_run"/>
    <node name="dynamixel_manager" pkg="dynamixel_controllers" type="controller_manager.py" required="true" output="screen">
        <rosparam>
            namespace: dxl_manager
            serial_ports:
                reflex_sf_port:
                    port_name: "/dev/ttyUSB0"
                    baud_rate: 57142
                    min_motor_id: 1
                    max_motor_id: 4
                    update_rate: 20
        </rosparam>
    </node>
    <rosparam file="$(find reflex)/yaml/reflex_sf.yaml" command="load"/>
    <rosparam file="$(find reflex)/yaml/reflex_sf_zero_points.yaml" command="load"/>
    <node name="reflex_sf_controller_spawner" pkg="dynamixel_controllers" type="controller_spawner.py"
          args="--manager=dxl_manager
                --port reflex_sf_port
                reflex_sf_f1 reflex_sf_f2 reflex_sf_f3 reflex_sf_preshape"
          output="screen"/>
    <node name="reflex_sf_hand" pkg="reflex" type="reflex_sf_hand.py" required="true" output="screen"/>
</launch>
```

Figure 3-1 The launch file of controlling software

From the launch file above, the Reflex package is built upon the lower level Dynamixel controller and thus has to be called first. Once all of the Dynamixel packages have been successfully initialized, the Reflex package can then initialize and allow the user to fully control the hand. From the packages, the proper way to control the hand is to use the terminal command.

```
rostopic pub /reflex_sf/command_position reflex_msgs/PoseCommand "f1: 1.0
f2: 1.0
f3: 1.0
preshape: 0.0"
```

Figure 3-2 Position command message template from terminal line

This would be a pretty slow and manual way to send out a message. Therefore, the goal of this project was also to improve the controlling package for the Reflex hand by implementing a GUI allow easier and more intuitive control of the hand. This new package will be built with the idea of modularity and compatibility in mind as well. Meaning it can work with different numbers and models of motors.

3.3. Data glove

With our developed GUI, we can control our robot hand easier than just using terminal command. However, for motion that require continuous and complicated motion for example some in-hand manipulation that rotate a cylinder object. To test and study way to do this motion, using the GUI alone would be difficult. Therefore, our idea is to have an input device that able to control the robot hand easier. And what could be easier than our hand itself. There are multiple ways to map our hand motion into number so that we can control the robot hand.

First solution is using computer vision which is similar to the system that we used to map our arm motion to Baxter. This solution has benefit as it was already implemented in the lab. However, one drawback of this is if we used the current setup which is used for the arm, we also need to change it to fit with the finger. Another problem with this solution is that our fingers are usually couple and in the way of others finger, therefore camera had difficult time capture motion of our fingers. Therefore, this solution actually had a lot of drawbacks with the current set up in the lab. Also If the robot hand was develop for nursing purpose, so the nurse need to able to control Baxter and the robot hand at the same time.

Because of the limitation of the first solution, we came up with the second solution which used direct bending motion of our finger to number to control the robot hand. The solution was to use sensor, and more specific, flexible sensor, to measure the bending motion of our hand. First we looked at product on the market to see if there was already some product that had these feature. There are a few of these glove, and Capto Glove was one of them. At first we thought buying an existing product with pre-build sensors and software development kit would be easier to implement, however, later on we discover a lot of limitation of these glove, and we decided to build our own glove to go with the robot hand. The section below will summarize our process and implementation, benefit and disadvantage of each solution.

4. Mechanical design iteration

4.1. Changing servos

In the original design of the Reflex SF by RightHand Robotics, the motors of choice are four MX-28 servos: three motor to thread the fingers' tendon and the last one to couple the two front fingers together using a gear box.

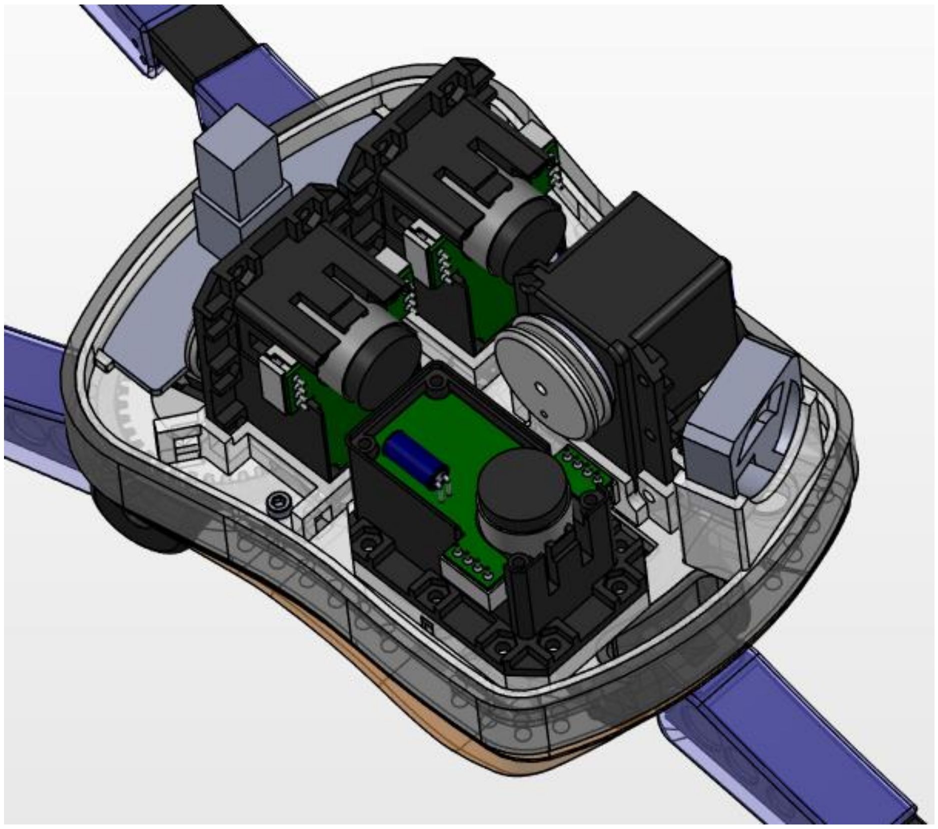


Figure 4-1 Reflex SF with 4 MX-28 servos

The Dynamixel servo line was originally chosen since it provides a lot of functionality in one package: a DC motor connected to a built-in speed reduction gear system, a circuit board for motor control with a provided driver firmware that can be configured using a computer software from Robotis and two TTL communication board on each motors for communication. The TTL channel is meant for connecting multiple servos together and provide a single communication line for all of the servos. The on board configuration of the servos also allows different baud rate for different communication purposes. This TTL line also provide power to the servos as well. In short, the Dynamixel servos provide many functionalities of the shelf with the internal circuit board and the TTL communication channel provides a simplify method for a single communication channel, which greatly reduce the load onto the main controller.

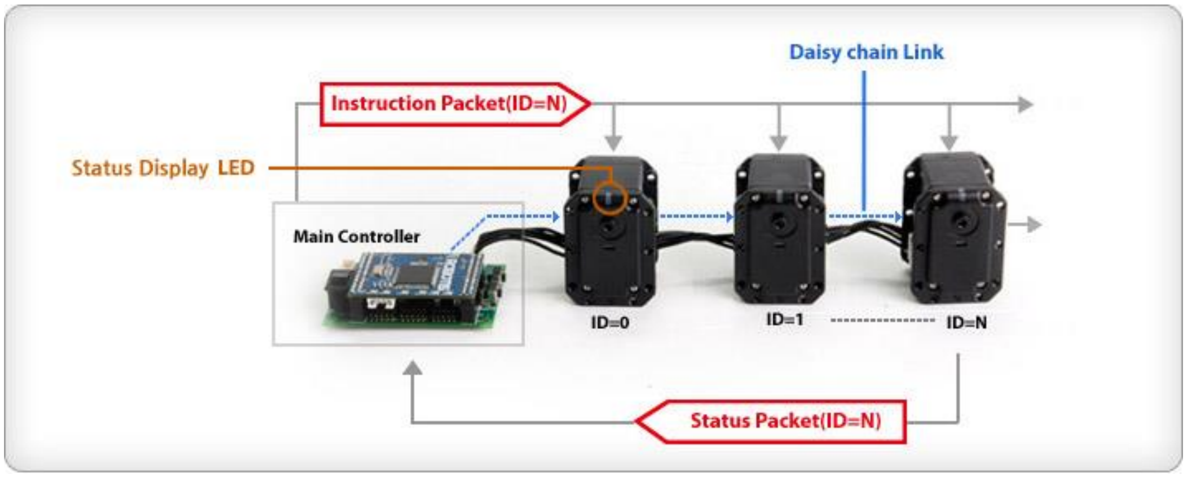


Figure 4-2 Summary of the TTL communication channel from Dynamixel

The original chosen servo to operate the Reflex hand is the Dynamixel MX-28. The motor is compact, provides multiple functions with high maximum torque and speed output as well. However, the tradeoff is each motor costs around $200.

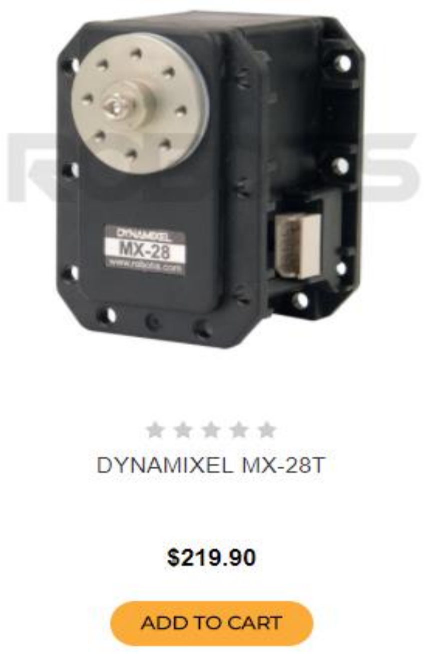
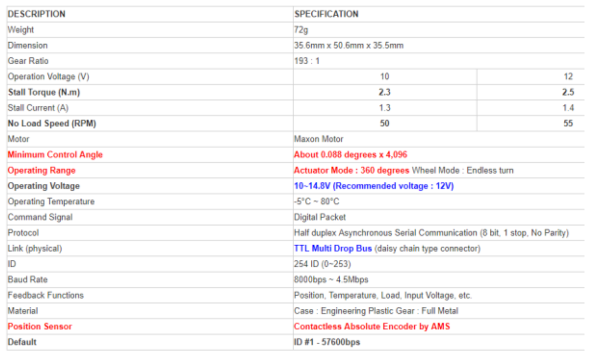


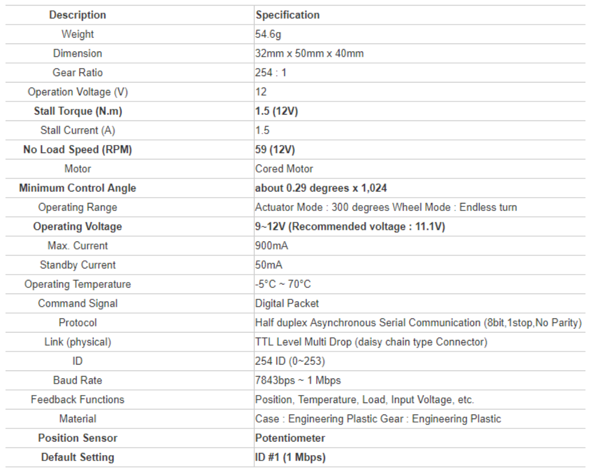
Figure 4-3 Price of a Dynamixel MX-28T servo from Robotis

As the purpose of this project is to create a replicable hand for research purpose, such that different scholars can achieve the same study as well. A finished robotics hand would require at least 4 motors currently to achieve necessary dexterity, thus would make the cost of only motors to around $1000. It would be way too much for many to successfully replicate. Therefore, the servos need to be changed to a lower cost, more affordable servo that can provide the closest specifications as possible. Of the possible choices, the AX-12A seems to be a good candidate.



DESCRIPTION	SPECIFICATION	
Weight	72g	
Dimension	35.6mm x 50.6mm x 35.5mm	
Gear Ratio	193 : 1	
Operation Voltage (V)	10	12
Stall Torque (N.m)	**2.3**	**2.5**
Stall Current (A)	1.3	1.4
No Load Speed (RPM)	**50**	**55**
Motor	Maxon Motor	
Minimum Control Angle	**About 0.088 degrees x 4,096**	
Operating Range	**Actuator Mode : 360 degrees** Wheel Mode : Endless turn	
Operating Voltage	**10~14.8V (Recommended voltage : 12V)**	
Operating Temperature	-5°C ~ 80°C	
Command Signal	Digital Packet	
Protocol	Half duplex Asynchronous Serial Communication (8 bit, 1 stop, No Parity)	
Link (physical)	**TTL Multi Drop Bus** (daisy chain type connector)	
ID	254 ID (0~253)	
Baud Rate	8000bps ~ 4.5Mbps	
Feedback Functions	Position, Temperature, Load, Input Voltage, etc.	
Material	Case : Engineering Plastic Gear : Full Metal	
Position Sensor	**Contactless Absolute Encoder by AMS**	
Default	**ID #1 - 57600bps**	

Figure 4-4 MX-28 specification



Description	Specification
Weight	54.6g
Dimension	32mm x 50mm x 40mm
Gear Ratio	254 : 1
Operation Voltage (V)	12
Stall Torque (N.m)	**1.5 (12V)**
Stall Current (A)	1.5
No Load Speed (RPM)	**59 (12V)**
Motor	Cored Motor
Minimum Control Angle	**about 0.29 degrees x 1,024**
Operating Range	Actuator Mode : 300 degrees Wheel Mode : Endless turn
Operating Voltage	**9~12V (Recommended voltage : 11.1V)**
Max. Current	900mA
Standby Current	50mA
Operating Temperature	-5°C ~ 70°C
Command Signal	Digital Packet
Protocol	Half duplex Asynchronous Serial Communication (8bit,1stop,No Parity)
Link (physical)	TTL Level Multi Drop (daisy chain type Connector)
ID	254 ID (0~253)
Baud Rate	7843bps ~ 1 Mbps
Feedback Functions	Position, Temperature, Load, Input Voltage, etc.
Material	Case : Engineering Plastic Gear : Engineering Plastic
Position Sensor	**Potentiometer**
Default Setting	**ID #1 (1 Mbps)**

Figure 4-5 AX-12A specification

From the side by side comparison chart, the AX-12A has all of its dimensions either lesser than or equal to the original MX-28. This would be a great advantage for servo change as a more compact servo would result in more internal space for the hand design. Additionally, the servo is lighter as well. Both of them would take the same operating voltage. There are several tradeoffs, however. Firstly, the AX-12A is not equipped the same encoder method as the MX-28, as well as the angle resolution is much larger as well. However, since the purpose of the AX-12A will be pulling the tendon thread to open and close the fingers and not direct driving any joints, the angle precision should not post too big of a problem toward the overall hand performance. Secondly, the gear box of the AX-12A is made out of plastic instead of full metal gear like the MX-28. As the stall torque on the AX-12A is lower, there should be no need for such a metal gear. Now lastly, there is a difference between the stall torque and free rotating speed between the two motors.

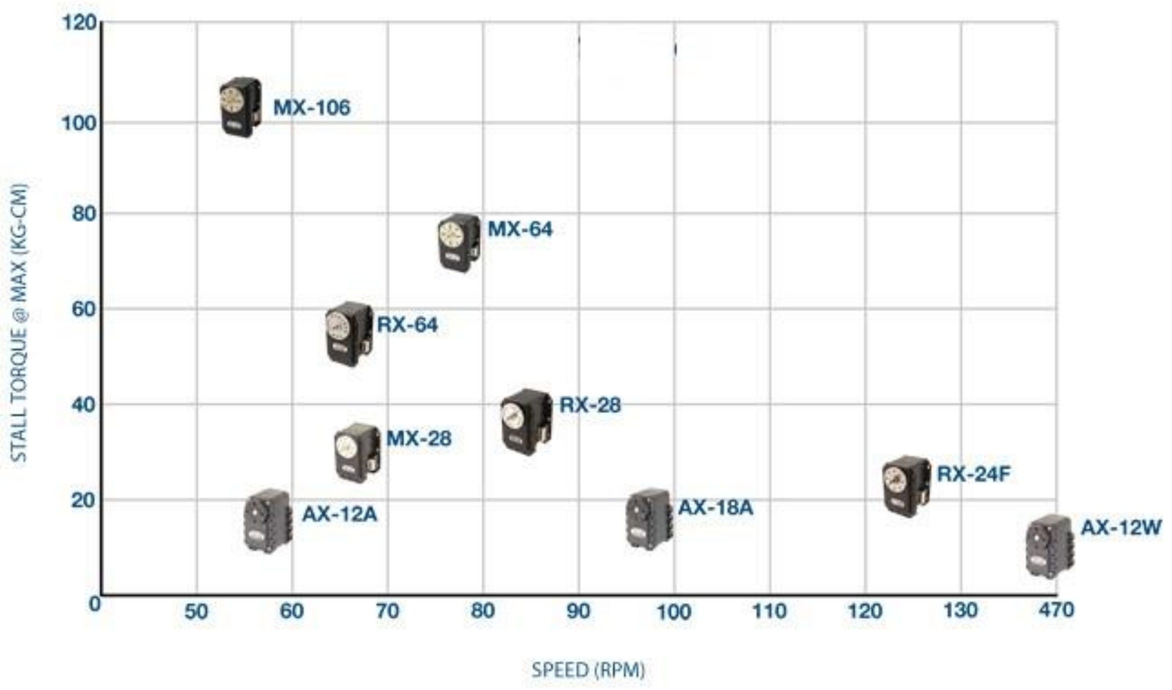


Figure 4-6 Speed Vs. Stall torque performance of various Dynamixel servo models

From the picture above, it is clear that the AX-12A model has both lower stall torque and free rotating speed. For the sake of pulling the tendon, the desirable servo really does not require much speed at all, so the lower free rotating speed should not affect the overall performance of the robotics hand at all. The stall torque, on the other hand, would be a considerable factor for servo sourcing. Based on the datasheet and diagram above, the MX-28 has almost twice the amount of stall torque as the AX-12A. If this servo were to be used, some sort of speed reduction would be required to increase the torque output for normal rotational operation. For pulling on the tendon for finger control would result in a weaker finger considering the low torque, as there is really nothing can be done to improve this. This might be a big tradeoff, but considering the price of one AX-12A is only $45, more than 5 times cheaper than the MX-28.

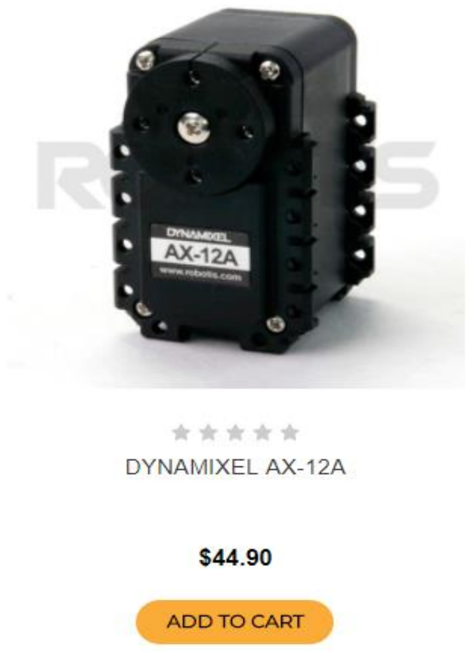


Figure 4-7 Price of one Dynamixel AX-12A from Robotis

Moreover, considering this hand is aiming to be used as a researching tool for dexterous in-hand manipulation, it would not be required to handle heavy objects. Thus, making the lower torque AX-12A servo an eligible choice for servo.

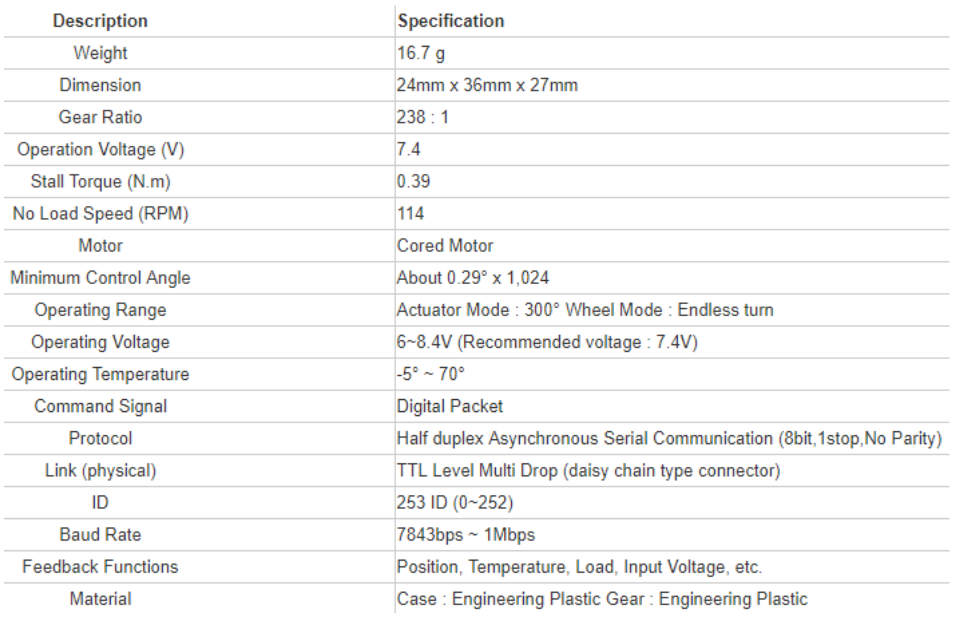
As good as a choice for tendon actuator the AX-12A is, it would be impossible to use this servo for direct driving the base of a finger considering the current hand design and layout of the servo. In order for a direct drive at the base of a finger to be possible, there must be a hole through the motor for the tendon to go through and tied onto the finger. Luckily, of the lineup of servos offered by Dynamixel, the XL-320 offered this unique choice. The XL-320 is also offered at a very tempting price point of around $20.

Figure 4-8 XL-320 offered by Robotis

The screw securing the main pulley can be taken off, revealing a through hole going from the back to the front of the servo, allowing a tendon thread to be put through.

[Insert actual mounting of the AX-12A and XL-320 with tendon thread pulled through]

Now the dimension of the XL-320 is even smaller and much lighter than the AX-12A. The lower weight and smaller dimension will help reduce a lot of space constraint and stress concentration due to the weight of the system as a whole.



Description	Specification
Weight	16.7 g
Dimension	24mm x 36mm x 27mm
Gear Ratio	238 : 1
Operation Voltage (V)	7.4
Stall Torque (N.m)	0.39
No Load Speed (RPM)	114
Motor	Cored Motor
Minimum Control Angle	About 0.29° x 1,024
Operating Range	Actuator Mode : 300° Wheel Mode : Endless turn
Operating Voltage	6~8.4V (Recommended voltage : 7.4V)
Operating Temperature	-5° ~ 70°
Command Signal	Digital Packet
Protocol	Half duplex Asynchronous Serial Communication (8bit,1stop,No Parity)
Link (physical)	TTL Level Multi Drop (daisy chain type connector)
ID	253 ID (0~252)
Baud Rate	7843bps ~ 1Mbps
Feedback Functions	Position, Temperature, Load, Input Voltage, etc.
Material	Case : Engineering Plastic Gear : Engineering Plastic

Figure 4-9 Specification on the XL-320

The XL-320 offered much lower stall torque as well no load speed. The speed will not be important as discussed earlier, and so would be the torque in this case. As the finger will be operated mostly from the tendon, the base will only need to position the finger so it can get to work properly. Therefore, the stall torque should not be an issue, as long as the finger design is light enough for the motor to drive. However, this servo does not offer the same operating voltage range as the AX-12A or the MX-28. This would be an issue but should be fixable within the electrical circuit design.

4.2. Prototype 0th

4.2.1. Objectives and description

Our main objective for prototype 0th is to understand how the prototype parts assemble and properties of chosen 3D printer. The model of prototype 0th in Solidworks is shown in figure 4.10.

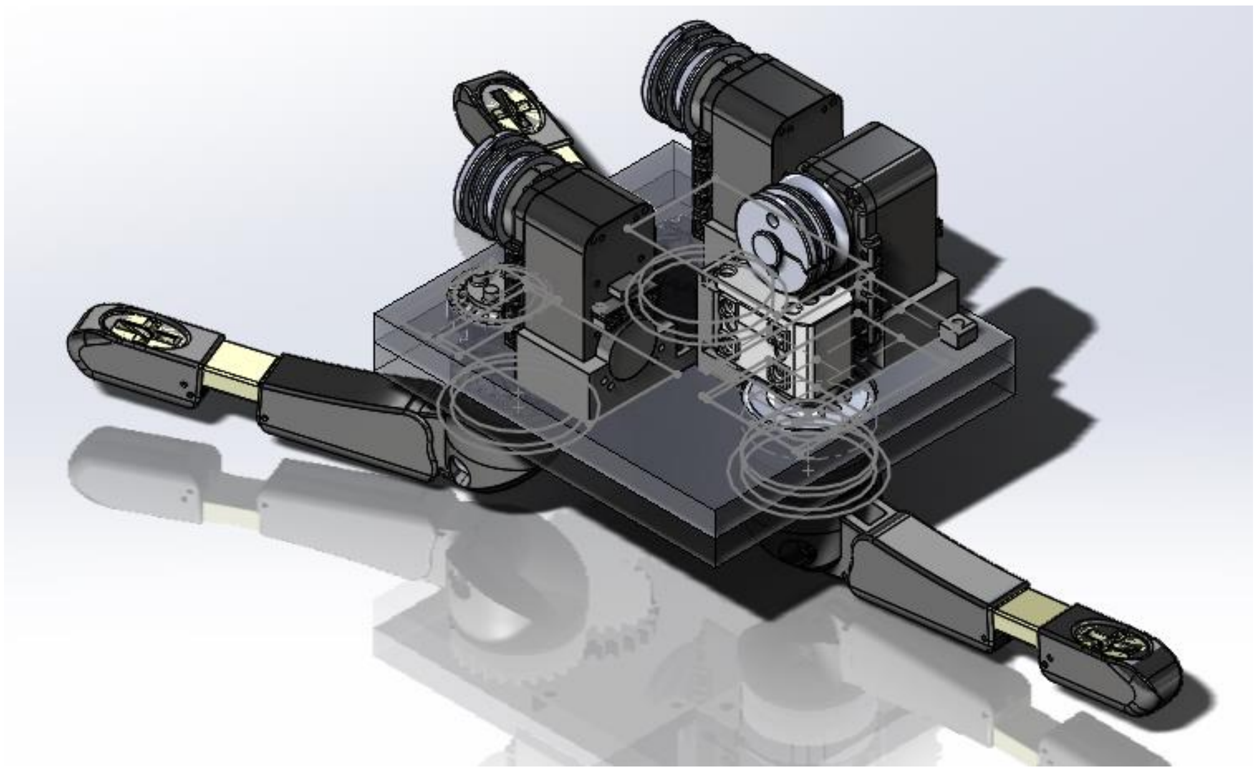


Figure 4-10 Prototype 0th Model

From our background research, Blair and Kramer (1981) state that thumb accounts for 40 percent of hand motion. In addition, from real-world observation, the index and middle fingers are coupled together. Therefore, we considered to use the gear train mechanism as in Reflex SF model to couple the forefingers. As a result, we could use the extra motor, Dynamixel XL-320, for another degree of freedom in thumb of the robot hand. Prototype 0th shows resemblance of Reflex SF model but the thumb of prototype 0th had one more degree of freedom than the thumb of Reflex SF model. We would discuss results and observations of prototype 0th from assembly in the next section.

4.2.2. Results and observations

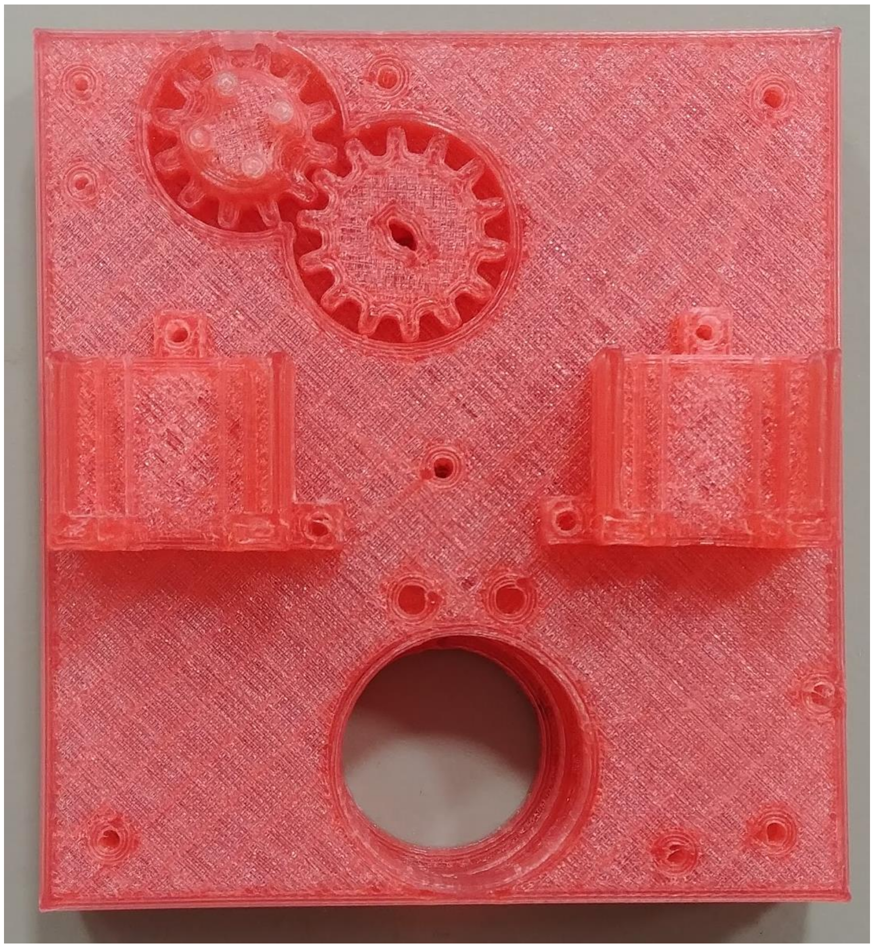


Figure 4-11 Prototype 0th Assembly

Figure 4.11 shows printed parts of prototype 0th. From observations, after printed, the design of prototype 0th had many rooms for improvements. Firstly, we observed that holes were misaligned. Holes on bottom base are shifted compared to the holes on top base (easily detected from figure 4.11). Secondly, the gear train was not connected together. The reason that leads to this failure in gear train assembly is the gears were not fixed in its place therefore there was space for gears to shift inside the holes. As a result, torque transmission between driving gears and gears to drive fingers is reduced. During testing, the driving gear could not drive gears on finger because the gears were not in contact with each other. Thirdly, the string tendon that connects motor pulley to fingers was not vertical due to position of motor mounts on base. The string was in contact with holes inside both Dynamixel XL-320 motors and hand base. This would increase friction on string and affect the adduction and abduction movements of fingers. Finally, the angular velocity reduction of prototype 0th's gear train was 0.75. This ratio was higher than the ratio of Reflex SF's gear train. In addition, the driven motor of prototype 0th, which is Dynamixel XL-320, has smaller torque so in the next iterations, we planned to reduce the angular velocity reduction ratio even further.

4.2.3. Modifications for next iteration

From the observations, firstly we decided to have a meeting with the person who was in charge of 3D printing our parts to improve the print quality first. After the meeting, we understood that the small dimension of our design (in millimeters) caused the shift in hole alignment so we planned to invest in better 3D printing equipment, namely 3D printing nozzle on millimeter scale to have better print quality. On mechanical design aspect, we planned to re-design the gear train to achieve a reduction ratio of 0.5. According to equations of angular velocity reduction in "Speed reduction calculation" figure in section 3.1 the reduction ratio depends on gears' teeth and diameters. Initially, the gear train consisted of 6. The lower bound for distance between two gears on fingers was determined by the diameter of the intermediate gears and the upper bound was determined by the shape of the base. The compact design of robot hand lead to many constraints for how large the base was. We utilized this advantage, made a program using Python script in appendix which took into account for the constraints then output the diameters of gear train which satisfied the velocity reduction ratio of 0.5. After changing configurations in the program, we found that we could achieve the reduction ratio with less number of gears by directly connecting the driven gear on the top base to an intermediate gear on the bottom plate and having the intermediate gear driven one of the gears on fingers. From this result, we could also achieve a more compact design than prototype 0th design. Furthermore, we also considered the problems of size and positions of gear holes and vertical string tendon for the next iteration. We will discuss the details of iteration 1 in the next section.

4.3. Iteration 1

4.3.1. Objectives and description

Our main objective for iteration 1 design is to address the problems we found in prototype 0th and have a working model by the end of the second term. By "a working model", we mean that the assembly with fingers would run smoothly and accordingly with the GUI controller and CaptoGlove. The model of iteration 1 in Solidworks is shown in figure 4.12.

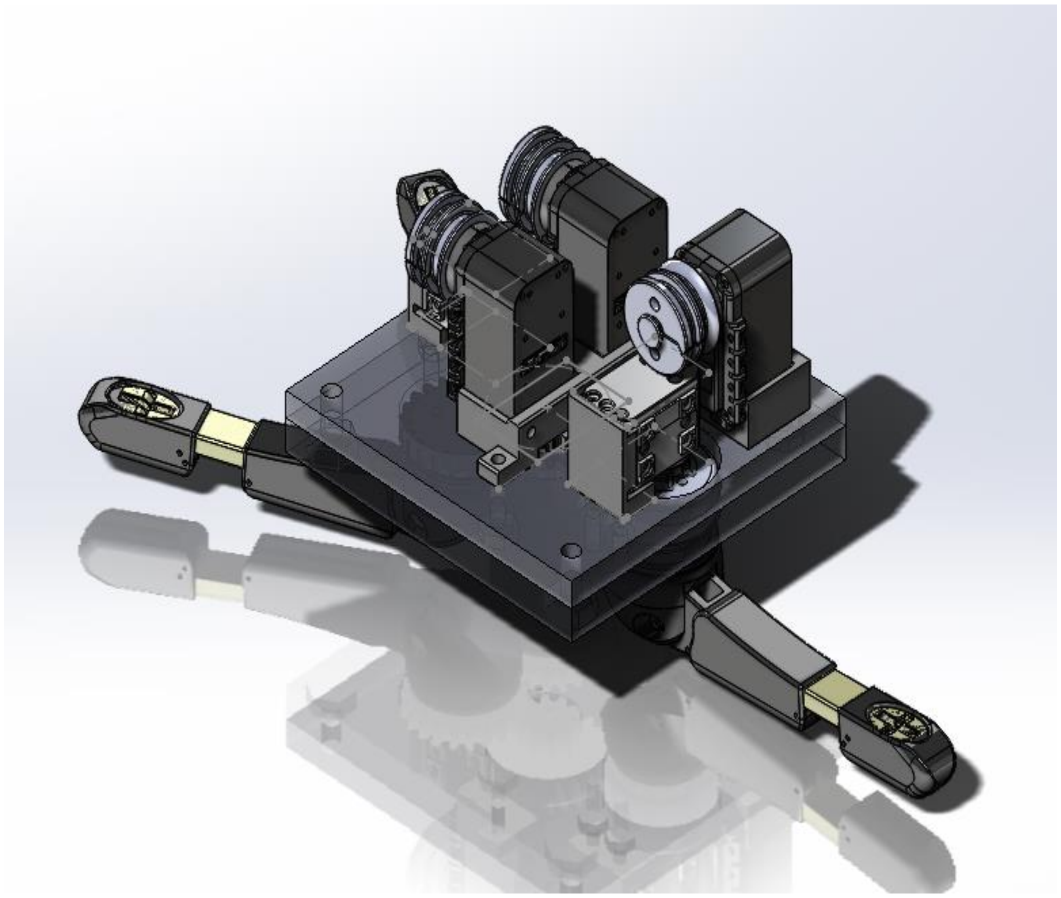


Figure 4-12 Iteration 1 Model

In the discussion of modification for prototype 0th, we mentioned that we needed to find a way to directly connect the driven gear on top base with an intermediate gear on bottom base and make the intermediate gear drive the gears on fingers. We planned to 3D print the connecting shaft between the driven gear and the intermediate gear but we found that the 3D printer could not print the shaft due to its small dimension. Therefore, we designed a rigid two stage gear, larger on top base and smaller on bottom base, and used it as the intermediate gear to drive gears on fingers. We needed to hold and fix the two stage gear firmly in its position so we modified the mount for Dynamixel AX-12 of fingers to hold and fix position of two stage gear. We combined the separate mounts into one fixture and used a connecting shaft to hold the two stage gear. The problem of reduction ratio and vertical string tendon was also fixed.

4.3.2. <u>Results and observations</u>

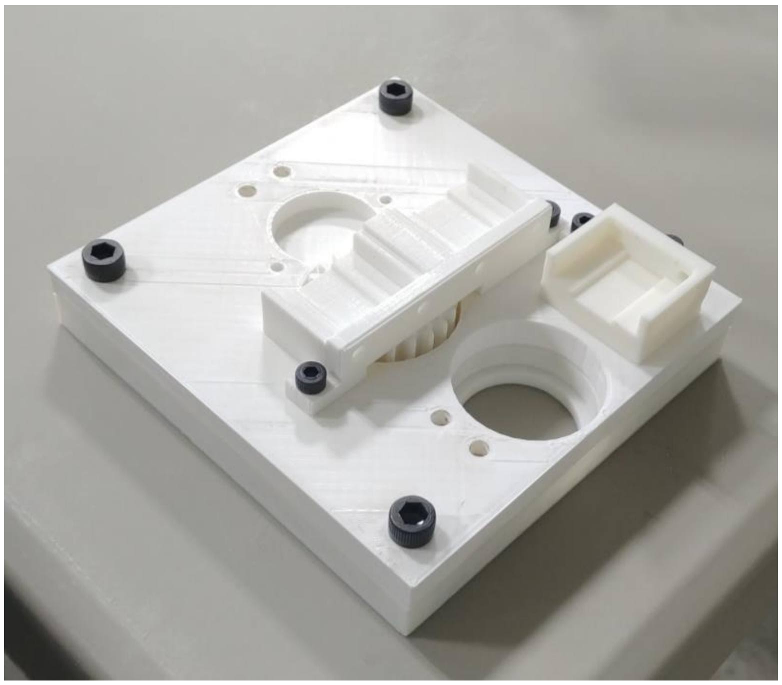


Figure 4-13 Iteration 1 Assembly

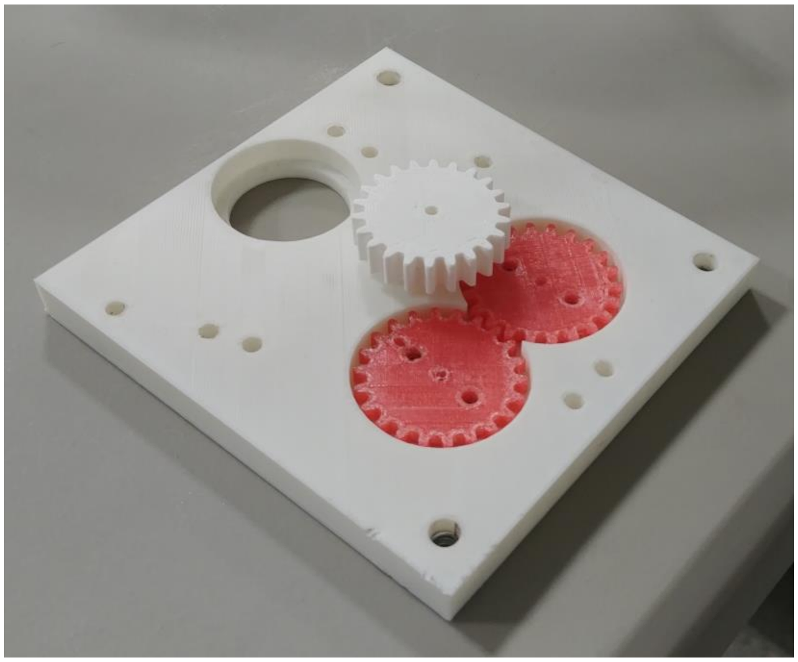


Figure 4-14 Iteration 1 Gear Train Assembly

The assembly does not have electrical parts and fingers to show its mechanical aspects. In this iteration, a noticeable difference was the closer distance between two forefingers due to reduction in number of gears. The closer distance lead to our first problem which was the interference of thumb finger and forefingers in grasping motion. There was also interference between the bases of forefingers which lead to friction between the bases and affected overall performance of the hand. When assembling, we found that the pulley on Dynamixel AX-12 motor in thumb position interfered with Dynamixel XL-320 motor causing friction between the parts and leading to rough adduction or abduction motion in thumb finger. We also noticed that the problem of torque transmission between gear train was not solved. There was space among gears leading to rough motions of fingers. Under a configuration of driven gear and two stage gear where the gears were not perfectly meshed, the gear train blocked finger motions instead of driving them. The following link is a video demonstrates how iteration 1 works with GUI controller and CaptoGlove: Video link

4.3.3. *Modifications for next iteration*

From the results, we firstly decided to increase the gap between two forefingers and run the Python script discussed in section 4.2.3 to get new dimensions for gear diameters at the cost of reducing torque transmission and size of hand base. We applied the distance between two forefingers of Reflex One prototype to our next iteration and used this as one constraint in the program. By increasing the gap between two forefingers, the mounting fixture would be re-designed to be two motor mounts separate from each other. The height of thumb mount would be increased to account for the interference of the pulley on Dynamixel AX-12 motor and Dynamixel XL-320 motor in thumb position. In the next iteration, we also needed to incorporate the PCB board onto the top base and re-design the hand base so the base would be aesthetically appealing and mechanically functioned. We will discuss the details of iteration 2 in the next section.

4.4. Iteration 2

4.4.1. *Objectives and description*

Our main objective for iteration 2 is to address all the problems mentioned in iteration 1 and move on finalize the design for robot hand. The model of iteration 2 is shown in figure 4.15.

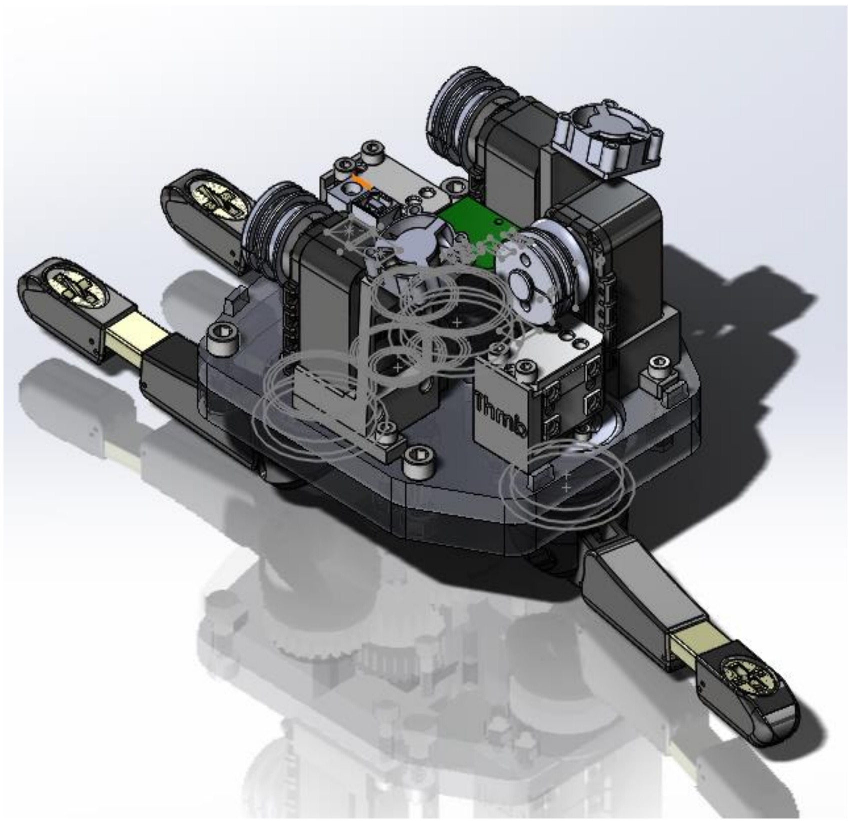


Figure 4-15 Iteration 2 Model

From figure 4.15, the first improvement compared to iteration 1 is that iteration 2 is aesthetically better with the round edges of hand base. The gears increased from 4 to 5 thus increasing the width of the base by 10 mm to satisfy new configuration of gears for reduction ratio of 0.5. In this iteration, the gear holes were smaller and gears were fixed in place by stainless steel rigid and hollow shafts. We used rigid shafts to prevent gears from shaking inside the holes. We also used hollow shafts to fix the gears on fingers because the string tendon needed to pass through the center holes on finger gears. When making the design for iteration 2, we encountered a problem of awkward design of the PCB board. Initially, the PCB board, which we received from RightHand Robotics, Inc., was designed for Reflex One. However, when we tried to incorporate the PCB board into our design, the board was not fit in the place that we intended for PCB board mounting. Therefore, we needed to design a mount above top base surface to prevent the PCB board from contacting other parts while maintaining the ease of connecting cables from motors to PCB board.

4.4.2. Results and observations

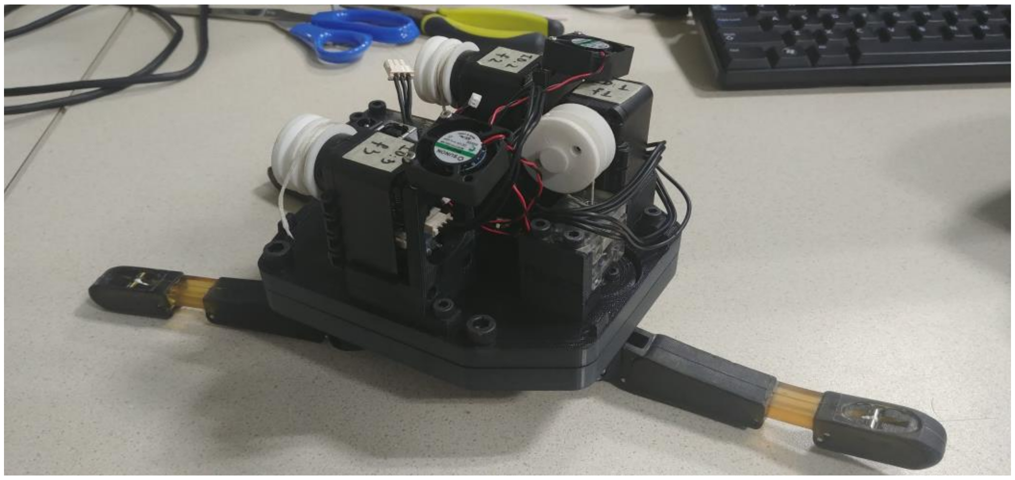


Figure 4-16 Iteration 2 Assembly

After assembly, we found that the shaking movement of gears stopped and torque transmission was better. We determined that on mechanical aspect, the design was good enough to finalize with other add-on parts such as cooling fans, hand housing, and mounting adapter to Baxter Robot arm, if necessary.

4.4.3. Modifications to finalize the design

From the results, we planned to modify the motor mounts to incorporate cooling fans. After that, we would move on to design the housing and mounting adapter. We will discuss the design process in the next section.

4.5. Finalization

4.5.1. Housing

On one hand, we designed the housing to be rigid enough to prevent the impact from surrounding environment. One the other hand, the housing design also supported cooling the motors inside with air ventilation system. The design of housing is shown in figure 4.17.

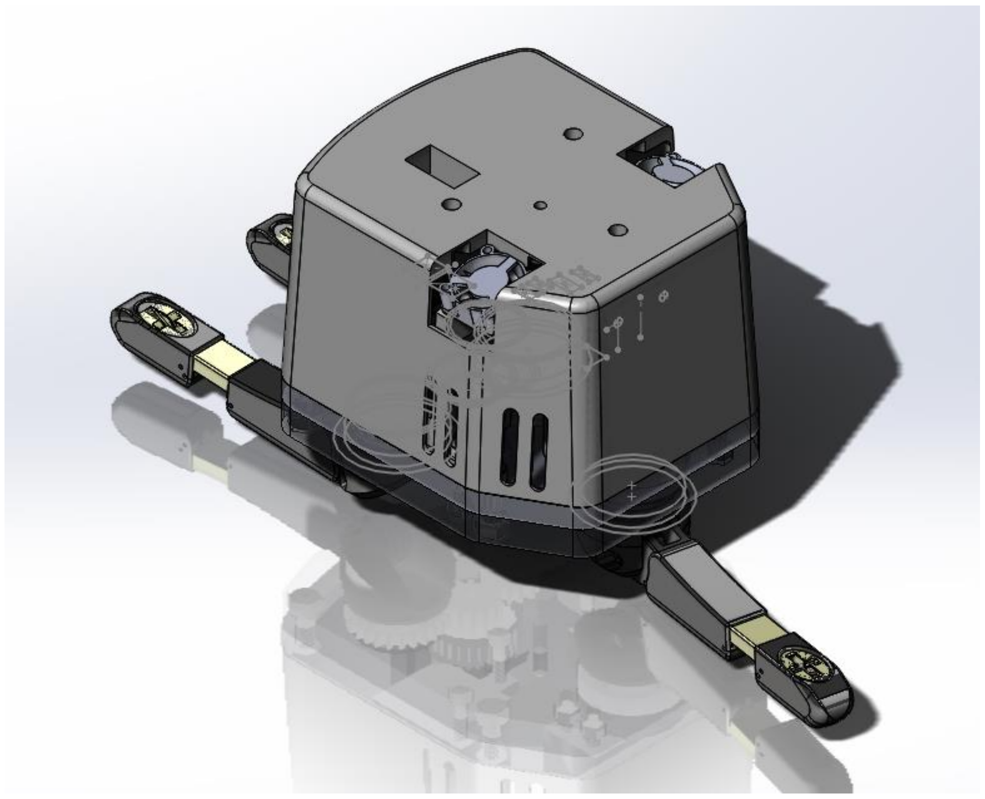


Figure 4-17 Housing design in Solidworks

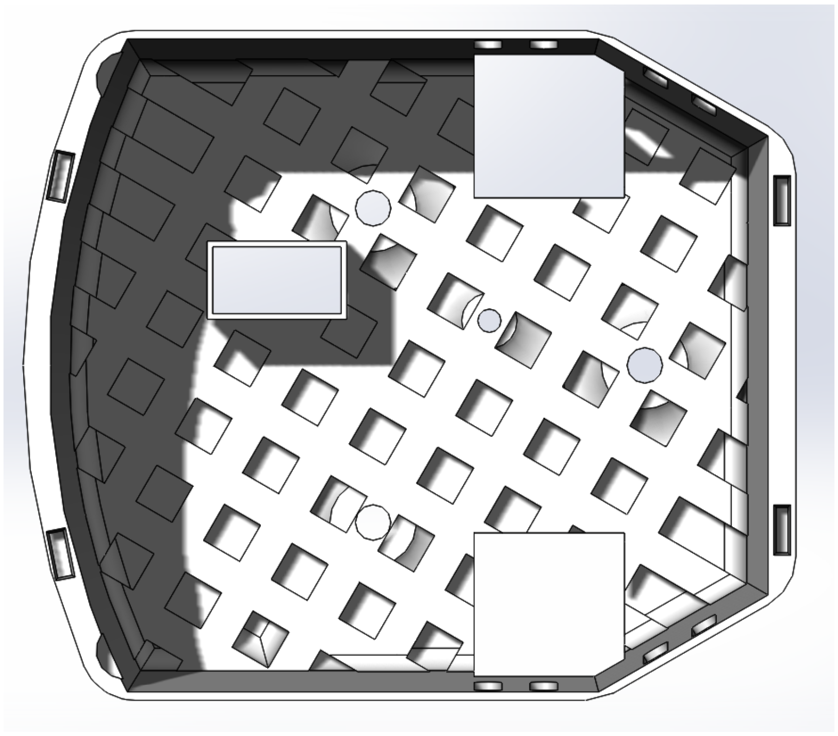


Figure 4-18 Stress-wide-distribution mechanism inside housing

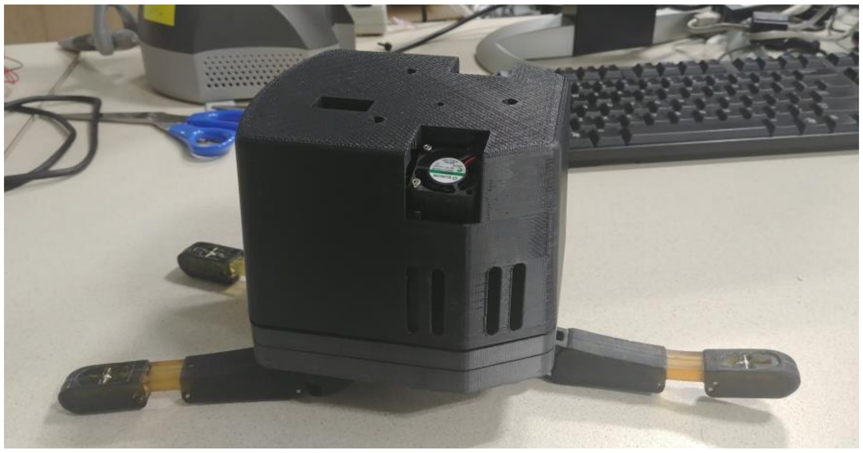


Figure 4-19 Housing design assembly

The housing maximum thickness is 5 millimeters and minimum thickness is 1.5 millimeters. On the top of housing, we made a crossing-bar design to distribute the stress uniformly and strengthen the material near the holes that connect the hand to Baxter Robot arm adapter. In addition, there are two large square holes on the top and slots near the bottom of housing for cooling fans to support air ventilation and cooling process of motors.

4.5.2. Mounting adapter to Baxter Robot

Our team considered of incorporating the hand to Baxter Robot arm by modifying the concurrent mounting adapter of the Baxter Robot. However, due to time limitation, we are not able to finish the mounting of the hand to the Baxter Robot.

5. Controlling software improvement

5.1. GUI Controller

5.1.1. Motivation

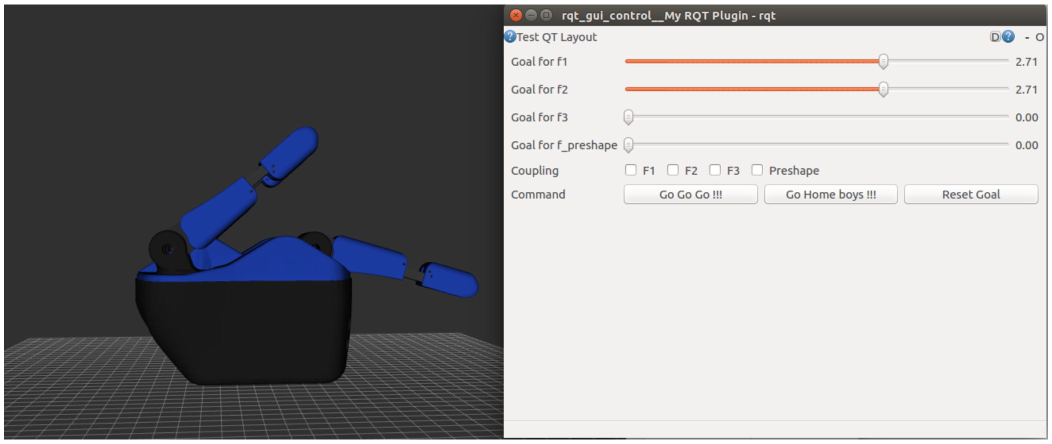
With the current setup of the TRINA system right now, users can only do open and close command to the ReflexSF hands. Users cannot do more in-hand manipulation or specifically control each finger or preshape easily. Because the users for the system ideally will be nurses, therefore they do not have any prior experience on programming and ROS. They will find it very difficult to learn and operate the hand through programming and linux command. Therefore, we want to create a user friendly GUI to control simple motion of the Reflex hands. Not only that, we want to create a GUI that can control variety of different robot hands easily, for example the Reflex Hand and Our implemented Hand on the same GUI. We also want it to be as easy as possible to add another hand to control.

5.1.2. Design

We want to take advantage of all the possible commands we have for the Reflex Hand and implemented them also for our new robot hand. There are three type of control commands we can use for ReflexSF, position control, velocity control and position and velocity control. We also want to control each motor with the GUI therefore if we put all of them in the same interface, it would be difficult for the users to know what is going on. Therefore, in our design we want to make GUI as simple as possible and all the different control and element should be separated. Therefore, we used the Plug-In from Rqt to open multiple windows (or widgets) at the same time while we can move them around, attach them to a dock or display all of them at the same time depend on user preferences.

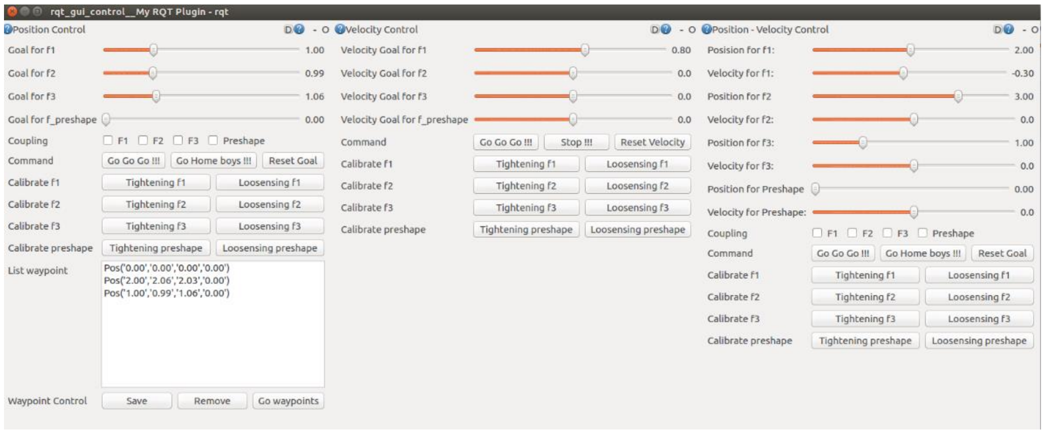
In more details, Rqt is a Qt-based framework for GUI development for ROS. Rqt is a software framework of ROS that implements the various GUI tools in the form of plugins. One can run all the existing GUI tools as dock-able windows within rqt. The tools can still run in a traditional standalone method, but rqt makes it easier to manage all the various windows on the screen at one moment. Main purpose of rqt is rqt is a Qt-based framework for GUI development for ROS. Therefore, inside each widget, we can use ROS without any difficulty.

The design in the past can be seen in the next few figure.



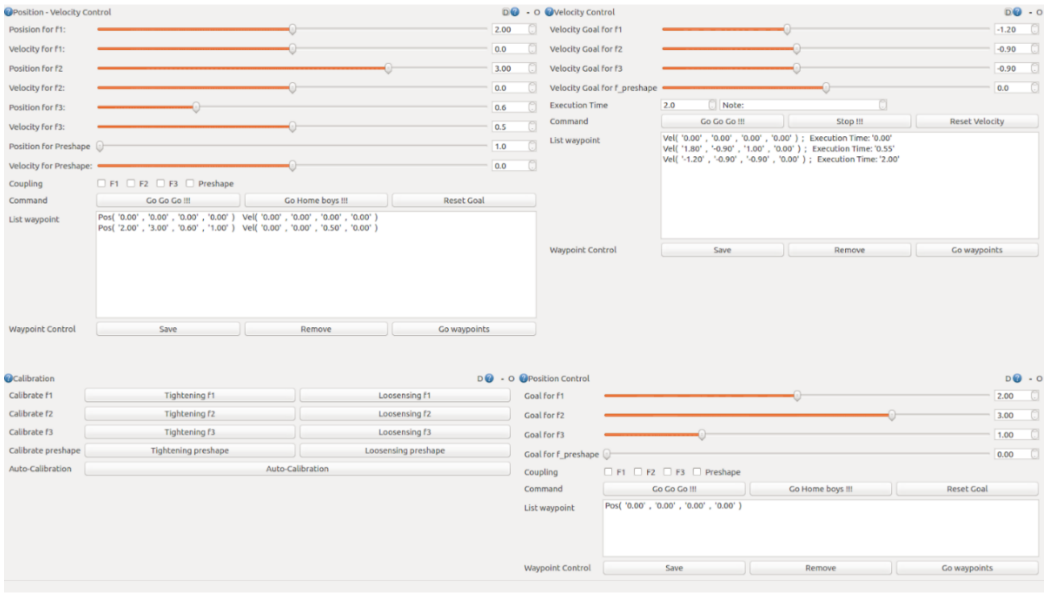
Goal for f1 2.71
Goal for f2 2.71
Goal for f3 0.00
Goal for f_preshape 0.00
Coupling ☐ F1 ☐ F2 ☐ F3 ☐ Preshape

Figure 5-1 Initial Design and test with simulation.



```
Pos('0.00','0.00','0.00','0.00')
Pos('2.00','2.06','2.03','0.00')
Pos('1.00','0.99','1.06','0.00')
```

Figure 5-2 Design iteration 2



```
Pos( '0.00' , '0.00' , '0.00' , '0.00' )  Vel( '0.00' , '0.00' , '0.00' , '0.00' )
Pos( '2.00' , '3.00' , '0.60' , '1.00' )  Vel( '0.00' , '0.00' , '0.50' , '0.00' )
```

```
Vel( '0.00' , '0.00' , '0.00' , '0.00' ) ;  Execution Time: '0.00'
Vel( '1.80' , '-0.90' , '1.00' , '0.00' ) ;  Execution Time: '0.55'
Vel( '-1.20' , '-0.90' , '-0.90' , '0.00' ) ;  Execution Time: '2.00'
```

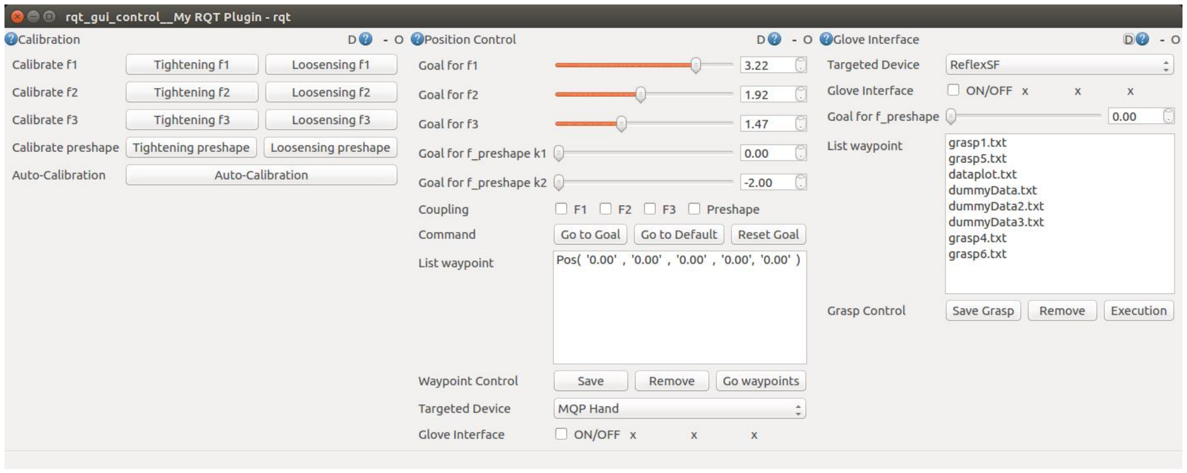
```
Pos( '0.00' , '0.00' , '0.00' , '0.00' )
```

Figure 5-3 Design iteration 2

5.1.3. Implementation

We started with the rqt plugin tutorial from ROS website and continue to build based on that. For plugin function to work we first need a python file that is the main controller window which jobs is to initialize all the different widgets we have. Then each widgets python file will be displaying all the label and button and slider we have for controlling the hand.

Our final design of the GUI is shown in Figure below.



Calibrate f1
Calibrate f2
Calibrate f3
Calibrate preshape
Auto-Calibration

Goal for f1: 3.22
Goal for f2: 1.92
Goal for f3: 1.47
Goal for f_preshape k1: 0.00
Goal for f_preshape k2: -2.00
Coupling: ☐ F1 ☐ F2 ☐ F3 ☐ Preshape
Command
List waypoint: Pos('0.00' , '0.00' , '0.00' , '0.00', '0.00')
Waypoint Control
Targeted Device: MQP Hand
Glove Interface: ☐ ON/OFF x x x

Targeted Device: ReflexSF
Glove Interface: ☐ ON/OFF x x x
Goal for f_preshape: 0.00
List waypoint: grasp1.txt, grasp5.txt, dataplot.txt, dummyData.txt, dummyData2.txt, dummyData3.txt, grasp4.txt, grasp6.txt
Grasp Control

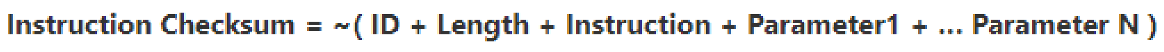
Figure 5-4 Final Iteration

In our final design iteration of the GUI, we have three special GUI. The first one is calibration tab which button click correspond to send service to the reflex node to run our own function for calibrating finger, instead of calibrating finger manually in the command line, which took a lot of them to go through each motor. Second tab is position control tab, which we can save series of pose of the hand, and then save all of them into a txt file and save them into a grasp, which data can be filtered and then play back to the hand using position control. We also added a grasp GUI, which users can add grasp play grasp using pre-existing grasp txt. In both position control tab and glove interface tab, we added the targeted device section and glove section so that the user can choose what device it wants to use the GUI for and had option of using the data glove or not.

5.2. Improving control package

5.2.1. Dynamixel Protocol 2.0

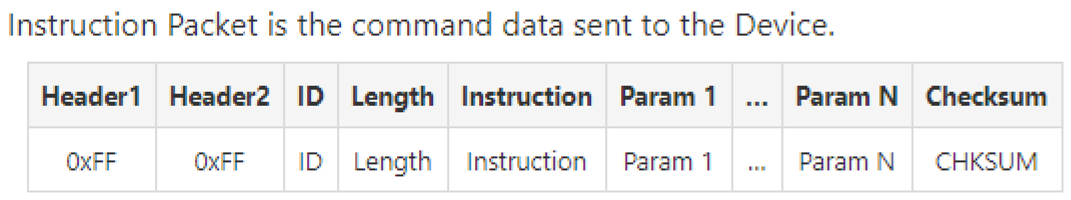
The original Dynamixel servos came along with the Protocol 1.0 for TTL communication between the servos. This protocol provides a simple package format that is divided into two packets. The instruction packet is the command data sent to the servo, usually from a higher level controller. This will get processed by the servo itself and responded via a status packet after a set period of time that can be configured by accessing the EEPROM table of the Dynamixel motor. Both packets consisted of a 2-bit header, following by a 1-bit motor ID, a 1-bit data length following by the data parameter itself, then ends with a 1-bit checksum. The checksum algorithm is pretty simple as well.



Instruction Checksum = ~(ID + Length + Instruction + Parameter1 + ... Parameter N)

Figure 5-5 Checksum for Protocol 1.0

Now there are of course some difference between the instruction packet sent to the servo and the returned status packet.

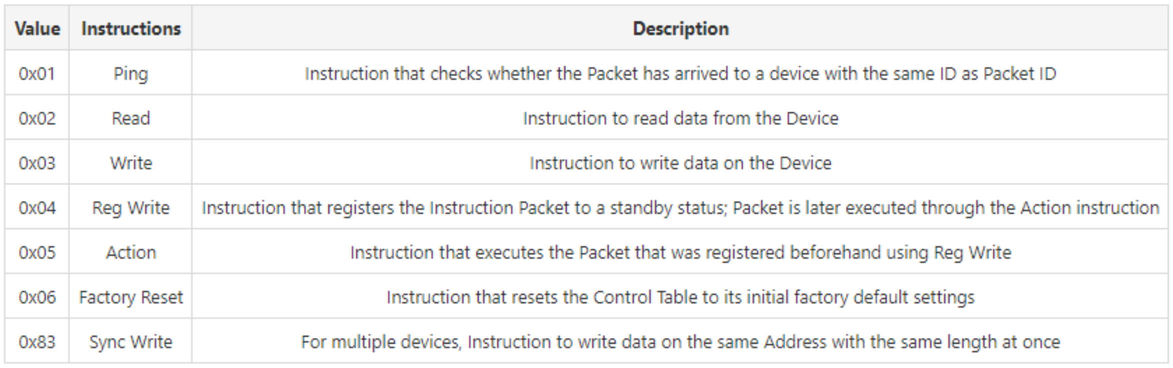


Instruction Packet is the command data sent to the Device.

Header1	Header2	ID	Length	Instruction	Param 1	...	Param N	Checksum
0xFF	0xFF	ID	Length	Instruction	Param 1	...	Param N	CHKSUM

Figure 5-6 Instruction packet format

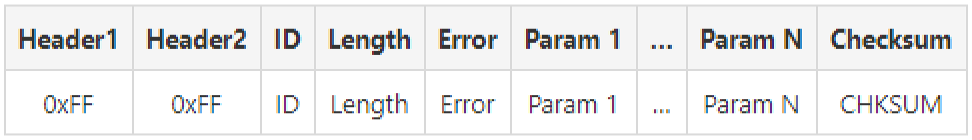
Notice here the instruction field will contain whatever instruction that the packet wishes the servo to do. Considering the early implementation of the communication protocol, there are actually a limited number of instructions one can do with the servo. Following the instruction bit will be the data parameters required for the said instruction.



Value	Instructions	Description
0x01	Ping	Instruction that checks whether the Packet has arrived to a device with the same ID as Packet ID
0x02	Read	Instruction to read data from the Device
0x03	Write	Instruction to write data on the Device
0x04	Reg Write	Instruction that registers the Instruction Packet to a standby status; Packet is later executed through the Action instruction
0x05	Action	Instruction that executes the Packet that was registered beforehand using Reg Write
0x06	Factory Reset	Instruction that resets the Control Table to its initial factory default settings
0x83	Sync Write	For multiple devices, Instruction to write data on the same Address with the same length at once

Figure 5-7 Protocol 1.0 instruction table

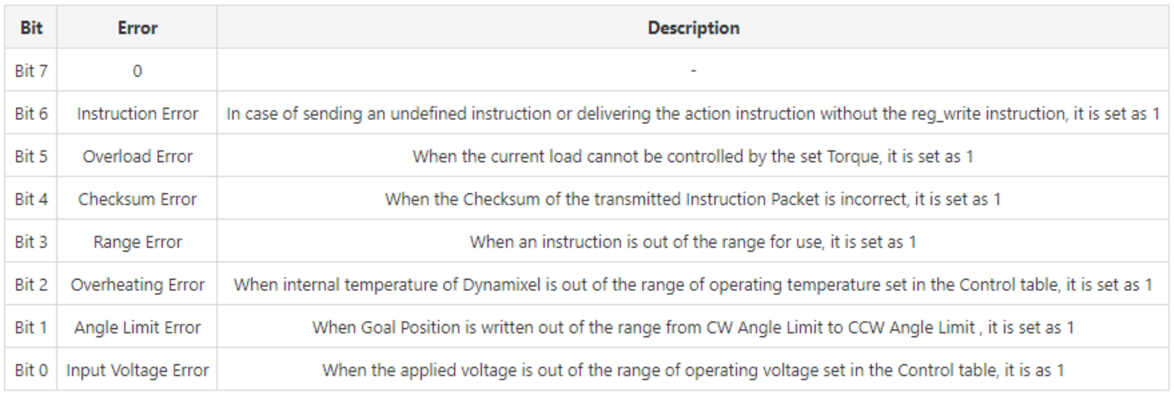
Now these are some basic instructions for the Dynamixel servos, and the descriptions themselves are self-explanatory. One instruction worth noticing here is the sync write instruction. This would only work for the same model of Dynamixel servos communicated, as different models will have different address values. This will be discussed further in the following chapter where the controller software has to handle both the AX-12A and the XL-320. Moving on to the status packet, the format is somewhat the same.



Header1	Header2	ID	Length	Error	Param 1	...	Param N	Checksum
0xFF	0xFF	ID	Length	Error	Param 1	...	Param N	CHKSUM

Figure 5-8 Protocol 1.0 Status packet format

The status packet instruction field is replaced by the error field, then followed by the data parameters requested by the instruction packet then ends with a checksum, with the same checksum algorithm where the instruction value will be replaced with the error value of course.



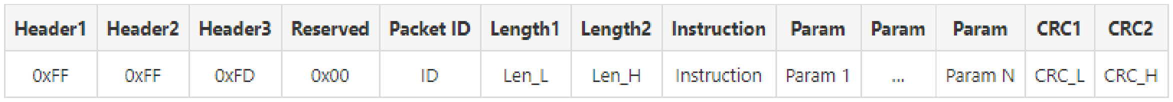
Bit	Error	Description
Bit 7	0	-
Bit 6	Instruction Error	In case of sending an undefined instruction or delivering the action instruction without the reg_write instruction, it is set as 1
Bit 5	Overload Error	When the current load cannot be controlled by the set Torque, it is set as 1
Bit 4	Checksum Error	When the Checksum of the transmitted Instruction Packet is incorrect, it is set as 1
Bit 3	Range Error	When an instruction is out of the range for use, it is set as 1
Bit 2	Overheating Error	When internal temperature of Dynamixel is out of the range of operating temperature set in the Control table, it is set as 1
Bit 1	Angle Limit Error	When Goal Position is written out of the range from CW Angle Limit to CCW Angle Limit , it is set as 1
Bit 0	Input Voltage Error	When the applied voltage is out of the range of operating voltage set in the Control table, it is as 1

Figure 5-9 Error bit field for Protocol 1.0

The description behind every error value is self-explanatory. This status packet is very helpful for the controller as a separated error instruction does not have to be sent every time to check for error. The packet not only reply with the necessary information requested, but also provide a report of its current status as well, then the higher level controller can take actions accordingly, assuring the system running without errors. This is especially useful for a system of multiple Dynamixel servos daisy chained together, as the response packet has a servo ID to identify what motor is having issue.

Along with the release of later lines of products, Dynamixel also introduced the Protocol 2.0 for communication across servos. This later protocol is supposed to be better than the previous, as well as providing more functionalities as well. However, this new Protocol is not compatible with the previous Protocol, as the packet format order is significantly changed. Moreover, Dynamixel's servos has a lot of variations in term of supported communication protocol: some will only support Protocol 1.0 (the Dynamixel AX-12A for example), while some will only support Protocol 2.0 (the Dynamixel XL-320 for example) then there are some that can support both versions (the Dynamixel MX-28 with firmware upgraded for example). This provides a complication for a system with different Dynamixel servos models, something that will be discussed more in the controller software implementation section.

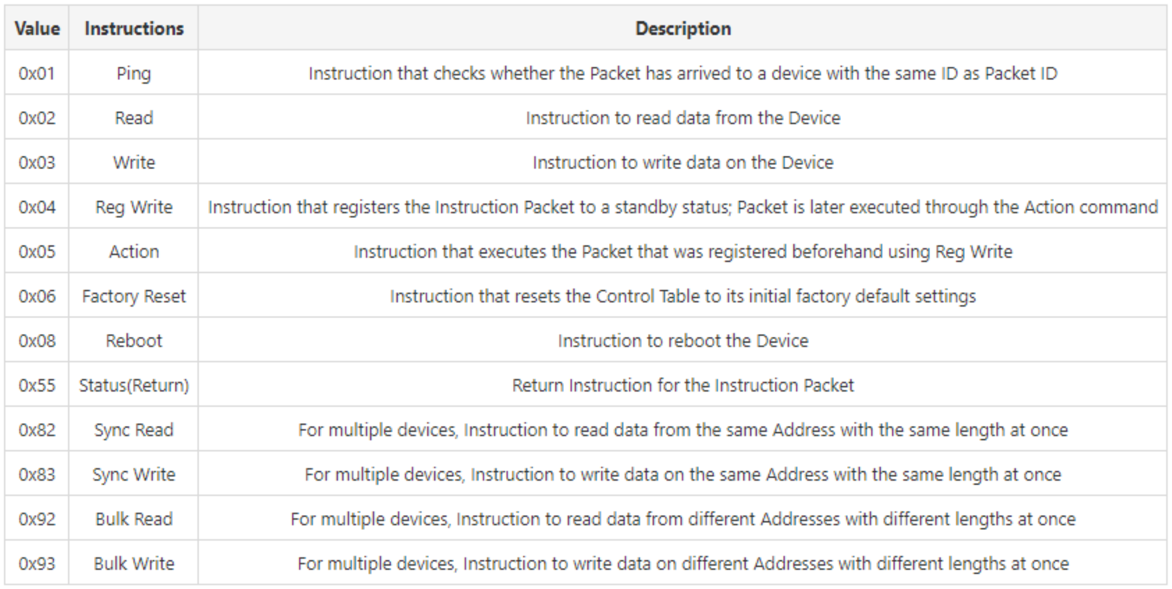
Now the first difference between Protocol 1.0 and Protocol 2.0 from Dynamixel is the packet layout, both on the status and instruction: the header contains 3-bit of info, then followed by a 1-bit reserved, the 1-bit packet ID, the length is changed to 2-bit, 1-bit instruction follow by the data parameters, and lastly ends with a 2-bit CRC.



Header1	Header2	Header3	Reserved	Packet ID	Length1	Length2	Instruction	Param	Param	Param	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	ID	Len_L	Len_H	Instruction	Param 1	...	Param N	CRC_L	CRC_H

Figure 5-10 Protocol 2.0 instruction packet layout

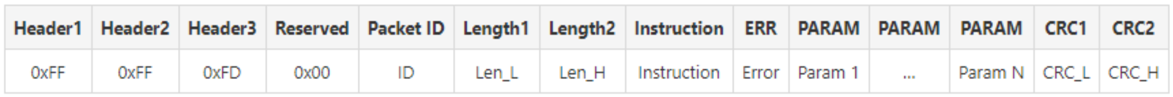
The 2-bit checksum also means a change in the checksum calculation protocol: instead of using the fixed bitwise complement like in Protocol 1.0, Protocol 2.0 uses a more complicated checksum algorithm that requires a fixed input table for calculation that is somewhat similar to the CRC-16[10]. Along with the difference in each section's length, these are the main issues that have been causing issues with compatibility among the two communication protocols. Despite those issues, there are actually many improvements with the Protocol 2.0. The more complicated checksum algorithm means it is less likely to missed error data and damaged the whole system. The increased in length of the header will actually help with noise filtration as well, such that error packets will be dropped by either end. The increased bit size of some segments allowed more data to be transferred under one packet as well. Lastly, there are new instructions added for the Protocol 2.0 to allow more complicated commands.



Value	Instructions	Description
0x01	Ping	Instruction that checks whether the Packet has arrived to a device with the same ID as Packet ID
0x02	Read	Instruction to read data from the Device
0x03	Write	Instruction to write data on the Device
0x04	Reg Write	Instruction that registers the Instruction Packet to a standby status; Packet is later executed through the Action command
0x05	Action	Instruction that executes the Packet that was registered beforehand using Reg Write
0x06	Factory Reset	Instruction that resets the Control Table to its initial factory default settings
0x08	Reboot	Instruction to reboot the Device
0x55	Status(Return)	Return Instruction for the Instruction Packet
0x82	Sync Read	For multiple devices, Instruction to read data from the same Address with the same length at once
0x83	Sync Write	For multiple devices, Instruction to write data on the same Address with the same length at once
0x92	Bulk Read	For multiple devices, Instruction to read data from different Addresses with different lengths at once
0x93	Bulk Write	For multiple devices, Instruction to write data on different Addresses with different lengths at once

Figure 5-11 Instructions for Dynamixel's Protocol 2.0

The basic functionalities are retained the same from Protocol 1.0, and many more are added as well. The sync read and write instructions allow effective communication packet sends across multiple devices daisy chained together. Imagine having a system of five Dynamixel servos daisy chained together, a single sync read/write command can replace five individual packets send across the system, reducing the chance of error occurrence and load on the communication channel. Additionally, the bulk read and write commands are extremely helpful to gain access to multiple addresses at the same time, thus again reducing the risk of error packets and the load on the communication line. The status packet is also changed accordingly to the instruction packet, with one additional bit to reflect the instruction sent by the instruction packet. This feature will, again, help eliminate the chance of having an error packet in case it got pass the checksum algorithm.



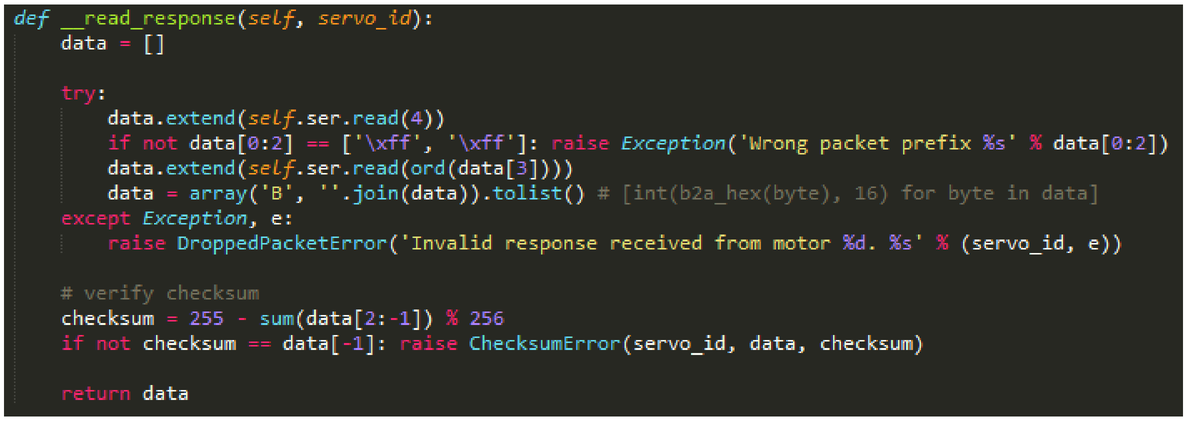
Header1	Header2	Header3	Reserved	Packet ID	Length1	Length2	Instruction	ERR	PARAM	PARAM	PARAM	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	ID	Len_L	Len_H	Instruction	Error	Param 1	...	Param N	CRC_L	CRC_H

Figure 5-12 Status packet from Dynamixel's Protocol 2.0

The bit retains the same error value table from Protocol 1.0.

5.2.2. Motor controller modification

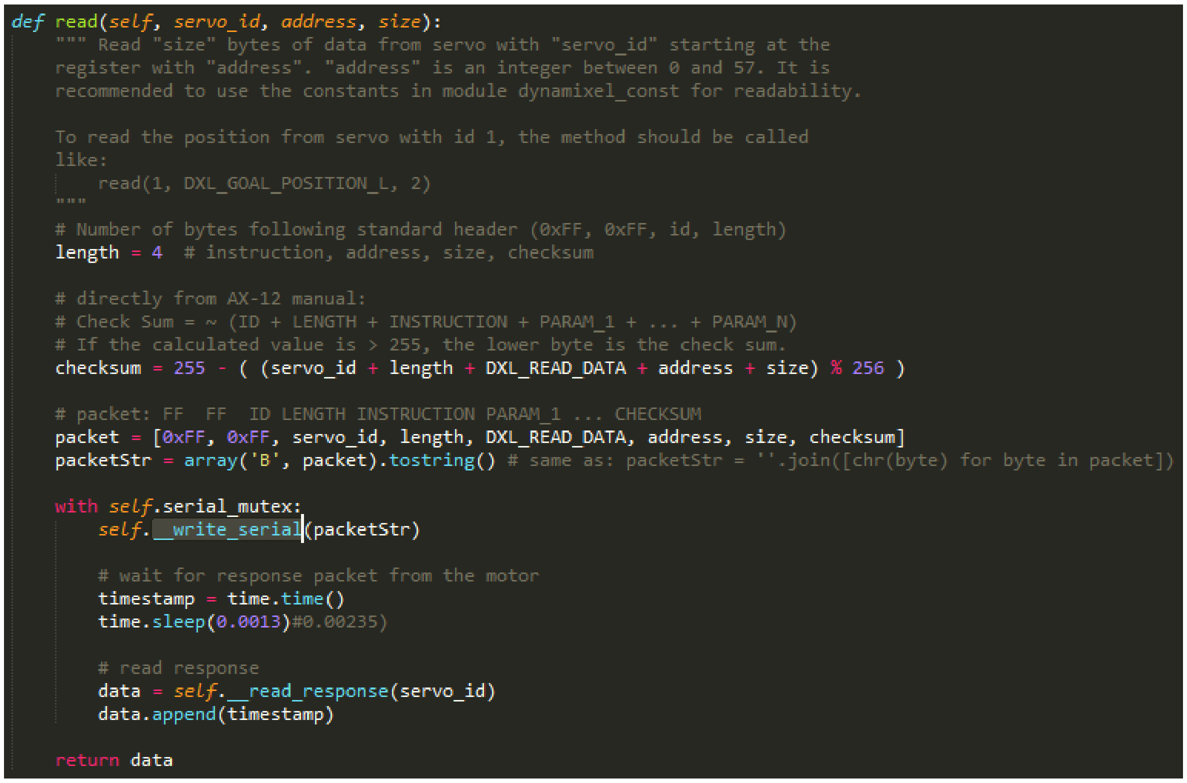
The original Dynamixel controller package developed by Antons Rebguns only supports Protocol 1.0 and was long abandoned.



```python
def __read_response(self, servo_id):
    data = []

    try:
        data.extend(self.ser.read(4))
        if not data[0:2] == ['\xff', '\xff']: raise Exception('Wrong packet prefix %s' % data[0:2])
        data.extend(self.ser.read(ord(data[3])))
        data = array('B', ''.join(data)).tolist() # [int(b2a_hex(byte), 16) for byte in data]
    except Exception, e:
        raise DroppedPacketError('Invalid response received from motor %d. %s' % (servo_id, e))

    # verify checksum
    checksum = 255 - sum(data[2:-1]) % 256
    if not checksum == data[-1]: raise ChecksumError(servo_id, data, checksum)

    return data
```

Figure 5-13 Read response function within the Dynamixel controller package

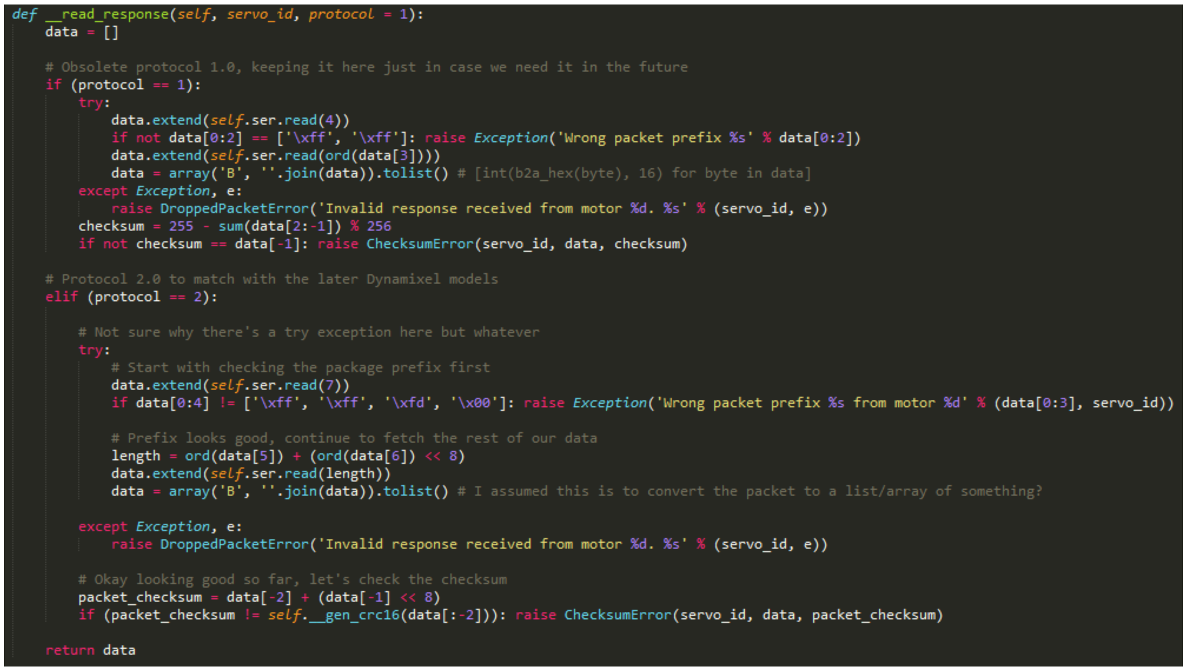
From the code section above, the read response function will search for a header value of 0xFF 0xFF from Protocol 1.0, or an exception is raised and the income packet will be dropped. This function implementation will actually find the header position, but then the rest of the data will be confused, and then a checksum error is raised by the end as the checksum algorithm is wrong as well. Now this read response function is the heart of the whole controller itself, as every read and write command to the motor will required an instruction packet sent to the motor, then wait for the status packet is responded and read that packet to verify everything is working smoothly. The command to send out the instruction packet is actually embedded within the read and write functions itself so it can be fixed separately.



```
def read(self, servo_id, address, size):
    """ Read "size" bytes of data from servo with "servo_id" starting at the
    register with "address". "address" is an integer between 0 and 57. It is
    recommended to use the constants in module dynamixel_const for readability.

    To read the position from servo with id 1, the method should be called
    like:
        read(1, DXL_GOAL_POSITION_L, 2)
    """
    # Number of bytes following standard header (0xFF, 0xFF, id, length)
    length = 4  # instruction, address, size, checksum

    # directly from AX-12 manual:
    # Check Sum = ~ (ID + LENGTH + INSTRUCTION + PARAM_1 + ... + PARAM_N)
    # If the calculated value is > 255, the lower byte is the check sum.
    checksum = 255 - ( (servo_id + length + DXL_READ_DATA + address + size) % 256 )

    # packet: FF  FF  ID LENGTH INSTRUCTION PARAM_1 ... CHECKSUM
    packet = [0xFF, 0xFF, servo_id, length, DXL_READ_DATA, address, size, checksum]
    packetStr = array('B', packet).tostring() # same as: packetStr = ''.join([chr(byte) for byte in packet])

    with self.serial_mutex:
        self.__write_serial(packetStr)

        # wait for response packet from the motor
        timestamp = time.time()
        time.sleep(0.0013)#0.00235)

        # read response
        data = self.__read_response(servo_id)
        data.append(timestamp)

    return data
```

Figure 5-14 Example of how the read function Is constructed

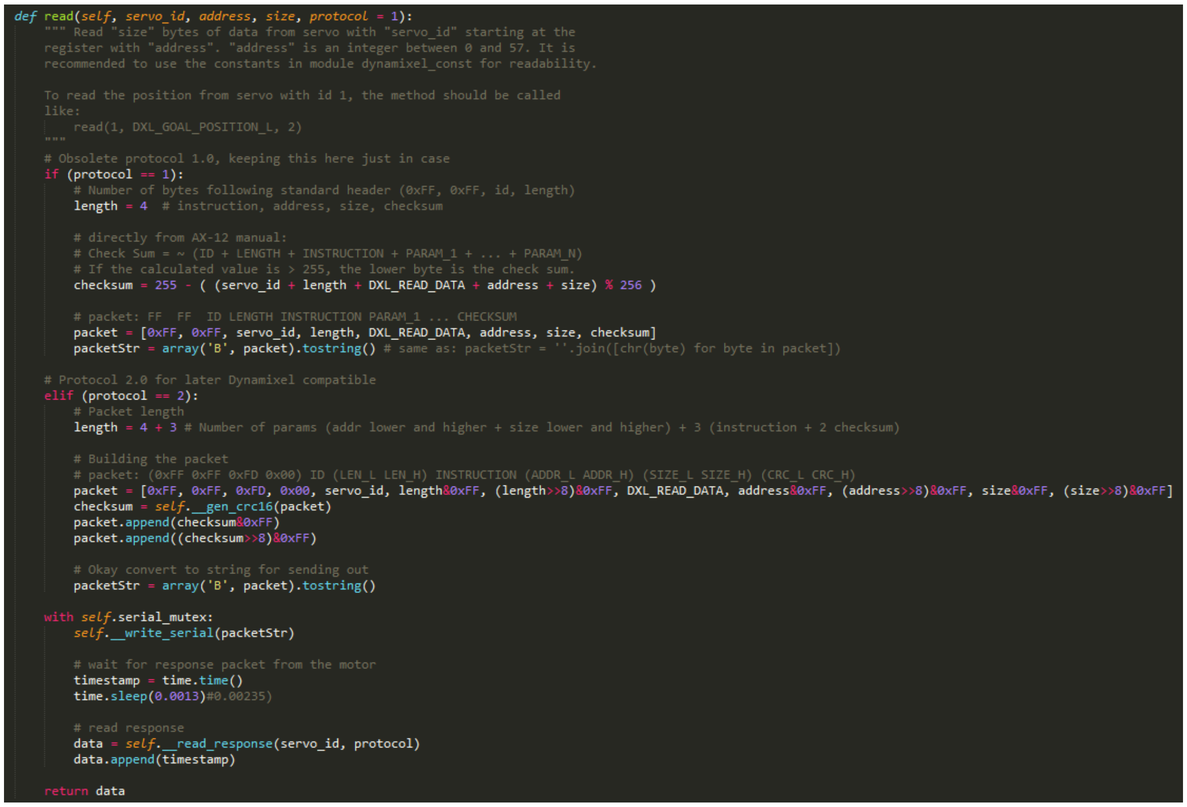
Considering the code layout, implementation of Protocol 2.0 should not be too much of an issue. First, created a similar checksum generation function based on the algorithm provided by Dynamixel to generate the checksum required for Protocol 2.0. Then, since compatibility with Protocol 1.0 is a requirement, put an additional field into the input of these functions as an indication of what protocol to use.



```
def __read_response(self, servo_id, protocol = 1):
    data = []

    # Obsolete protocol 1.0, keeping it here just in case we need it in the future
    if (protocol == 1):
        try:
            data.extend(self.ser.read(4))
            if not data[0:2] == ['\xff', '\xff']: raise Exception('Wrong packet prefix %s' % data[0:2])
            data.extend(self.ser.read(ord(data[3])))
            data = array('B', ''.join(data)).tolist() # [int(b2a_hex(byte), 16) for byte in data]
        except Exception, e:
            raise DroppedPacketError('Invalid response received from motor %d. %s' % (servo_id, e))
        checksum = 255 - sum(data[2:-1]) % 256
        if not checksum == data[-1]: raise ChecksumError(servo_id, data, checksum)

    # Protocol 2.0 to match with the later Dynamixel models
    elif (protocol == 2):

        # Not sure why there's a try exception here but whatever
        try:
            # Start with checking the package prefix first
            data.extend(self.ser.read(7))
            if data[0:4] != ['\xff', '\xff', '\xfd', '\x00']: raise Exception('Wrong packet prefix %s from motor %d' % (data[0:3], servo_id))

            # Prefix looks good, continue to fetch the rest of our data
            length = ord(data[5]) + (ord(data[6]) << 8)
            data.extend(self.ser.read(length))
            data = array('B', ''.join(data)).tolist() # I assumed this is to convert the packet to a list/array of something?

        except Exception, e:
            raise DroppedPacketError('Invalid response received from motor %d. %s' % (servo_id, e))

        # Okay looking good so far, let's check the checksum
        packet_checksum = data[-2] + (data[-1] << 8)
        if (packet_checksum != self.__gen_crc16(data[:-2])): raise ChecksumError(servo_id, data, packet_checksum)

    return data
```

Figure 5-15 Modified read response command taking both Protocol 1.0 and Protocol 2.0

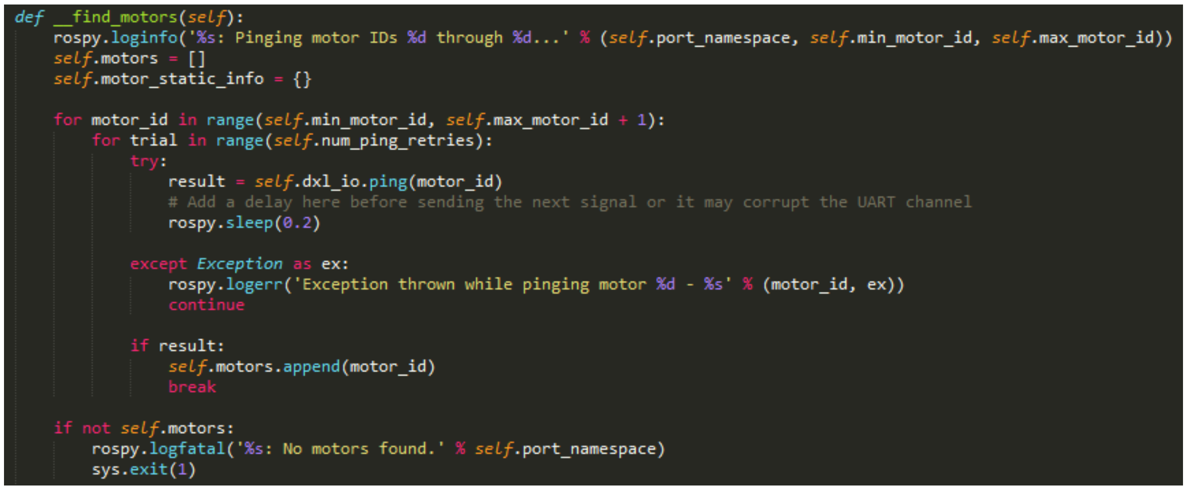
The protocol input argument works as a state machine for the read response function in this case, and is set to a default value of 1 if not specified, meaning use Protocol 1.0 by default. This protocol input is passed into from higher level function calls, such as read or write functions as shown below.



```python
def read(self, servo_id, address, size, protocol = 1):
    """ Read "size" bytes of data from servo with "servo_id" starting at the
    register with "address". "address" is an integer between 0 and 57. It is
    recommended to use the constants in module dynamixel_const for readability.

    To read the position from servo with id 1, the method should be called
    like:
        read(1, DXL_GOAL_POSITION_L, 2)
    """
    # Obsolete protocol 1.0, keeping this here just in case
    if (protocol == 1):
        # Number of bytes following standard header (0xFF, 0xFF, id, length)
        length = 4  # instruction, address, size, checksum

        # directly from AX-12 manual:
        # Check Sum = ~ (ID + LENGTH + INSTRUCTION + PARAM_1 + ... + PARAM_N)
        # If the calculated value is > 255, the lower byte is the check sum.
        checksum = 255 - ( (servo_id + length + DXL_READ_DATA + address + size) % 256 )

        # packet: FF  FF  ID LENGTH INSTRUCTION PARAM_1 ... CHECKSUM
        packet = [0xFF, 0xFF, servo_id, length, DXL_READ_DATA, address, size, checksum]
        packetStr = array('B', packet).tostring() # same as: packetStr = ''.join([chr(byte) for byte in packet])

    # Protocol 2.0 for later Dynamixel compatible
    elif (protocol == 2):
        # Packet length
        length = 4 + 3 # Number of params (addr lower and higher + size lower and higher) + 3 (instruction + 2 checksum)

        # Building the packet
        # packet: (0xFF 0xFF 0xFD 0x00) ID (LEN_L LEN_H) INSTRUCTION (ADDR_L ADDR_H) (SIZE_L SIZE_H) (CRC_L CRC_H)
        packet = [0xFF, 0xFF, 0xFD, 0x00, servo_id, length&0xFF, (length>>8)&0xFF, DXL_READ_DATA, address&0xFF, (address>>8)&0xFF, size&0xFF, (size>>8)&0xFF]
        checksum = self.__gen_crc16(packet)
        packet.append(checksum&0xFF)
        packet.append((checksum>>8)&0xFF)

        # Okay convert to string for sending out
        packetStr = array('B', packet).tostring()

    with self.serial_mutex:
        self.__write_serial(packetStr)

        # wait for response packet from the motor
        timestamp = time.time()
        time.sleep(0.0013)#0.00235)

        # read response
        data = self.__read_response(servo_id, protocol)
        data.append(timestamp)

    return data
```

Figure 5-16 Sample read function with Protocol 2.0 implemented

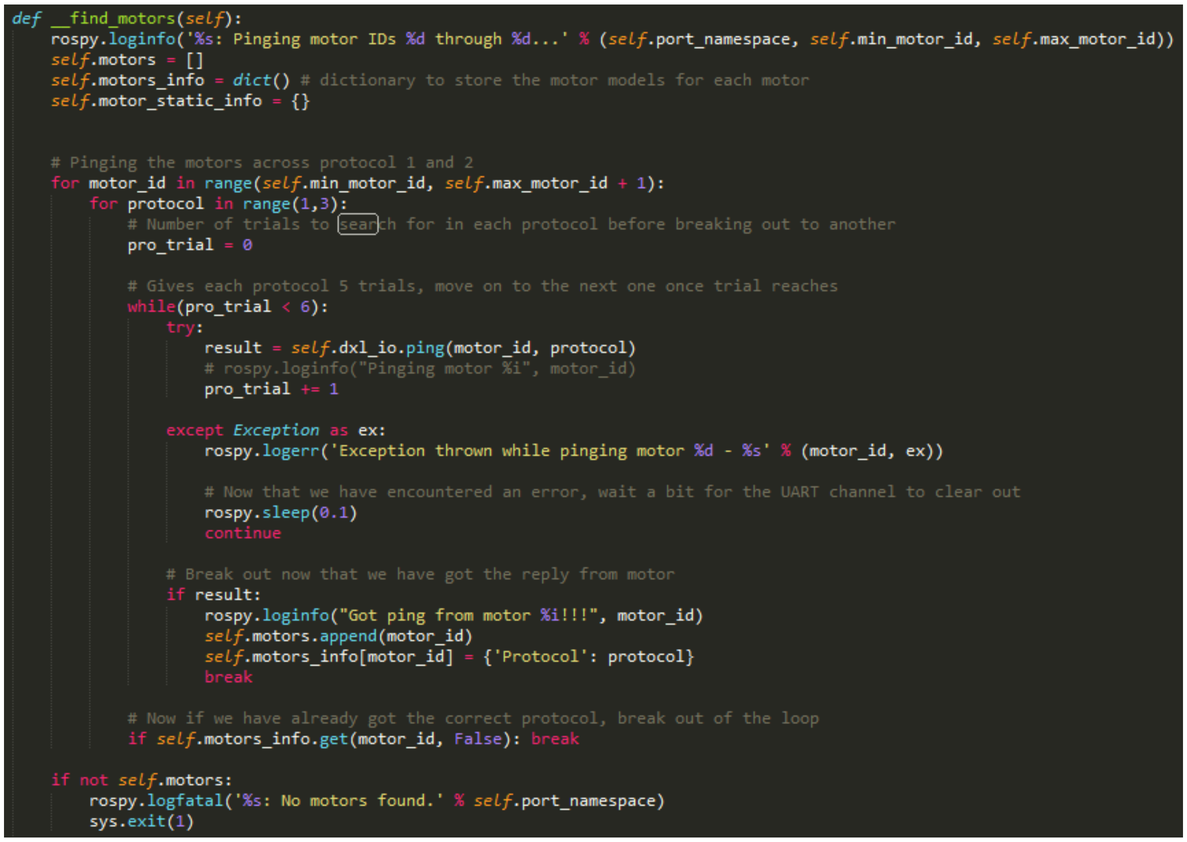
Using the same implementation of state machine, the instruction packet will be built accordingly to the chosen communication protocol. Furthermore, the read response function will be using the same protocol version to wait for the status packet from the servo. The other instruction methods provided by Dynamixel is modified with the same principle: having one more additional input argument to keep track of what communication protocol version to use, with Protocol 1.0 as a default value. With the goal of scalability, meaning this controller can be used with a wide variety of system setup of different protocol motors, and the functions must be using the correct protocol to communicate every time. This requires a modification in the motor searching function.



```python
def __find_motors(self):
    rospy.loginfo('%s: Pinging motor IDs %d through %d...' % (self.port_namespace, self.min_motor_id, self.max_motor_id))
    self.motors = []
    self.motor_static_info = {}

    for motor_id in range(self.min_motor_id, self.max_motor_id + 1):
        for trial in range(self.num_ping_retries):
            try:
                result = self.dxl_io.ping(motor_id)
                # Add a delay here before sending the next signal or it may corrupt the UART channel
                rospy.sleep(0.2)

            except Exception as ex:
                rospy.logerr('Exception thrown while pinging motor %d - %s' % (motor_id, ex))
                continue

            if result:
                self.motors.append(motor_id)
                break

    if not self.motors:
        rospy.logfatal('%s: No motors found.' % self.port_namespace)
        sys.exit(1)
```

Figure 5-17 Default motor finding function

The default servo finding function will try to ping the servos for a certain number of times until it receives the status packet back. The status packet will contain the model information of the servo. After a set of fixed trials, the controller will throw an exception if no status packet received back from the pinging servo, thus this will always happen for a Protocol 2.0 servo. A fix has to be issued in order to ensure the controller can pick up both Protocol 1.0 and Protocol 2.0 servos, then account for which servo has which ID, protocol version and model. This would require a dictionary to store the information during finding servos phase.



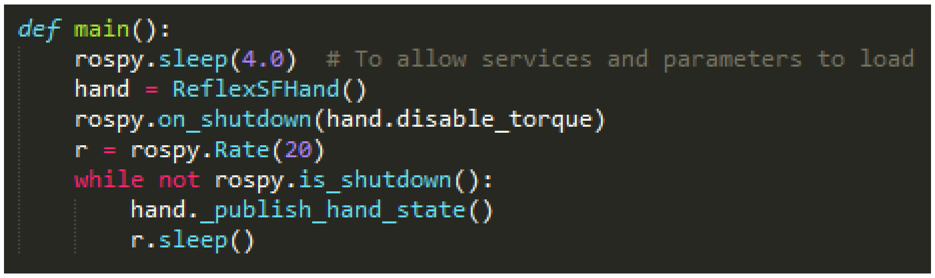
```python
def __find_motors(self):
    rospy.loginfo('%s: Pinging motor IDs %d through %d...' % (self.port_namespace, self.min_motor_id, self.max_motor_id))
    self.motors = []
    self.motors_info = dict() # dictionary to store the motor models for each motor
    self.motor_static_info = {}


    # Pinging the motors across protocol 1 and 2
    for motor_id in range(self.min_motor_id, self.max_motor_id + 1):
        for protocol in range(1,3):
            # Number of trials to search for in each protocol before breaking out to another
            pro_trial = 0

            # Gives each protocol 5 trials, move on to the next one once trial reaches
            while(pro_trial < 6):
                try:
                    result = self.dxl_io.ping(motor_id, protocol)
                    # rospy.loginfo("Pinging motor %i", motor_id)
                    pro_trial += 1

                except Exception as ex:
                    rospy.logerr('Exception thrown while pinging motor %d - %s' % (motor_id, ex))

                    # Now that we have encountered an error, wait a bit for the UART channel to clear out
                    rospy.sleep(0.1)
                    continue

                # Break out now that we have got the reply from motor
                if result:
                    rospy.loginfo("Got ping from motor %i!!!", motor_id)
                    self.motors.append(motor_id)
                    self.motors_info[motor_id] = {'Protocol': protocol}
                    break

            # Now if we have already got the correct protocol, break out of the loop
            if self.motors_info.get(motor_id, False): break

    if not self.motors:
        rospy.logfatal('%s: No motors found.' % self.port_namespace)
        sys.exit(1)
```

Figure 5-18 Flexible motor finding function

This modified version of the motor finding function will be able to pick up both communication protocols and adjust the communication method accordingly depending on which servo is currently required to communicate with.

5.2.3. Reflex package modification

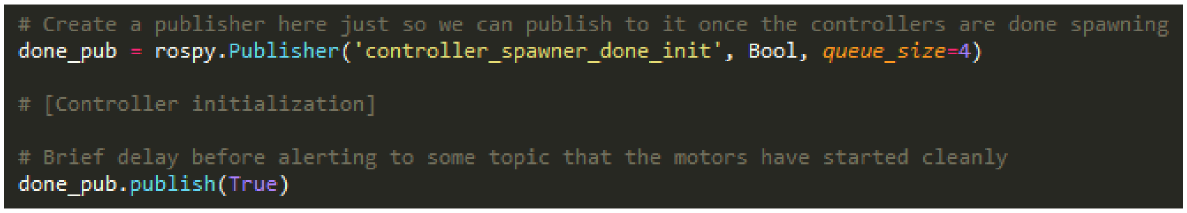
Now with the Dynamixel controller modification completed for integrating with the current system design, the Reflex package needs some modification as well.



```python
def main():
    rospy.sleep(4.0)  # To allow services and parameters to load
    hand = ReflexSFHand()
    rospy.on_shutdown(hand.disable_torque)
    r = rospy.Rate(20)
    while not rospy.is_shutdown():
        hand._publish_hand_state()
        r.sleep()
```

Figure 5-19 Reflex SF starting sequence

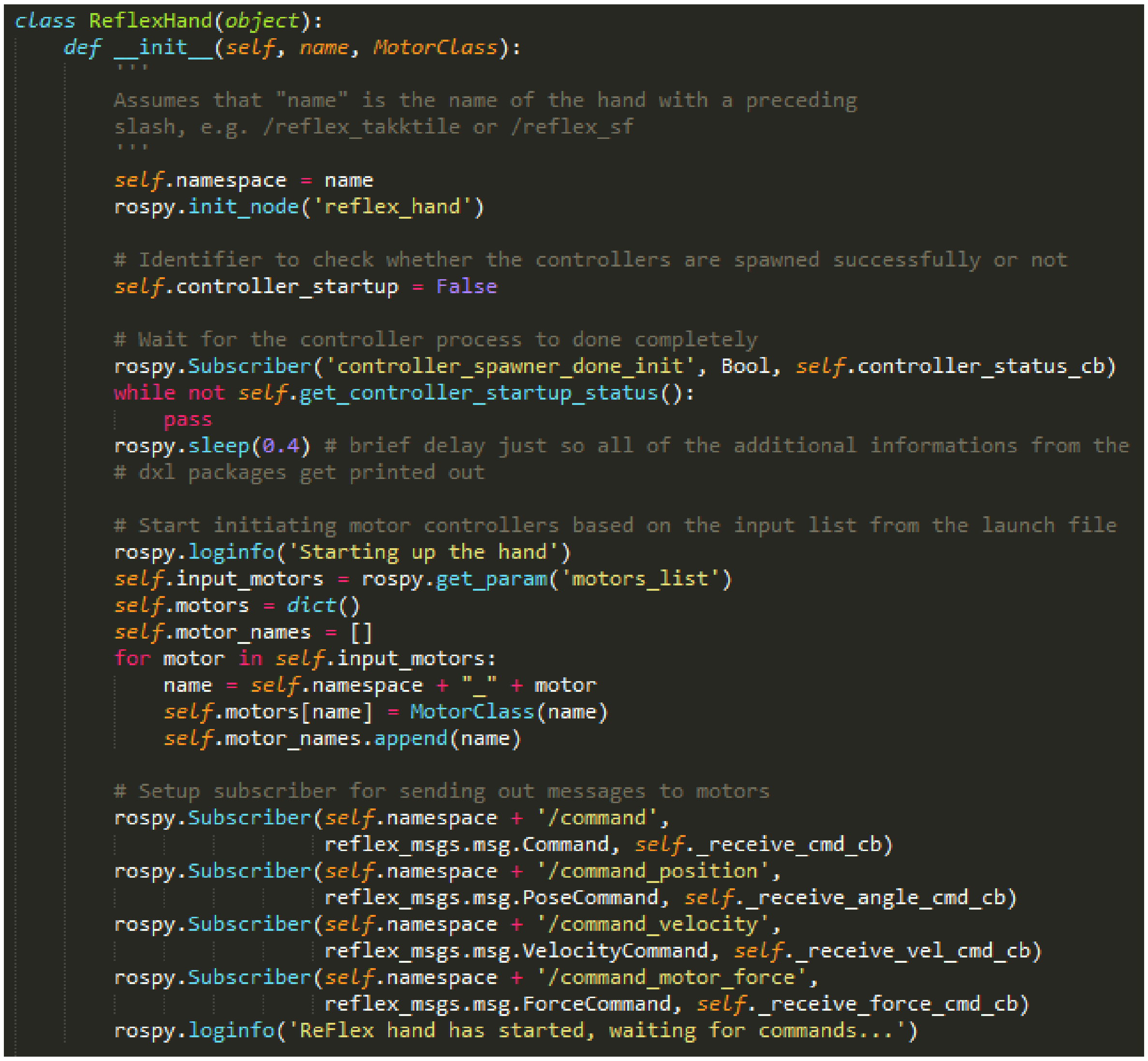
The Reflex package initialize by first calling all of the controllers for the servos, then wait for some fixed amount of time before starting. Since this package is built upon the Dynamixel controllers, the Reflex package will fail if it starts before the controllers finished initializing. In this case, it is mainly due to the controllers take more time to start up than anticipated by the Reflex package. Adding more motors, requiring more retries while finding motors, etc., can change this anticipated timer and would require a manual change in the system for the package to run normally. Moreover, the anticipation time hard coded into the Reflex package is only a trial and errors value, meaning it can be different across systems with many variables contributing to the delay time: USB speed, baud rate, etc. Therefore, a more appropriate approach is to have the Dynamixel controllers publish to a topic once it has done initializing. Then the Reflex package can subscribe to that same topic and wait for the start signal.



```python
# Create a publisher here just so we can publish to it once the controllers are done spawning
done_pub = rospy.Publisher('controller_spawner_done_init', Bool, queue_size=4)

# [Controller initialization]

# Brief delay before alerting to some topic that the motors have started cleanly
done_pub.publish(True)
```

Figure 5-20 Topic to signal initialization complete

This way, there is a dedicated topic for signaling the initialization process. And in the Reflex package, it only needs to subscribe and wait for the signal to be published before starting. Additionally, the Reflex package will only need to subscribe to this topic and wait for the done signal before launching.

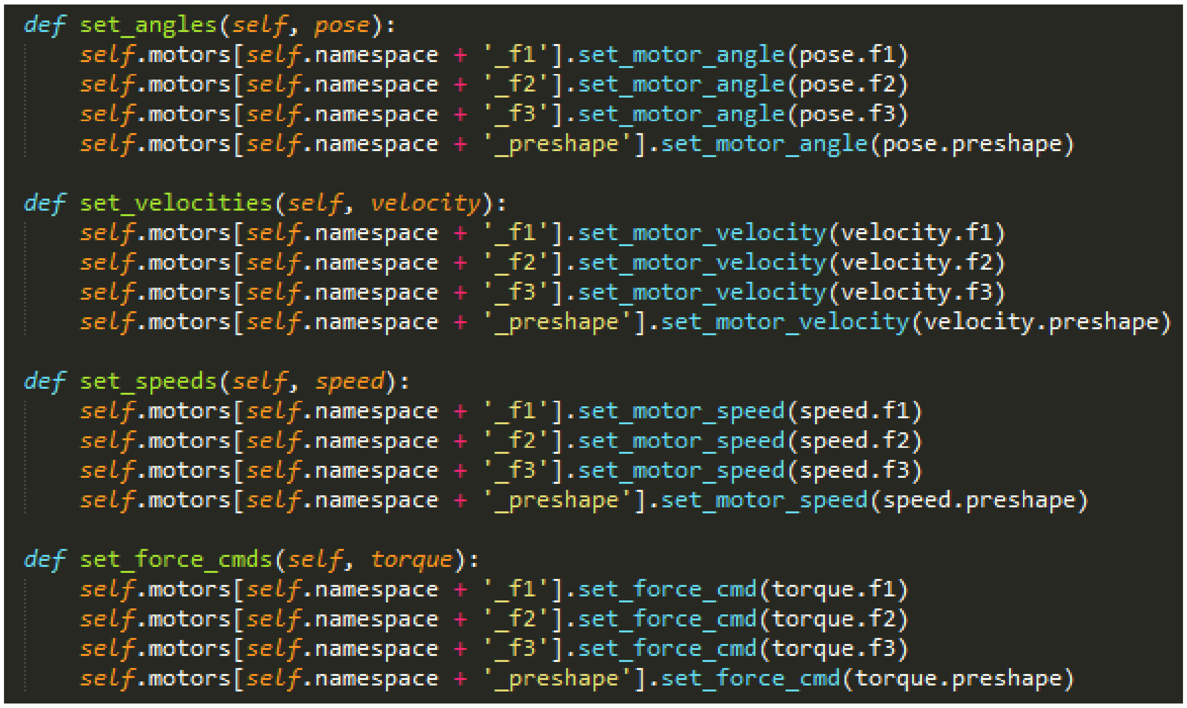


```python
class ReflexHand(object):
    def __init__(self, name, MotorClass):
        '''
        Assumes that "name" is the name of the hand with a preceding
        slash, e.g. /reflex_takktile or /reflex_sf
        '''
        self.namespace = name
        rospy.init_node('reflex_hand')

        # Identifier to check whether the controllers are spawned successfully or not
        self.controller_startup = False

        # Wait for the controller process to done completely
        rospy.Subscriber('controller_spawner_done_init', Bool, self.controller_status_cb)
        while not self.get_controller_startup_status():
            pass
        rospy.sleep(0.4) # brief delay just so all of the additional informations from the
        # dxl packages get printed out

        # Start initiating motor controllers based on the input list from the launch file
        rospy.loginfo('Starting up the hand')
        self.input_motors = rospy.get_param('motors_list')
        self.motors = dict()
        self.motor_names = []
        for motor in self.input_motors:
            name = self.namespace + "_" + motor
            self.motors[name] = MotorClass(name)
            self.motor_names.append(name)

        # Setup subscriber for sending out messages to motors
        rospy.Subscriber(self.namespace + '/command',
                         reflex_msgs.msg.Command, self._receive_cmd_cb)
        rospy.Subscriber(self.namespace + '/command_position',
                         reflex_msgs.msg.PoseCommand, self._receive_angle_cmd_cb)
        rospy.Subscriber(self.namespace + '/command_velocity',
                         reflex_msgs.msg.VelocityCommand, self._receive_vel_cmd_cb)
        rospy.Subscriber(self.namespace + '/command_motor_force',
                         reflex_msgs.msg.ForceCommand, self._receive_force_cmd_cb)
        rospy.loginfo('ReFlex hand has started, waiting for commands...')
```

Figure 5-21 Modification of the Reflex package super class

The topic subscription is modified directly into the super class instead of the main initialization function as shown before, so that any further development sub classes will inherit this feature. With this modification, the fixed delay timer is eliminated and allow much more flexibility in motor initialization.

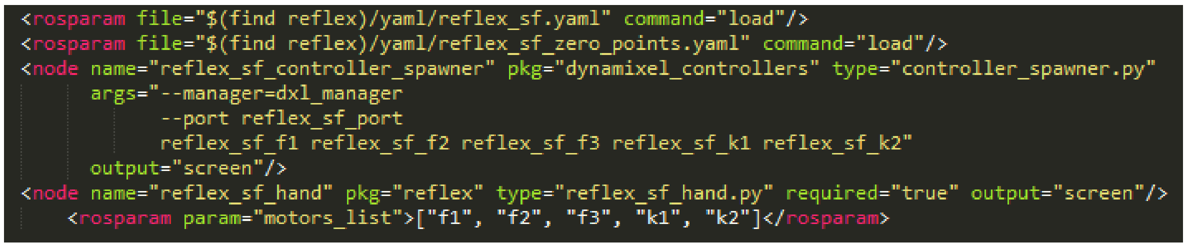
One more aspect required modification from the Reflex package is how the message is sent out to the controller to handle from the higher level. Currently, the namespace of the servos is manually declared and controller, thus requiring a subclass for every different system design.



```
def set_angles(self, pose):
    self.motors[self.namespace + '_f1'].set_motor_angle(pose.f1)
    self.motors[self.namespace + '_f2'].set_motor_angle(pose.f2)
    self.motors[self.namespace + '_f3'].set_motor_angle(pose.f3)
    self.motors[self.namespace + '_preshape'].set_motor_angle(pose.preshape)

def set_velocities(self, velocity):
    self.motors[self.namespace + '_f1'].set_motor_velocity(velocity.f1)
    self.motors[self.namespace + '_f2'].set_motor_velocity(velocity.f2)
    self.motors[self.namespace + '_f3'].set_motor_velocity(velocity.f3)
    self.motors[self.namespace + '_preshape'].set_motor_velocity(velocity.preshape)

def set_speeds(self, speed):
    self.motors[self.namespace + '_f1'].set_motor_speed(speed.f1)
    self.motors[self.namespace + '_f2'].set_motor_speed(speed.f2)
    self.motors[self.namespace + '_f3'].set_motor_speed(speed.f3)
    self.motors[self.namespace + '_preshape'].set_motor_speed(speed.preshape)

def set_force_cmds(self, torque):
    self.motors[self.namespace + '_f1'].set_force_cmd(torque.f1)
    self.motors[self.namespace + '_f2'].set_force_cmd(torque.f2)
    self.motors[self.namespace + '_f3'].set_force_cmd(torque.f3)
    self.motors[self.namespace + '_preshape'].set_force_cmd(torque.preshape)
```

Figure 5-22 Reflex package message handling

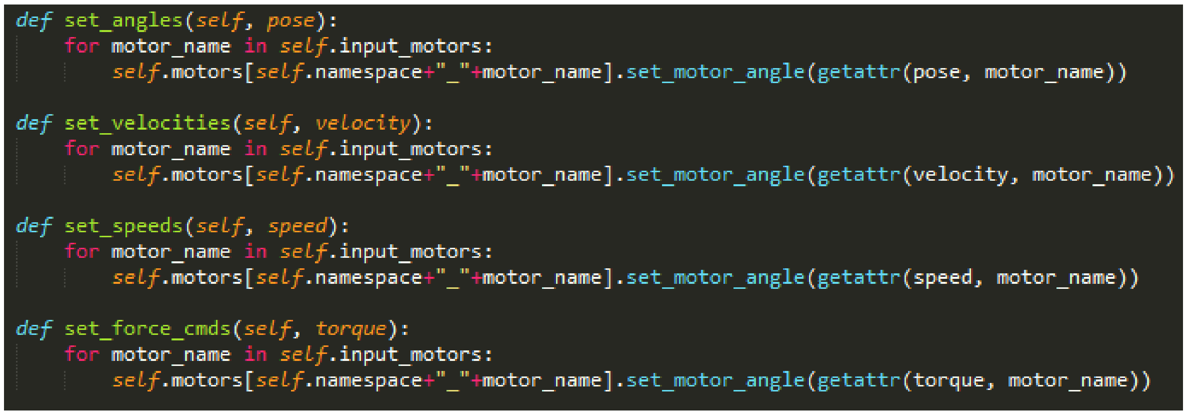
This implementation is fine but it is very system specific. Meaning the same code will not be able to work with a system setup of more than four servos. It would also require direct modification to the source code or creating a new subclass to run the new system. Therefore, a new modification is added by adding an additional parameter in the launch file containing the namespace for the servos. Then these functions can iterate through that namespace and send out the signal accordingly.



```
<rosparam file="$(find reflex)/yaml/reflex_sf.yaml" command="load"/>
<rosparam file="$(find reflex)/yaml/reflex_sf_zero_points.yaml" command="load"/>
<node name="reflex_sf_controller_spawner" pkg="dynamixel_controllers" type="controller_spawner.py"
      args="--manager=dxl_manager
            --port reflex_sf_port
            reflex_sf_f1 reflex_sf_f2 reflex_sf_f3 reflex_sf_k1 reflex_sf_k2"
      output="screen"/>
<node name="reflex_sf_hand" pkg="reflex" type="reflex_sf_hand.py" required="true" output="screen"/>
    <rosparam param="motors_list">["f1", "f2", "f3", "k1", "k2"]</rosparam>
```

Figure 5-23 Adding a new parameter for namespace

Reading the information from the input parameter, the package can just iterate through the namespace every time to send out the signals to the servos. This approach would only require a minor modification in the launch file for compatibility instead of going through the complete source code and modify the namespace line by line.



```python
def set_angles(self, pose):
    for motor_name in self.input_motors:
        self.motors[self.namespace+"_"+motor_name].set_motor_angle(getattr(pose, motor_name))

def set_velocities(self, velocity):
    for motor_name in self.input_motors:
        self.motors[self.namespace+"_"+motor_name].set_motor_angle(getattr(velocity, motor_name))

def set_speeds(self, speed):
    for motor_name in self.input_motors:
        self.motors[self.namespace+"_"+motor_name].set_motor_angle(getattr(speed, motor_name))

def set_force_cmds(self, torque):
    for motor_name in self.input_motors:
        self.motors[self.namespace+"_"+motor_name].set_motor_angle(getattr(torque, motor_name))
```

Figure 5-24 Sending out packets accordingly to the namespace input parameter

6. Making of the data glove

6.1. Glove Experiments

6.1.1. Capto Glove

- What is CaptoGlove

Our first solution is to use CaptoGlove, an existing product out on the market. CaptoGlove is glove that used for variety of purpose such as a normal mouse or Virtual Reality controller and we think with all its function we can easily control not only our robot hand but also the wrist movement of the arm.

- Why we choose to use it

CaptoGlove has total of 5 flex sensor with accelerometer of all three axes, therefore this glove has the capability to operate our robot hand easily. The glove also connected to the host machine through Bluetooth similar to wireless mouse, therefore we can control the robot hand remotely from the working station. Remote control is also convenient with the motion capture setup, which capture operator arm motion and map it to Baxter. CaptoGlove also had its own software and SDK for developers.

- CaptoGlove’s software

First, our team tried use its software and SDK to see the capable of streaming data from the glove to our ROS hand controller with rate of at least 20 Hz because it is the fastest rate that the robot hand can accept command. The biggest challenge that our team face was its software and SDK was developed on Windows, however our ROS controller was on Linux, therefore we need a way to communicate between the two system. The first solution is very basic, we would run two systems: one run on windows that connected through Bluetooth, then another run on Linux which operate the robot hand, the two system then communicate through Wi-Fi using socket communication. Our second solution for the problem was to change the CaptoGlove SDK to Linux, which solve the problem of multiple wireless system trying to communicate with each other and also later on we want add the glove into input of Trina system, therefore having a Linux based software to control the glove would be easier to integrate.

For the first solution, we ran the CaptoGlove SDK provided from the company which is written in C++. It took us a while to understand what their SDK have and how to used them and how we can get the data we need, the motion of bending finger to number then when we able to get those number we setup a server client socket communication every time it received the message of sensors. The flow of the solution we had is shown below.

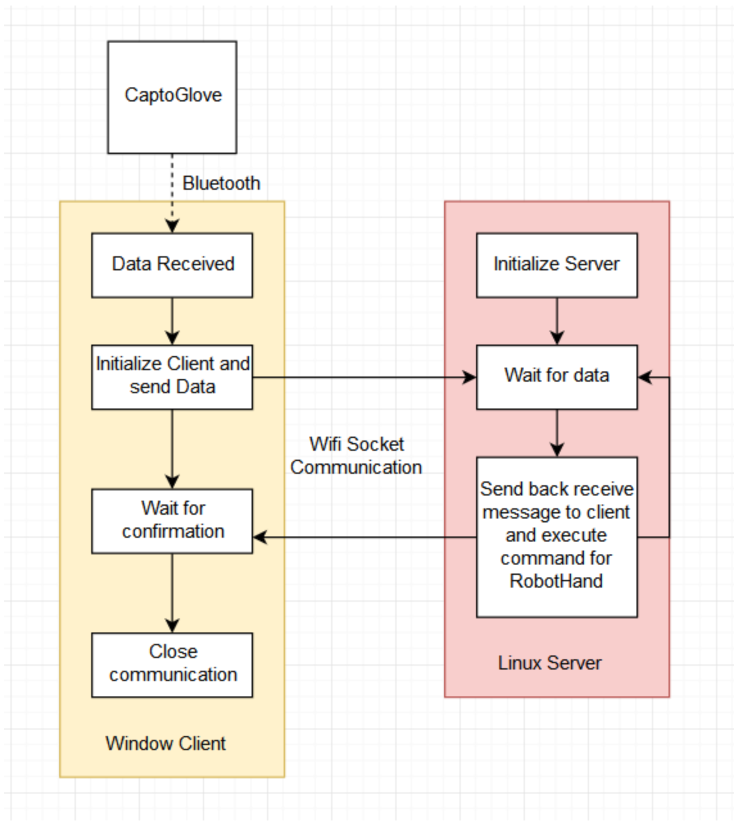


Figure 6-1 Flow Diagram of Server and Client for CaptoGlove Implementation

As in the figure above, we only need to initialize one server, however, we need to make a new client every time the client received information from the remote glove. Because of the way the glove SDK works, where we can only get raw data from the sensor through a constructor of a class, therefore it is very difficult and need modification to kept a single client holding communication while the program is running. There were also others issue regarding the SDK program worked, where the program started to crash and have exception cases we cannot solve because it happened inside a close source material from the CaptoGlove company. Therefore, the program could run for 20 to 30 second before terminated. During operation the data rate of transmission is also inconsistent as data received from Bluetooth CaptoGlove was unstable. The data rate could vary from 5 Hz to only 0.5 Hz. We were still able to control the robot hand and do minimal testing with it.

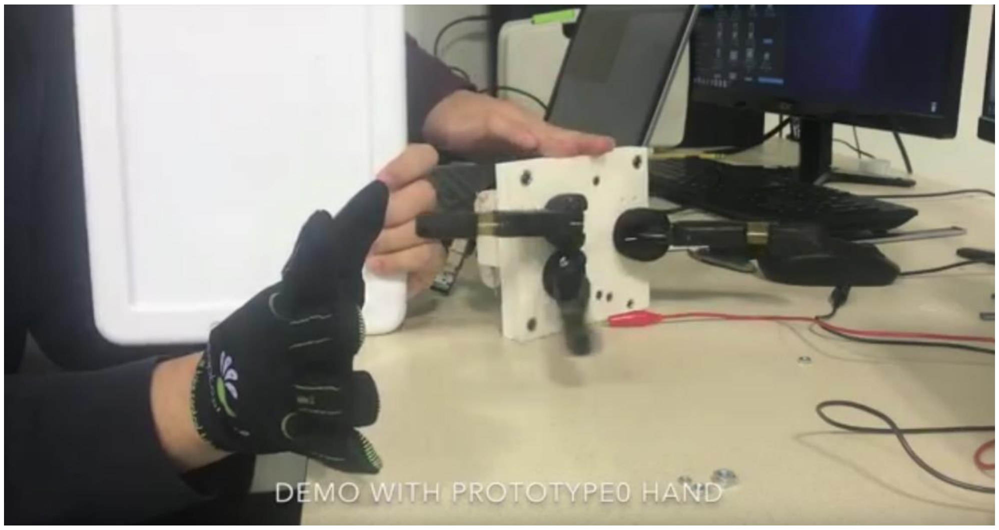


Figure 6-2 Demo of CaptoGlove and First Prototype Hand. [Video link]

As in the video, there were a noticeable delay when we move our hand and when the robot actually move (around 1 to 2 seconds). This is caused by the latency of the whole process of having client server system addition to the Bluetooth wireless communication. The system was also very difficult to debug because we used SDK from CaptoGlove

For our second solution, because the CaptoGlove is close source therefore trying to move the SDK from window to Linux was very complicated and challenging, therefore we did not follow this solution.

Pros: the glove is wireless so user don't have to stay in one place, the glove has five flex sensors and accelerometer for all axis.

Cons: Data rate is unstable and low therefore we cannot control the robot hand in real-time.

- Conclusion

Based on those testing, we figure out that this implementation does not meet the requirement of the problem where we want to map human finger motion to Robot motion. With unstable data rate and delay in the system, we conclude that we cannot use this solution. Therefore, a different solution is needed

6.1.2. <u>Data Glove Implementation</u>

- Motivation

At first because we think using existing product for our project will be quicker than making our own input device, however because of all the challenges and difficulty from CaptoGlove, we decided to build our own data glove input.

- Initial Design,

We want to glove to be able to control as much degree of freedom of the hand as possible, however with limitation human motion and the robot hand constraint motion is different from human, the preshape for both the thumb and two index finger is very difficult to mimic. Therefore, we only decide to first just have three degree of freedom on our glove to control three finger, for the two other degree of freedom, we decide to use GUI to control the robot hand. We used similar sensor with the CaptoGlove. We decided to use UBS wire to connect from the controller of the glove direct to PC to avoid any wireless (Bluetooth) issue we saw from the CaptoGlove.

We also decided to use the glove from CaptoGlove as our testing glove, because it has two layer separately, one for our finger and one for the flex sensor to be put in. There are two way to put the flex sensor, on top or go under our finger. We have tested both and the first way (sensor goes on top) has minimal effect on wire and bending that could damage the sensors and wire permanently compare to the second solution.

We also want our controller unit small enough to fit our glove and has minimal effect to the user. We take a look at a few of possible microcontroller and products on the market and we decided to use the Arduino Nano for the task. There are three benefit of Arduino Nano compare to others products. First, Arduino Nano is small and can fit perfectly in our glove. Second, Arduino has packages for ROS, running ROS node directly on the microcontroller itself, therefore it is easy to implement a simple ROS node that get information from the sensor and publish information at a certain rate. Third, Arduino Nano has few other function such as more analog and digital ports and Bluetooth integrated, therefore it would be better for future development to use the Arduino Nano.

The flex sensor circuitry needs a voltage divider, because according to the datasheet of flex sensor, its resistant vary from 20k to 30k ohm. Therefore, we have made a few calculations to determine which resistor we should use to make sure we can capture the largest range out of the flex sensor. Our calculation shown that if we used resistor has value in between the value of flex sensor maximum and minimum bending will be better, therefore we decided to use 100k resistor.

The design for the glove shown in figure below

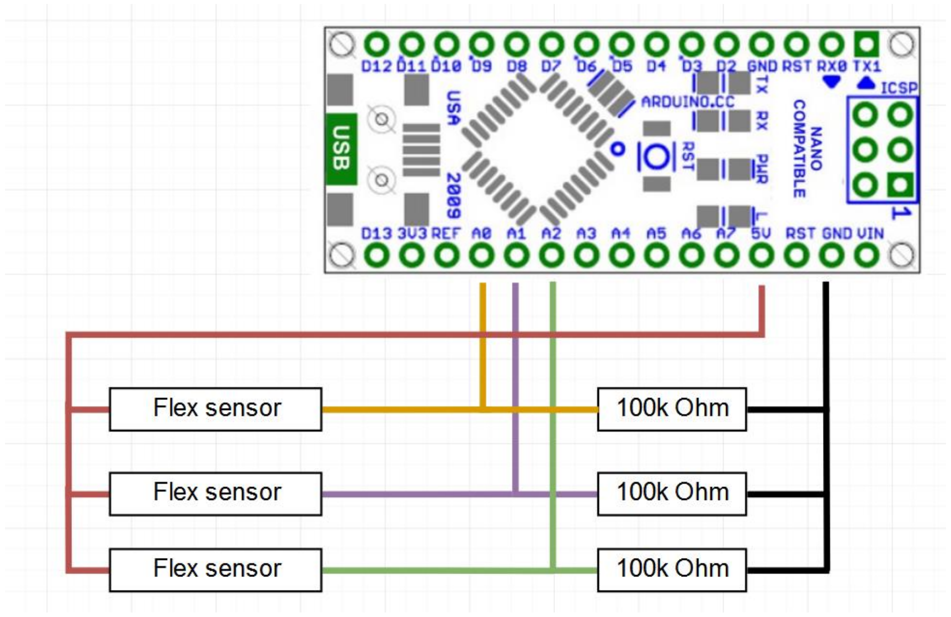


Figure 6-3 Simple initial design and circuit of the glove

The port that we used to read the value from the sensor are basically analog read (ADC) to read voltage across the voltage divider. The value then will need to send to the host machine and publish to a ROS topic.

- Implementation:

For the sensor connection with the Arduino, at first we want to make a PCB that connect all the sensors and the Arduino Nano together. However, because our circuit is very simple therefore order to make a PCB will cost time and money more than necessary. Therefore, we decided to use a solder-able breadboard for making the circuitry, not only it is simple to make but also because for future development we can easily add more sensors and change the circuitry easily without having to redesign and order a different PCB.

The result for the implementation is shown below

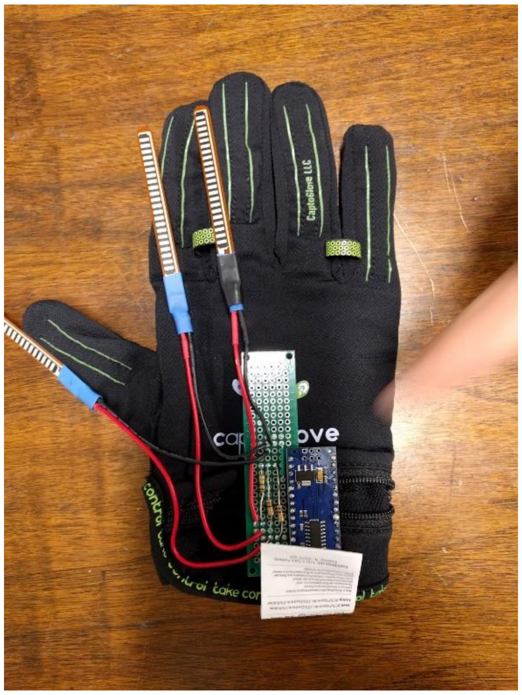
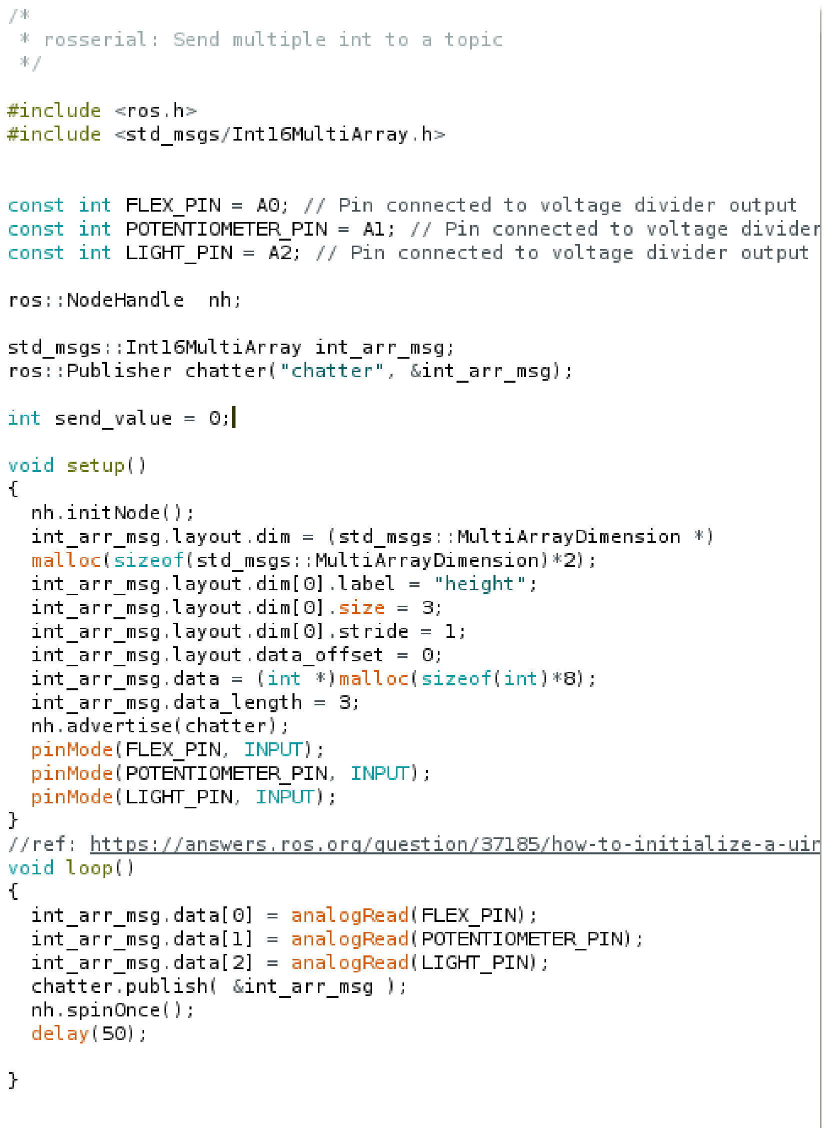


Figure 6-4 Implementation of the glove

For the Arduino code side, we used the default ROS packages for Arduino and able to publish the data rate to 100Hz, which is better than what we wanted. The implementation for coding part is pretty simple, one thing we need to note is that we need to use malloc for making dynamic memory for sending ROS message, which is just array of integers. This process took a lot of memory in the Arduino Nano, however the code and everything else still run good.

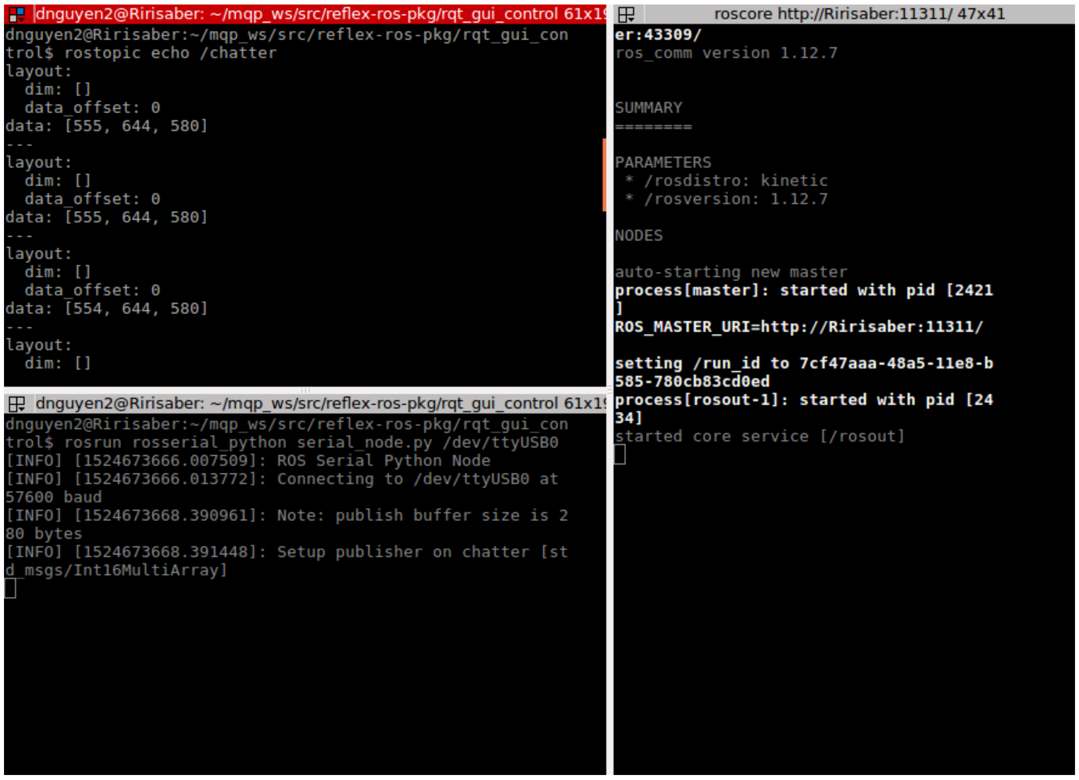


```
/*
 * rosserial: Send multiple int to a topic
 */

#include <ros.h>
#include <std_msgs/Int16MultiArray.h>


const int FLEX_PIN = A0; // Pin connected to voltage divider output
const int POTENTIOMETER_PIN = A1; // Pin connected to voltage divider
const int LIGHT_PIN = A2; // Pin connected to voltage divider output

ros::NodeHandle  nh;

std_msgs::Int16MultiArray int_arr_msg;
ros::Publisher chatter("chatter", &int_arr_msg);

int send_value = 0;

void setup()
{
  nh.initNode();
  int_arr_msg.layout.dim = (std_msgs::MultiArrayDimension *)
  malloc(sizeof(std_msgs::MultiArrayDimension)*2);
  int_arr_msg.layout.dim[0].label = "height";
  int_arr_msg.layout.dim[0].size = 3;
  int_arr_msg.layout.dim[0].stride = 1;
  int_arr_msg.layout.data_offset = 0;
  int_arr_msg.data = (int *)malloc(sizeof(int)*8);
  int_arr_msg.data_length = 3;
  nh.advertise(chatter);
  pinMode(FLEX_PIN, INPUT);
  pinMode(POTENTIOMETER_PIN, INPUT);
  pinMode(LIGHT_PIN, INPUT);
}
//ref: https://answers.ros.org/question/37185/how-to-initialize-a-uir
void loop()
{
  int_arr_msg.data[0] = analogRead(FLEX_PIN);
  int_arr_msg.data[1] = analogRead(POTENTIOMETER_PIN);
  int_arr_msg.data[2] = analogRead(LIGHT_PIN);
  chatter.publish( &int_arr_msg );
  nh.spinOnce();
  delay(50);

}
```

Figure 6-5 Code implementation for running ROS node and publish message at a certain rate from Arduino Nano

- Testing,

For testing our glove we separate into multiple part, first is testing the sensors, then test the circuitry ROS topic then running the glove on simulation and then on the real robot. The first test was simple test that just printing out the value of sensors themselves when we tested with different resistor.



```
dnguyen2@Ririsaber: ~/mqp_ws/src/reflex-ros-pkg/rqt_gui_control 61x19
dnguyen2@Ririsaber:~/mqp_ws/src/reflex-ros-pkg/rqt_gui_con
trol$ rostopic echo /chatter
layout:
  dim: []
  data_offset: 0
data: [555, 644, 580]
---
layout:
  dim: []
  data_offset: 0
data: [555, 644, 580]
---
layout:
  dim: []
  data_offset: 0
data: [554, 644, 580]
---
layout:
  dim: []
```

```
dnguyen2@Ririsaber: ~/mqp_ws/src/reflex-ros-pkg/rqt_gui_control 61x19
dnguyen2@Ririsaber:~/mqp_ws/src/reflex-ros-pkg/rqt_gui_con
trol$ rosrun rosserial_python serial_node.py /dev/ttyUSB0
[INFO] [1524673666.007509]: ROS Serial Python Node
[INFO] [1524673666.013772]: Connecting to /dev/ttyUSB0 at
57600 baud
[INFO] [1524673668.390961]: Note: publish buffer size is 2
80 bytes
[INFO] [1524673668.391448]: Setup publisher on chatter [st
d_msgs/Int16MultiArray]
```

```
roscore http://Ririsaber:11311/ 47x41
er:43309/
ros_comm version 1.12.7


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.7

NODES

auto-starting new master
process[master]: started with pid [2421
]
ROS_MASTER_URI=http://Ririsaber:11311/

setting /run_id to 7cf47aaa-48a5-11e8-b
585-780cb83cd0ed
process[rosout-1]: started with pid [24
34]
started core service [/rosout]
```

Figure 6-6: Raw value reading as a ROS Topic. Data sends from the glove.

The value for each sensor (top left corner terminal) is different however in the range that we expected and also our ROS node setup seems to work fine.

The next test is using those value map them out to fit the range and motion of the control for our robot hand and real-time human control

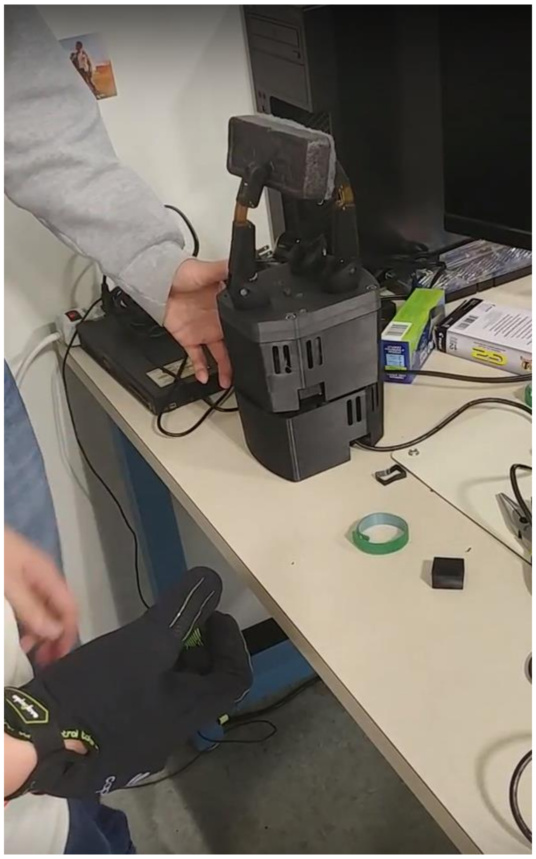


Figure 6-7: Demo of the glove with our MQP hand

We notice a few things when using the glove to control the hand. When we control the Reflex SF, the rate that the robot can accept value is around 5 Hz, if it accept anything higher than that, the robot will stack up all the position the glove send. This create big delay in the system. With our MQP hand, the rate that the robot can accept the value is only 4Hz, therefore if we used the same setup like the ReflexSF, the control loop will receive some delay in the system.

6.2. Glove housing

With the electrical components of the glove figured out, the next step is to make a housing to safely wear and shield the circuit board. Since we have had the CaptoGlove, we decided to follow the same design principle for our circuit board. The design goal is to have a housing that can render the board in place during motion sessions with enough rooms for the sensors to come out and USB connection, which not only serve as a data transmission line instead of Bluetooth protocol but also a power source for the board to function properly.

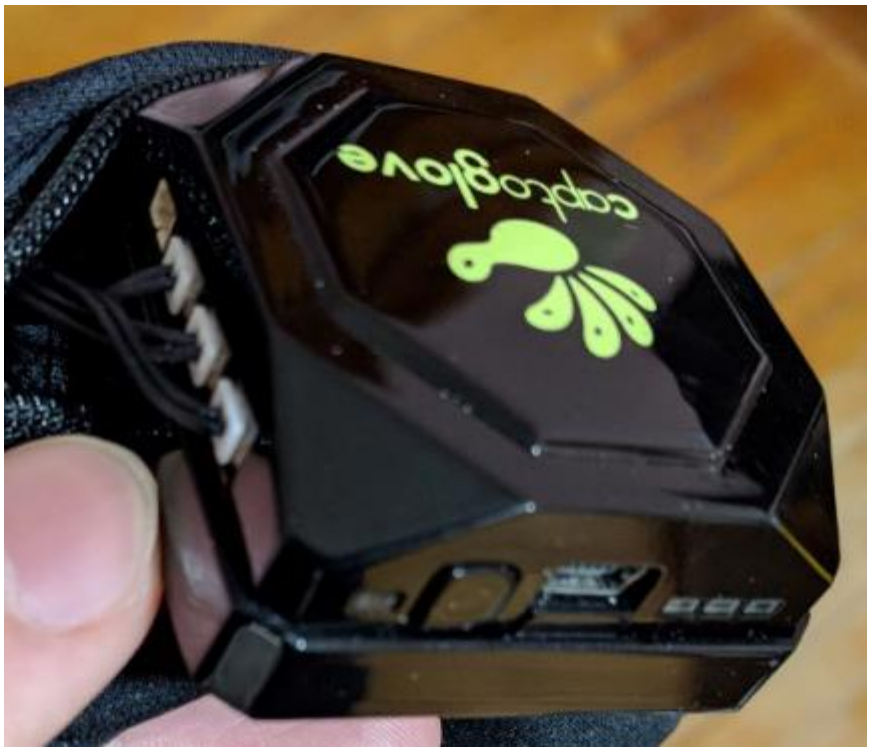


Figure 6-8 CaptoGlove housing with USB and sensor ports

Once the housing is designed and manufactured, it will be taped together using electrical tape to eliminate additional room requirements for screws. Glue was not considered in the case of further addition or repair. There is already a small area of Velcro provided inside the glove, thus is the main mounting mechanism for the CaptoGlove housing.

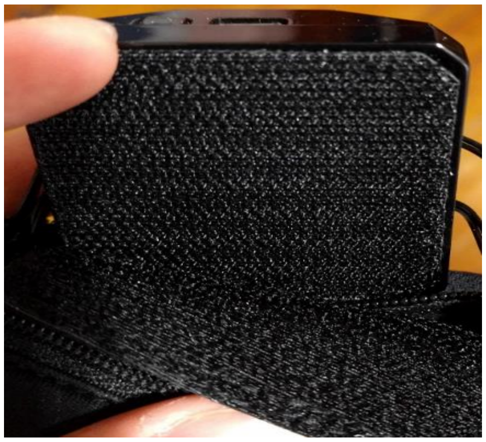


Figure 6-9 Provided Velcro surface beneath the CaptoGlove textile

Therefore, the goal of the design will be to have flat bottom surface such that Velcro can be stick to so that the housing can be mounted securely onto the glove. The Velcro tape can be purchased pretty cheap from most online purchase. For the case of this design, the following Velcro tape was considered from Amazon.

Figure 6-10 Velcro tape used for glove housing design

A thin slice of Velcro was more desirable in this case; as multiple slide of Velcro tape can be applied until the surface becomes completely covered. With the goal basically layout the housing design process was pretty straight forward measure and design. The final design consisted of two pieces. The bottom piece will have a flat surface for the board to reside in with supporting ridges on the side just so the boards can comfortably sink into and stayed in place. There are also half holes designed for the sensor wiring to come out as well as the USB cable connection port.

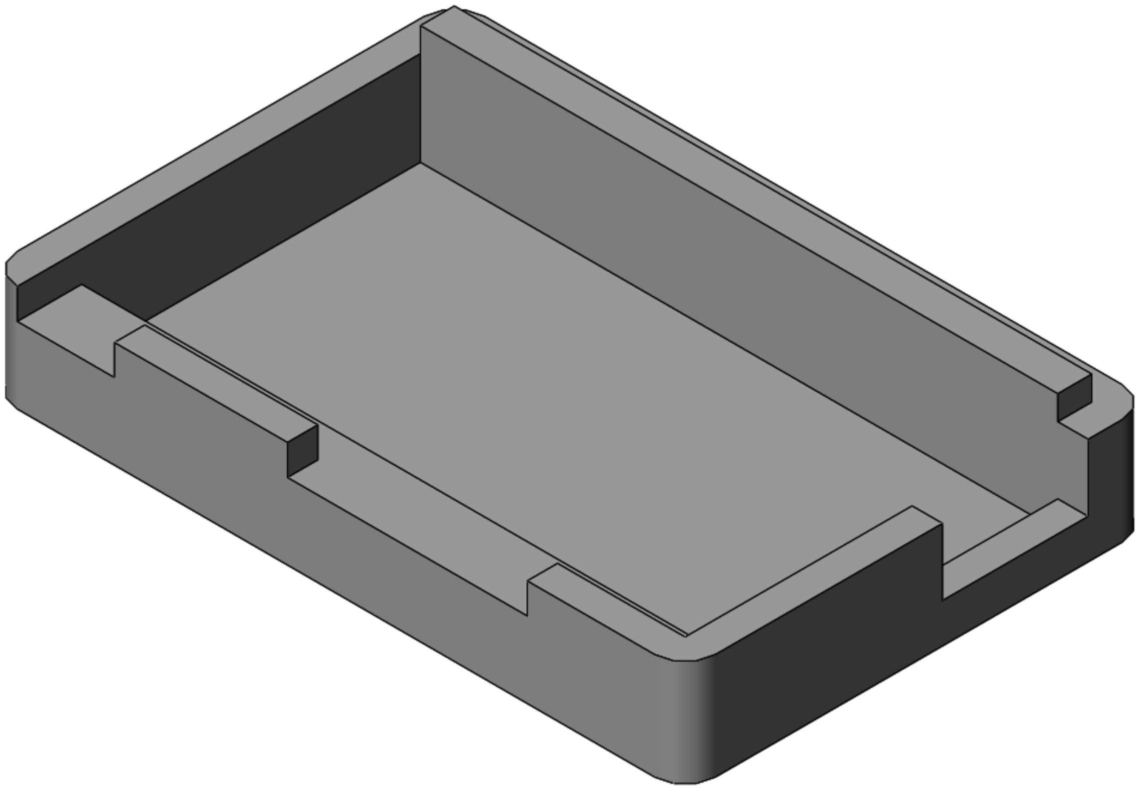


Figure 6-11 Bottom housing piece with uneven support ridges

The other piece is the top that will lay on top of the circuit boards, which are already fixed in place on the bottom piece. The two piece putting together will create a gap between the ridges just small enough to hold the circuit boards in place and not wiggling around. On the top surface there is also slots created for additional air ventilation for cooling just in case. The ridges on the top piece is also un even to match the dimension of the electrical board with half holes to completely allow the sensors and USB cable to connected. The total wall thickness of the design turns out to be 1mm thick. As this housing does not really have to support anything much, it will only act as a cover to provide the board from the frictions caused by the glove during motion sessions.

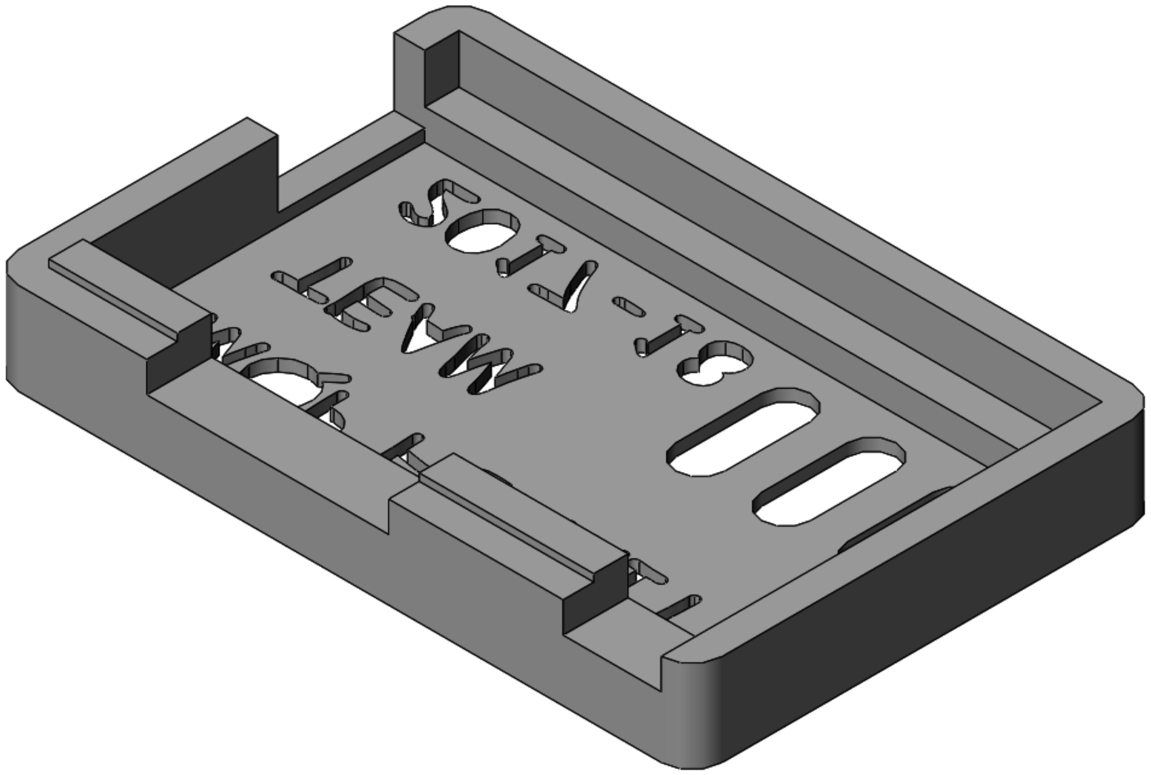


Figure 6-12 Top housing piece with ridges, holes and slots

The design is printed out using the same manufacturing method for the robotics hand and secured together with electrical tape and secured to the glove with electrical tape. It was slightly bigger than the design from CaptoGlove, which makes sense considering there was no single custom made circuit board and everything was bought off the shelf then soldered together.

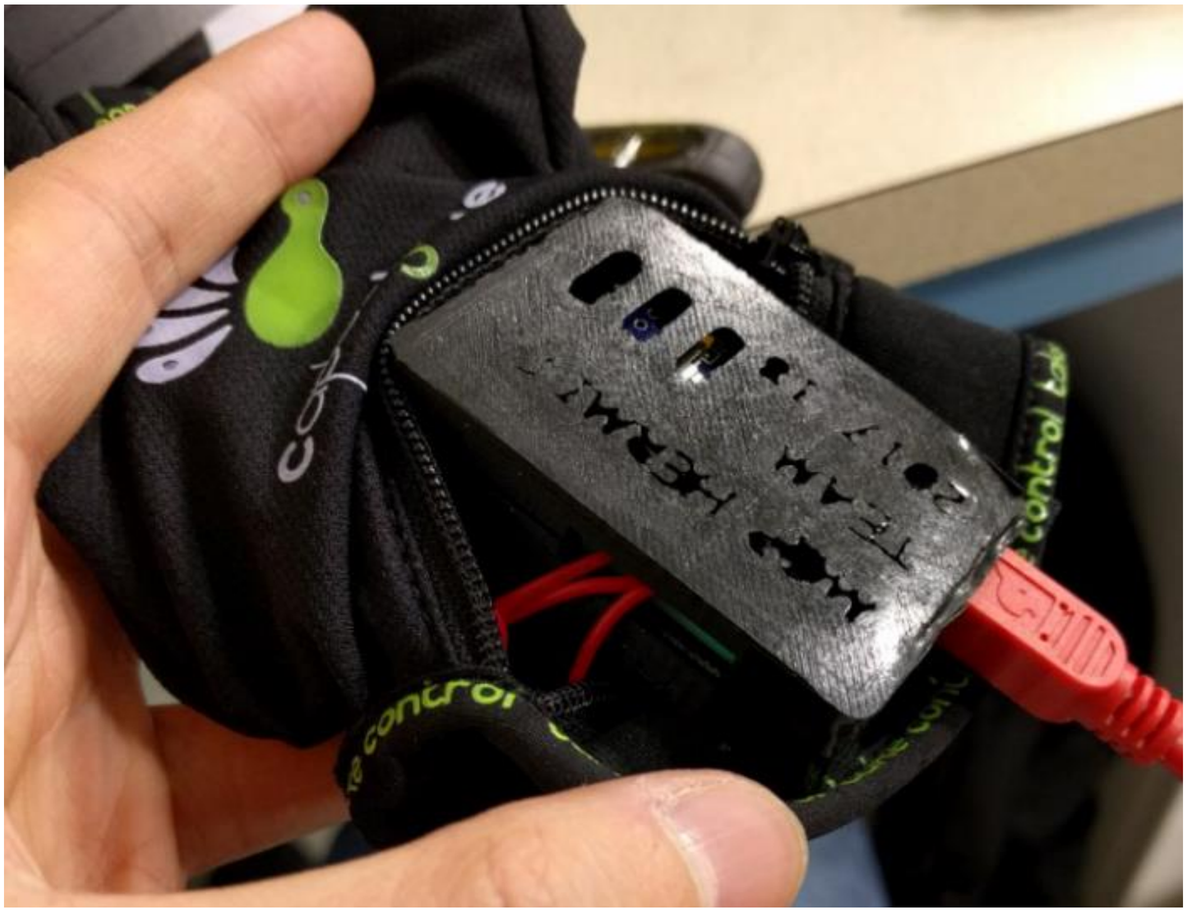


Figure 6-13 Complete data glove built

7. Result

Through multiple hardware design and modification iterations, the robotics manipulator has been assembled and successfully tested for operation. The manipulator is capable of dexterous manipulation with the aid of the control glove, though the result is pretty unstable and slow. However, with some improvement, it is completely possible for this set of manipulator and controller glove would be completely capable of such tasks.

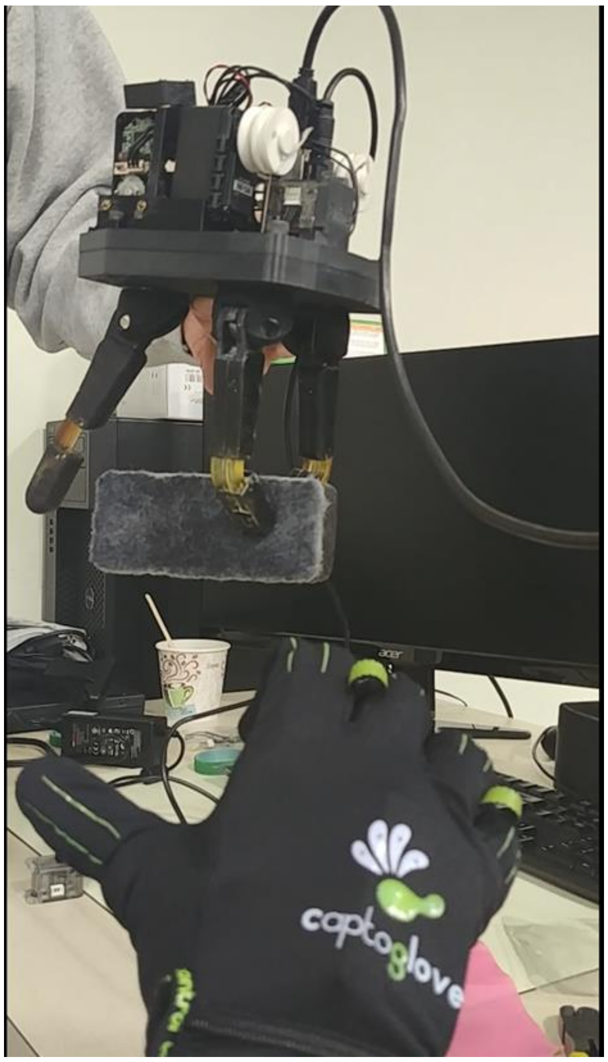


Figure 7-1: Human controlling the robotics hand to pick up a board eraser

Despite several implementation design, the data transfer rate is not quite as high as it was originally expected. Due to the heartbeat signal sending back from servos to the ROS node to monitor the hand status, as well as the stack handling method for messages sent through the ROS network, the data transfer rate was reduced from the original desired rate of 100Hz to 33.33Hz. This reduction leads to a further effect on the response time of the control system as well. However, considering this as a base foundation for future research and improvements, the project is considered a success.

8. Discussion

Although the project has been completed, there are still rooms for future development of the system. Firstly, the circuit board of the robotics hand can really use a complete redesign. Considering the lack of circuit board design on our current team, we had to use the board design for the Reflex One from RightHand Robotics to control. Since this board is specifically designed to fit into the Reflex One, it has a distinctive shape that ended up taking a lot of space within our mechanical design. If the board were to be redesigned, the servo layout and housing design can be adjusted. This would result in a decrease in the size of the robotics hand. Moreover, the current board design only supports one power plug for the fan, while the current mechanical design would require two fans for air circulation and cooling. Therefore, a completely new board design, follow up with some mechanical adjustment, would result in a more compact and thermodynamically efficient.

Secondly, the housing on the data glove definitely need improvement as well. As mentioned earlier, due to the limitation to circuit board design, the electrical components on the data glove are created without any necessary filtration for the sensors, also using a bread board to connect the components has significantly increased the size of the electrical components in total, resulting in a bigger, chunkier housing required. Also this method also results in a less secured circuit. The current data glove seems to be working without much significant issues, however, the signal received from the processor proves to be very noisy. Additionally, more features would result in a better data glove as well, such as wireless protocol, portable power source, additional sensors such as IMU, encoders, etc. Therefore, the data glove really needs some improvement in term of the electrical design and manufacture as well.

Thirdly, this might not have too much effect on the current system, but it should be better to swap the AX-12A servos to the more recent XL430-W250. The servo itself is not much more expensive than the AX-12A but it is more compatible with the XL-320, as they both belongs to the XL series offered by Dynamixel. The XL-320 servo cannot be changed as it is the only model offered by Dynamixel that has a through hole for the tendon to thread through from the finger into the pulling motor. The XL430-W250 has very similar controlling table to the XL-320. This will allow synchronous and bulk operations between servos that can ultimately result in a reduction of traffic within the UART communication line. Furthermore, the XL430-W250 supports Dynamixel's Protocol 2.0 that allows more security data transmission. The drawback, however, is modification in the mechanical design as the dimensions are expected to be different between the XL430-W250 and AX-12A. Furthermore, this model was not experimented with before since it only released in recent years. However, the improvements should definitely worth the drawback.

9. Conclusion

Through WPI's project presentation day, this MQP has had its chance to be shown to many parents and students. Many of the audience has expressed their interest toward this project. The main reason is allowing them to directly control the manipulator using the control glove without requiring any prior knowledge to motion planning or robotics controls and such. Furthermore, the idea is quite new to the crowd as well, such that one person has even expressed his interest toward the system for his robotics integration company. Lastly, all of the parts manufactured for the system can either be bought off the shelf or manufactured quickly with the aid of 3D printing. Through this presentation day, the team was able to gage various invaluable feedback from the professional and from the general crowd alike.

10. Appendices

The mechanical design iteration of this project can be accessed via this GitHub repository.

The Dynamixel servo controller package can be accessed via this GitHub repository.

The robotics hand controller package can be accessed via this GitHub repository.

The complete video demonstration can be accessed via this Google drive folder.

11. Reference

[1] Cleary, Kevin, and Charles Nguyen. "State of the Art in Surgical Robotics: Clinical Applications and Technology Challenges." Computer Aided Surgery, vol. 6, no. 6, 2001, pp. 312-328, http://dx.doi.org/10.1002/igs.10019, doi:10.1002/igs.10019.

[2] Ackerman, Evan. "This is the most Amazing Biomimetic Anthropomorphic Robot Hand we'Ve Ever Seen.", 2/18/, 2016, https://spectrum.ieee.org/automaton/robotics/medical-robots/biomimetic-anthropomorphic-robot-hand.

[3] "Dexterous Hand – Shadow Robot Company.", https://www.shadowrobot.com/products/dexterous-hand/.

[4] Shadow Dexterous Hand Technical Specification E2 Series. Shadow Robot Company, 2015, https://www.shadowrobot.com/wp-content/uploads/shadow_dexterous_hand_technical_specification_E_20150827.pdf.

[5] Odhner, Lael U., Raymond R. Ma, and Aaron M. Dollar. "Exploring Dexterous Manipulation Workspaces with the iHY Hand." Journal of the Robotics Society of Japan, vol. 32, no. 4, 2014, pp. 318-322.

[6] L.U. Odhner, L.P. Jentoft, M.R. Claffee, N. Corson, Y. Tenzer, R.R. Ma, M. Buehler, R. Kohout, R.D. Howe, and A.M. Dollar

[7] Demaitre, Eugene. "Soft Robotics Raises $5M for Flexible Fingers." Robotics Business Review, Robotics Business Review, 29 Dec. 2017, www.roboticsbusinessreview.com/agriculture/soft_robotics_raises_5m_for_flexible_fingers/.

[8] Brown, Eric, et al. "Universal Robotic Gripper Based on the Jamming of Granular Material." Proceedings of the National Academy of Sciences, National Academy of Sciences, 2 Nov. 2010, www.pnas.org/content/107/44/18809.

[9] Howe, Robert, et al. "Inexpensive, Durable Plastic Hands Let Robots Get a Grip." IEEE Spectrum: Technology, Engineering, and Science News, IEEE Spectrum, 21 Nov. 2014, spectrum.ieee.org/robotics/humanoids/inexpensive-durable-plastic-hands-let-robots-get-a-grip.

[10] "CRC-16 (IBM/ANSI)." ROBOTIS e-Manual, emanual.robotis.com/docs/en/dxl/crc/.