Exomuscular Robotic Sleeve for Upper Limb Stroke Rehabilitation

Major Qualifying Project

by

Ritesh Adhikari (ECE)

Mike Cross (ME)

Tim Forrest (ME)

Hosung Im (RBE/BME)

Jason Klein (RBE)

Anselm Mak (RBE/ME)

Julieth Ochoa (RBE)

Advised by

Gregory S. Fischer (ME/RBE)

Edward A. Clancy (ECE)

Cagdas D. Onal (ME)

AIM
WPI Automation and Interventional Medicine Laboratory

WPI

ABSTRACT

Traditional physical therapy for upper-limb impaired post-stroke patients does not involve the use of robotic equipment with the ability to quantitatively assess mechanical performance. As a consequence, it becomes difficult to study the therapeutic results achieved by the patient through the different stages of motor training. The use of robotic systems to assist upper-limb rehabilitation, however, offers possible improvements to these limitations. The aim of this project was to develop an exomuscular arm that could be actuated through a system of Bowden cables linked to precision DC motors housed in an actuation platform. The system assists and controls flexion and extension of the five fingers and the elbow, as well as pronation and supination of the wrist. Through a sensor array located throughout, a feedback system is able to collect quantitative data on joint angles, fingertip forces, motor torques, and control all degrees of freedom utilizing this data and several on-board processors.

EXECUTIVE SUMMARY

I. Background

Hemiparesis causes the majority of post-stroke patients to experience limited dexterity, sensitivity, strength, and coordination in their affected upper extremity. Due to this ensuing muscle weakness that severely affects their motor function, a great number of hemiparetic stroke survivors seek professional rehabilitative and/or assistive help. The conventional rehabilitation process for upper-limb impaired patients is greatly affected by time and money restrictions, which often results in patients not being able to achieve a maximum potential for recovery. The lack of quantitative data available for therapist to provide a complete assessment also limit the possible outcomes of therapeutic training. The use of robotic systems to assist upper-limb rehabilitation, however, offers possible improvements to these current methods. Through the use of robotics, the team aimed to create a device that would not have these shortcomings and would provide a holistic physical therapy solution.

II. Previous Related WPI MQP Work

A robotic glove system was designed in 2012 by Delph et al. [1], which used cable arrays starting at the fingertips and terminating at servomotors placed in a backpack to actuate finger movements. A soft robotic exomuscular brace that actuated the elbow was developed in 2013 by Brauckmann et al. [2] to serve as an assistive device to upper-limb impaired patients.

III. Proposed System

The developers of this 2014 MQP aimed to integrate these two designs into a unified system with added capabilities, making it a viable device to be used by upper-limb hemiparetic stroke survivors for rehabilitative and assistive purposes. Our proposed sleeve achieves

exomuscular actuation through a system of Bowden cables linked to precision DC motors housed in an actuation platform. The device provides the motorized assistance of flexion and extension of the fingers and elbow, as well as pronation and supination of the wrist. A feedback system collects quantitative data on position and pressure through a sensor array in order to control all degrees of freedom utilizing several on-board processors.

IV. Mechanical Design

The glove itself (see **Figure A**) is a thin neoprene glove modified with two zippers running parallel from the base of the knuckles to the distal end of the wrist on the dorsal side of the hand, which allows for ease in putting the glove on a stroke patient suffering from decreased hand function. Along the back of each finger and sown into the glove material is a pocket in which the flex sensors and leads for pressure sensors are inserted, with their electrical connections available at the back of the hand. These flex and pressure sensors are used to measure the angle of flexion/extension of the fingers and the force at the fingertips, respectively.

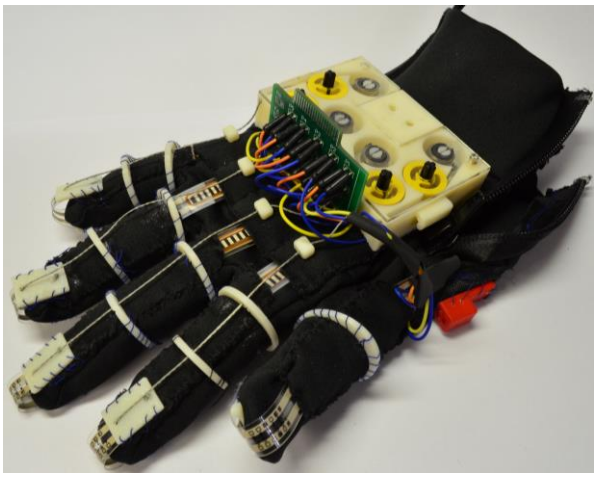


Figure A: Fully Assembled Exomuscular Glove with Flex and Pressure Sensors.

A detachable mechanism was designed for the wrist to allow cables to be disengaged for easy donning of the device (see **Figure B**). Force is transmitted to a Bowden system through a detachable axle that meshes the lower set to the upper set of yellow pulleys. Constant force springs are used to keep the cables in tension and in pulley groove while the mechanism is detached. The wrist support beam provides spring support in a set range of motion defined by the trapezoid shaped hard stops.

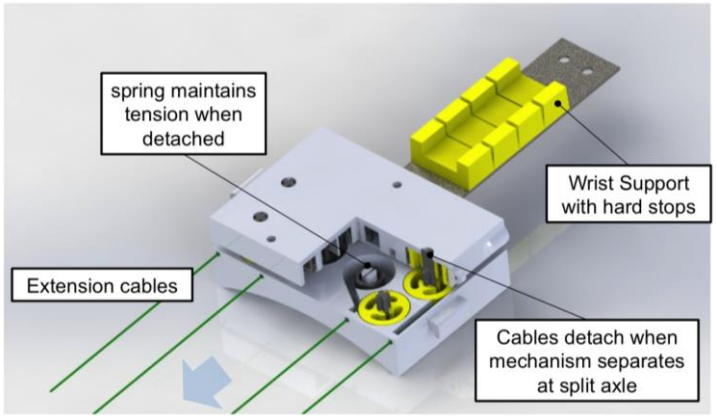


Figure B: Detachable Wrist Mechanism

A Bowden system of cables linked to motors is used to actuate flexion and extension of the elbow and pronation and supination of the wrist (see **Figure C**). Springs on ends of Dyneema cables allow for series elastic actuation. Force sensing is performed by measuring spring deflection through a membrane potentiometer. These redesigned smaller brace segments allow for better conformity to patient.

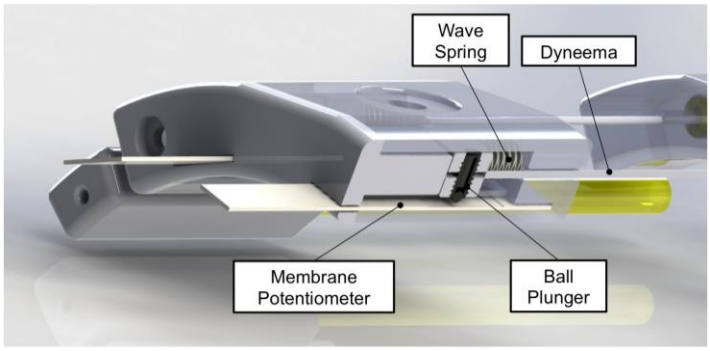


Figure C: Series Elastic Sensor for the Elbow

V. Electrical Architecture

The system includes three custom printed circuit boards (PCB), each with its own microcontroller (Microchip's PIC32MX795F5F512L). These Main, Sensor, and Motor boards are programmed to handle individual tasks with their specific components for each task. The Main board is directly connected with the Motor and Sensor boards as well as a computer running Ubuntu. A high-level functional diagram of these circuit boards and their communication with the computer can be seen below in **Figure D**.

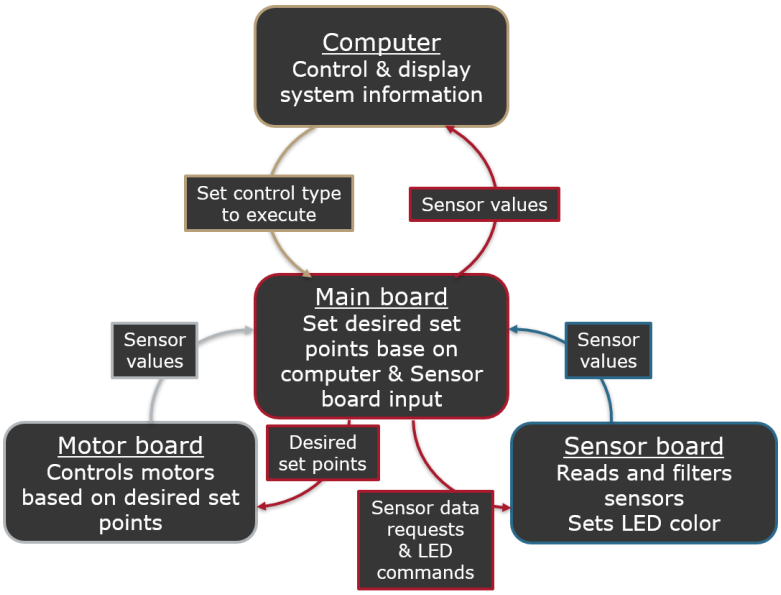


Figure D: High-Level System Functional Diagram.

VI. Results and Deliverables

The final prototype of the system (see **Figure E**) allows for motorized flexion and extension of the fingers and the elbow, as well as the pronation and supination of the wrist. This third iteration of the system improved from the previous two by adding a modular motor units that can be easily removed and modified, a wrist support with trapezoid-shaped hard stops, and a detachable mechanism that allows the extension cables to be disengaged while

dinning the device. Additional achievement of the team include the design of more compact and comfortable elbow braces, more robust design of finger cable guides to prevent losses from glove elasticity, as well as more compact and workable design for cable tensioning. The system is capable of accurately collecting and interpreting sensor data. Improvements were also made with the development of several PCBs for dedicated tasks, such as reading and interpreting sensor data, actuating and controlling the system, and communicating with the master computer. A demonstration video of the system can be found at the following link:

https://www.youtube.com/watch?v=Xs_7N57YGdU&feature=youtu.be

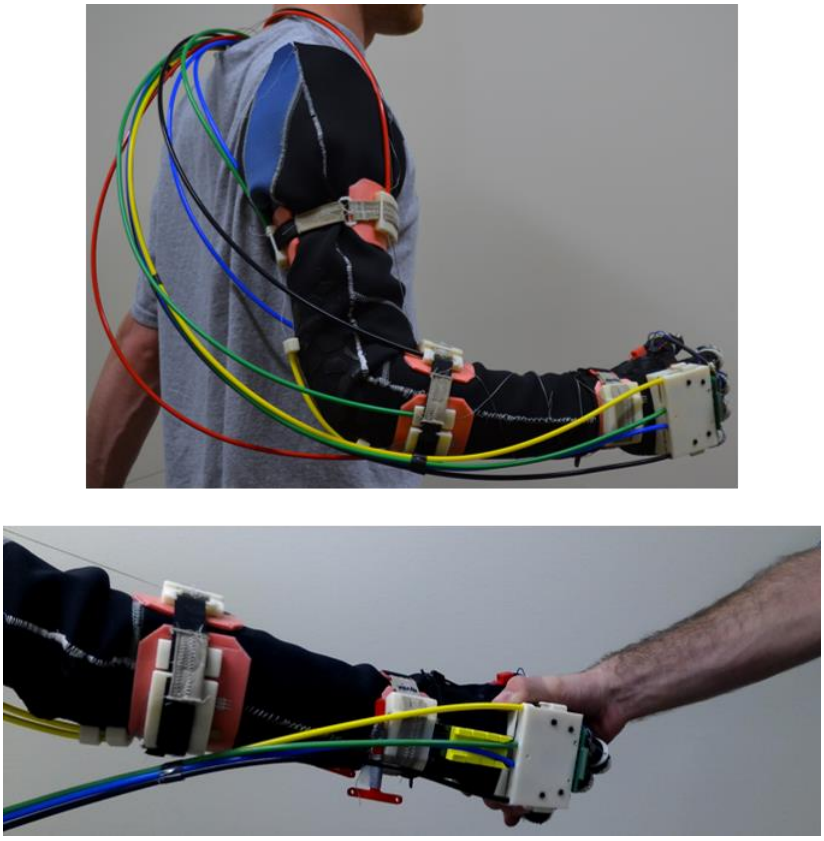


Figure E: Exomuscular Sleeve Final Prototype.

ACKNOWLEDGEMENTS

The developers of this project would like to thank the following individuals and organizations for their valuable support and contributions to the success of this MQP:

- Professors Gregory Fischer and Edward Clancy from Worcester Polytechnic Institute for advising our project, providing continuous guidance, advice, and feedback, as well as for their efforts in challenging us to properly understand the full scope of this project and enhance our learning experience.
- Robotic Engineering Doctoral Candidates Michael Delph and Christopher Nycz for their advice and support, especially for their anecdotal recommendations based on their experiences when working in their undergraduate senior projects.
- Joseph St. Germain, Alex Camilo, Robert Boisse, and Kevin Harrington from Worcester Polytechnic Institute's Robotics, AIM, and ECE labs (respectively) for their technical assistance.
- Aaron Cornelius, Erica Stults, Kevin Arruda, and Matt DiPinto for their assistance with fabricating parts.
- Worcester Polytechnic Institute's AIM Lab for supporting this MQP by providing a working environment and necessary supplies, thus making this project possible.
- The Robotics, Mechanical, and Electrical Engineering departments at Worcester Polytechnic Institute for contributing with funds and resources to supporting this project.

AUTHORSHIP

Abstract: Written by Julieth. Edited by Jason.

Executive Summary: Written by Julieth.

1. Introduction: Written by Julieth. Edited by Mak.

2. Background

2.1. Biomechanics of the Upper Extremity: Written by Mak, Tim, and Ritesh. Edited by Julieth.

2.2. Muscle Weakness and Motion Impairment in the Upper Extremity: Written by Mike and Tim. Edited by Julieth.

2.3. Conventional versus Robot-Assisted Rehabilitation: Written by Julieth and Jason.

2.4. Upper-Extremity Robotic Rehabilitation Methods: Written by Julieth. Edited by Jason.

2.5. Existing Upper-Extremity Rehabilitative and Assistive Devices: Written by Julieth. Edited by Jason.

2.6. Previous Work Leading to this Project: Written by Mak. Edited by Julieth.

3. Project Strategy

3.1. Client Statement: Written by Julieth. Edited by Mak.

3.2. Primary Objectives: Written by Julieth. Edited by Mak, Mike, and Hosung.

3.3. Primary Constraints: Written by Julieth. Edited by Mak, Mike, and Tim.

3.4. Approach: Written by Hosung. Edited by Julieth.

3.4.1. Research: Written by Julieth. Edited by Jason.

3.4.2. Design Phase: Written by Mike. Edited by Julieth.

3.4.3. Integration of Glove and Arm Brace: Written by Mike. Edited by Mak.

3.4.4. Prototyping and Testing: Written by Mike. Edited by Tim and Ritesh

4. Design Iterations

4.1. Mechanical System: Written and Edited by Mak, Mike, and Tim.

4.2. Electrical System Architecture: Written and Edited by Julieth, Jason, Hosung, and Ritesh.

4.3. Communication Protocols: Written by Jason. Edited by Julieth

5. Final Design

5.1. Mechanical Subsystem: Written and Edited by Mak, Mike, and Tim.

5.2. Electrical Subsystem: Written and Edited by Julieth, Jason, Hosung, and Ritesh.

5.3. Software Architecture: Written by Jason. Edited by Julieth.

6. Results and Discussion

6.1. Mechanical Subsystem: Written and Edited by Mak, Mike, and Tim.

6.2. Electrical Subsystem: Written and Edited by Julieth, Jason, Hosung, and Ritesh.

7. Future Work and Recommendations: Entire team's contribution.

References: Formatted by Julieth and Jason.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1. INTRODUCTION

According to the American Stroke Association (ASA), stroke is the leading cause of long-term disability in the United States, with one stroke occurring every forty seconds [3]. Up to 85% of stroke survivors suffer from upper-limb hemiparesis, which is a muscle weakness affecting one side of the body. Hemiparesis causes the majority of post-stroke patients to experience limited dexterity, sensitivity, strength, and coordination in their affected upper extremity. Due to this ensuing muscle weakness that severely affects their motor function, a great number of hemiparetic stroke survivors seek professional rehabilitative and/or assistive help. This help is often provided in the form of conventional rehabilitation or physical care. Unfortunately, the weekly frequency of such programs is strictly limited by the coverage most insurance companies provide. This limitation partially accounts for the low 5-20% complete functional recovery of stroke patients after six months of rehabilitation, with these recovery rates declining as the duration between the onset of stroke and the beginning of rehabilitation increases [4].

The conventional rehabilitation process for upper-limb impaired patients begins with the patient assessment conducted by the physical therapist in order to determine their level of pain, severity of the brain injury, dexterity, and motor functionality. The training process often continues with a combination of physical and verbal cues on how to execute proper arm movements as patients try to complete certain activities such as getting themselves dressed, getting in and out of bed, and performing many more daily tasks. This task becomes a

strenuous job for the occupational therapist, who has to visually monitor and manually guide the patient's movements.

Traditional upper limb training depends largely on the physical therapist's skills and experience. Since the patient's progress is not recorded in a quantitative manner, it becomes difficult to study the therapeutic results achieved through the different stages of the motor training. The use of robotic systems to assist upper-limb rehabilitation, however, offers possible improvements to these current methods. Another issue is that progress in rehabilitation is considerably proportional to the duration of the therapy. The longer patients are exposed to intensive, challenging, and highly repetitive therapy, the greater their chances to improve [4], [5], [6]. Having robotic systems at home that could be utilized for therapeutic purposes may increase the continuation of care outside a primary health setting and potentially enhance the patients' recovery. With this in mind, our project aims to develop a myoelectric-controlled exomuscular sleeve to be used by upper-limb impaired stroke survivors for rehabilitative and assistive purposes in both clinical and domestic settings.

A robotic glove system was designed as part of an MQP at WPI in 2012 by Delph *et al.* [1], which used cable arrays starting at the fingertips and terminating at servomotors placed in a backpack, to actuate finger movements. This device targeted the rehabilitation of the hand through both assistive forces to complement the user's attempt to flex/extend the fingers and resistive forces to emphasize the stability and coordination of the movements. A soft robotic exomuscular brace that covered the lower arm, elbow and upper arm, was developed in 2013 by Brauckmann *et al.* [2] to serve as an assistive device to upper-limb impaired patients. This robotic brace emphasized flexion and extension of the elbow.

The members of this 2014 MQP aimed to integrate these two designs into a unified system that provided flexion and extension of the fingers and elbow, as well as pronation and supination of the wrist, making it a viable device to be used by upper-limb hemiparetic stroke survivors in both clinical and domestic therapy settings. The proposed system was allow for future implementation of myoelectric control and to be used in the rehabilitation of upper-limb hemiparetic stroke patients. Cable actuation through precision DC gear motors was selected as the mechanism to facilitate the desired motions. The system was designed to facilitate the integration of sEMG electrodes to be placed on the patient's arm, which would serve as the main control source to assist with the therapeutic training. Additional sensor information obtained from flex and pressure sensors located on the fingers and wrist, as well as series elastic actuators placed on the elbow, was also intended to serve as indicators of exercise completion and safety regulators.

2. BACKGROUND

2.1 Biomechanics of the Upper Extremity

2.1.1 Lower Arm, Elbow and Upper Arm

The upper arm, i.e. the section of the arm between shoulder joints and elbow joints, is constituted by the humerus bone and the majority of the muscles responsible for flexion and extension of the elbow. The elbow consists of three main articulations, the humero-ulnar (HU), humero-radial (HR), and proximal radio-ulnar (RU) [7]. These articulations are the ones that grant the elbow its two active degrees of freedom: flexion/extension, and pronation/supination (see **Figure 2.1.1**). Flexion/extension and pronation/supination are the motions many exomuscular, upper-limb devices emphasize, and the ones implemented by the system described in this report. Flexion is the motion described by decreasing angles between two bones in a joint. Extension occurs when these angles increase. The humero-ulnar joint, i.e. where the humerus meets the ulna, is the joint that allows flexion and extension of the elbow. The muscles responsible for flexion of the elbow are the biceps brachii, brachialis and brachioradialis, while the muscles responsible for elbow extension are the anconeus and triceps brachii. The range of motion associated with elbow flexion-extension is 150 degrees [7]. Exceeding this range of motion could damage the humero-ulnar joint.

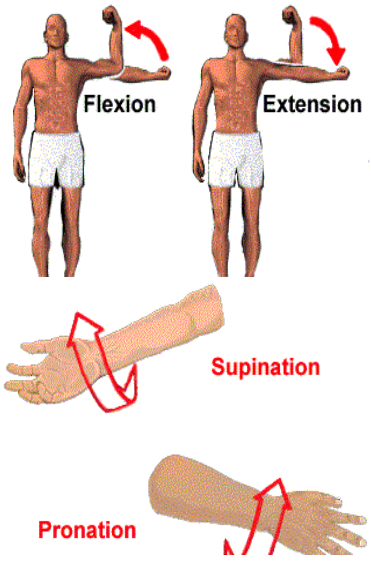


Figure 2.1.1: Elbow Motions from Top to Bottom:

- Flexion/Extension
- Pronation/Supination

Image retrieved from: www.teachmeanatomy.com 1/1/5

The radio-ulnar joint is where the head of the radius meets the ulna. This joint allows pronation and supination of the forearm. The muscles responsible for elbow pronation are the pronator teres and the brachioradialis. The muscles responsible for supination at the elbow are the supinator, biceps brachii and brachioradialis. The range of motion for pronation and supination of the elbow is from -70 to 70 degrees [8]. The section of arm that is between the wrist and elbow joints is the lower arm and it is composed of the radius, the ulna and the muscles that are responsible for motion of the forearm, wrist and fingers.

2.1.2 Wrist

The wrist is one of the three critical focus areas for upper-extremity rehabilitation of stroke patients. Fine wrist control is necessary for completing tasks that require precision, such as getting dressed or serving food. For tasks that require larger forces, it is important that the wrist be able to lock its position. In order to develop a mechanism to aid in the rehabilitation of the wrist, the biomechanics of the wrist must first be understood. One of the primary concerns is the typical flexion and extension range of motion. Taylor and Schwarz [9] measured an average of 44 degrees for flexion and 78 degrees for extension. Other equally important characteristics of the wrist are its ability to support weight and its maximum force output. Schweizer *et al.* [10] reported moments between 22.6 and 19.1 Newton-meters from the neutral position for wrist flexion and extension, respectively. However, the amount of weight supported by the wrist at different positions varies considerably.

2.1.3 Fingers

The hand is one of the most severely impacted extremities for stroke patients because its functional recovery involves the integration of muscle activity from the shoulders to the fingers. The fingers are required for grasping objects and fine manipulation of items, such as pens. Again, the two most important attributes of the fingers that must be considered are range of motion and force output. MATLAB models for the hand, such as the one developed by Malvezzi *et al.* [11], have served as a method for discovering how each of the fingers might move. Furthermore, studies have demonstrated that in a neutral position, the fingers can exert an excess of 66.7 Newtons of force [9]. In more extreme cases, however, fingers can support up to 180 Newtons of force for shorter periods of time [10].

2.2 Muscle Weakness and Motion Impairment

Stroke has recently been named the leading cause of physical disability in adults over the age of 45 [12]. Reduced capabilities in upper and lower limbs are recognized as the most common impairments of stroke survivors. Impairments typically affect one side of the body more severely, and opposite to the affected side of the brain. About 80% of stroke survivors suffer from a "one-side weakness" called hemiparesis. The effects of hemiparesis reduce the victims' ability to move their extremities such as their arms and legs. This deficit makes performing daily tasks difficult and amplifies the need for rehabilitative innovation in this area [13].

The severity of these impairments varies among patients proportionally to the severity of the stroke. Common symptoms include reduced strength and coordination of the affected

limbs, and can extend to reduced sensation and proprioception as well. Such symptoms can be caused by any combination of muscular weakening, nerve degeneration, or reduced cognition interfering with the brain's ability to utilize affected limbs for performing tasks.

A study conducted in 2003 by McCrea *et al.* [14] at the Canadian Institute of Health Research evaluating muscle performance in stroke victims discovered that the effects of stroke impairment were more severe than initially thought. The study found that not only was peak muscle force affected, but the times to achieve and alleviate peak torque were lengthened, i.e. it took stroke victims longer to fully grip and fully let go with the affected arm. Furthermore, these qualities were also reduced in the opposing arm as well, although with less severity.

The four main motor functions in the arm that are affected by hemiparesis are the flexion/extension of the elbow, extension and rotation of the wrist, and extension of the fingers. Wrist and finger extension are the hardest muscle movements to regain [15]. Additionally, muscle weakness of the hand can be further complicated when analyzing isolated finger movements as opposed to congruent finger movements such as the whole hand grasping an object. A common finding among stroke patients has been that their original grip strength can recover to a satisfactory degree, but they oftentimes lack the ability to independently move their fingers [16].

The elbow joint on the human body experiences higher torques than that of the wrist or fingers due to the weight and anatomical position of the forearm and hand. Additionally, an external load is placed on the elbow joint when an individual is carrying an object due to the lever arm nature of the forearm. Past research has revealed that the muscles that actuate the elbow joint have selective weakness in post-stroke patients. In particular, the elbow flexors

and extensors are weaker when the muscle fibers are in their shortened range, which reduces the working range of the elbow joint. Selective weakness in the elbow flexors and extensors affects the rehabilitation approaches for the elbow joint and has significant bearing on the rehabilitation of other stroke-affected muscle groups [17].

2.3 Conventional Versus Robot-Assisted Rehabilitation

An economic analysis of robot-assisted rehabilitation using existing systems compared to traditional physical therapy conducted by Wagner *et al.* [18] revealed that (excluding intervention costs) the total healthcare expenses at the end of 36 weeks averaged $12,679 for robot-assisted rehabilitation as opposed to $19,098 for the usual physical therapy. The incorporation of robotic systems in clinical settings can also increase the cost savings for patients since therapists can treat more than one subject simultaneously, which can in turn maximize the time and intensity of the therapy sessions.

Robotic rehabilitation enables the incorporation of different modes of therapy that are otherwise unrealizable through conventional methods, such as the use of precise force patterns that can be easily repeated by patients over time [19] (an overview of different upper-limb robotic systems is provided in **Section 2.5**). These methods can have great potential for performing patient assessment to obtain accurate measurements of motor functionality of the impaired limb, as well as for precisely monitoring and evaluating patient progress through the rehabilitation process. The aforementioned features of robotic systems can also help therapists determine the effectiveness of different therapeutic methods utilized. Furthermore, these systems can be used to predict patient progression over time and generate a multi-sessions plan for each individual user based on the previously recorded data. The sensors included in

robotic rehabilitation devices can produce precise measurements of specific aspects of motor function, such as spasticity, strength, coordination, smoothness of movement, the ability to perform continuous movements, as well as the number of self-corrections exercised along specific paths [20], [21].

A key component of typical physical therapy offered to post-stroke patients requires therapists to apply certain force using their sense of touch to guide and/or correct the motions of the patients. While therapists may become greatly skilled at this task over time, the method does not provide the same level of precision that a robotic system could offer. In this type of environment, robotic devices could also introduce more sophisticated forms of feedback that could not be accomplished in traditional therapeutic settings. In fact, several studies have demonstrated that robot-assisted therapy can increase treatment compliance by exposing patients to more interactive incentives such as computerized games [4]. By easily introducing new motion constraints, these computerized devices can enhance the complexity of the different motor tasks to be learned. Likewise, robotic devices can be of great utility to provide physical therapy to patients in all stages of recovery, including the chronic population, which is often not treated systematically since the conventional post-stroke rehabilitation settings are mostly intended for patients in the acute stages.

Studies conducted by Burgard *et al.* [22] have demonstrated successful clinical outcomes of robotic rehabilitation in assisting paretic arm movements when compared to conventional therapy techniques. In these studies, the group of stroke patients exposed to robot-assisted therapy exhibited greater improvements in motor function and strength in comparison to the control group. The clinical outcomes of robot-assisted therapy also suggest

promising post treatment results, specifically in the patients' ability to voluntarily reach towards targets. All of the numerous motivations aforementioned have led to escalating advances in the field of robotic rehabilitation in recent years.

2.4 Upper-Extremity Robotic Rehabilitation Methods

The major goal of post-stroke rehabilitation is to put into practice methods that motivate neural plasticity and recovery [19]. The integration of robotics into the post-stroke rehabilitation field offers great possibilities to accomplish these goals through either assistive or purely rehabilitative devices. These robotic systems can not only help patients complete various activities of daily living, such as grabbing a glass of water, but also provide different levels of physical therapy by challenging patients to produce force, control movement, or coordinate reaching motions.

The rehabilitation methods employed in upper-extremity robotic systems can vary greatly from one device to another. While some of the devices allow unrestricted motions of the shoulder and elbow joints for the completion of reaching movements in the horizontal plane, others aim to target the reaching direction along a linear path [23] [24]. A broad generalization of the different types of robotic rehabilitation methods employed by most current systems yields the following four categories: massed practice, bilateral training, fine motor movement, and feedback distortion [19].

Massed practice methods in robotic therapy refer to repetitive movement training of explicit learning patterns. The underlying idea behind these methods is that repetitive training of the impaired upper-limb over an elongated period of time can promote cortical reorganization and induce frequent use of the hemiparetic arm in daily activities [19]. The vast

majority of the upper-extremity rehabilitation devices that utilize these methods encourage gross motor improvements in reaching motions. These robotic systems can be either assistive or purely rehabilitative. The majority of purely rehabilitative devices utilize a combination of both assistive and resistive forces. Assistive forces are provided when the patients are not able to complete target movements by themselves. Resistive forces, on the other hand, are provided to counteract or constrain the patient's direction of movement.

Bilateral Movement Training (BMT) is an alternative approach employed by many rehabilitative robotic devices, which involves repetitive practice of symmetrical upper-limb movements. The purpose of this method is to stimulate the functional recovery of impaired limbs. This stimulation is done by activating the intact hemisphere in an attempt to facilitate the activation of the damaged hemisphere, thus promoting neural plasticity through better control of the movements involving the impaired limb [25]. Bilateral methods are also employed by some robotic devices to stimulate the training of distal arm movements, such as elbow pronation and supination as well as wrist flexion and extension [8], [26].

Fine motor movement methods specifically target rehabilitation of the hand and wrist. The goal of these methods can vary from improving range of motion, velocity and force capacity to increasing coordination during functional tasks [19]. Some upper-limb robotic rehabilitation devices using these methods promote coordinated movements of the hand as well as independent movement of each finger. In studies conducted by Boian *et al.* [27], subjects exposed to these methods were able to demonstrate an increased range of motion and velocity of their finger movements. Other similar devices that employ these fine motor

movement methods focus on recovering hand-gripping motions along with pronation and supination of the forearm.

The feedback distortion techniques employed by robotic systems used in upper extremity rehabilitation consist mainly of force perturbation and/or visual error augmentation [19]. Force feedback is mainly utilized to amplify movement errors generated by the patients when completing specific reaching exercises. Studies conducted by Patton *et al.* [28], [29] employed this error magnification paradigm by applying perturbing forces to the impaired limb in the opposite direction to specific reaching trajectories for a period of time. The subjects of this study counteracted the perturbing force in order to continue along the path of their desired reaching trajectories. A neuromuscular after-effect was noted when the perturbing forces were unexpectedly removed and the subjects were able to move in the desired trajectory. Another relevant result of these studies indicated that the training after-effects persisted for a longer period of time in post-stroke patients compared to healthy control subjects, thus indicating that error amplification techniques used in rehabilitative training may result in greater strength recovery.

Feedback distortion is also commonly employed by many upper-extremity robotic rehabilitation systems in the form of visual feedback displayed on computer screens. This technique is suitable during rehabilitation of chronic post-stroke patients who may not be willing to perform reaching movements beyond the limits of their comfortable range of motion [19]. Studies conducted by Brewer *et al.* [30] revealed that chronic post-stroke patients are able to follow visual distortion to great levels of performance. Subjects in these studies demonstrated functional improvements of their pincer grasp abilities. Furthermore, studies

conducted by Wei *et al.* [31] showed that subjects exposed to visual error augmentation demonstrated an accelerated learning process during training.

2.5 Existing Upper-Extremity Rehabilitative and Assistive Devices

Advances in the field of rehabilitation robotics have led to the development of many robotic systems used in physical assistance and rehabilitation in both clinical and domestic settings. Devices that have been developed specifically for upper-extremity treatment make use of the aforementioned robotic rehabilitation methods in an attempt to improve strength, coordination, spasticity, motor functionality, range of motion, and smoothness of reaching movements. For instance, MIT-Manus (see **Figure 2.5.1**) utilizes massed practice and force feedback methods to provide therapy that targets reaching motions towards an endpoint. The force feedback provided by this device consists of forces applied in the same direction as the reaching motion to assist muscles in task completion. Studies conducted to evaluate the effectiveness of robot-assisted therapy with the MIT-Manus revealed that this system can improve the clinical outcomes for repetitive, goal-directed therapy [32], [33], [34], [35], as well as improve the motor and

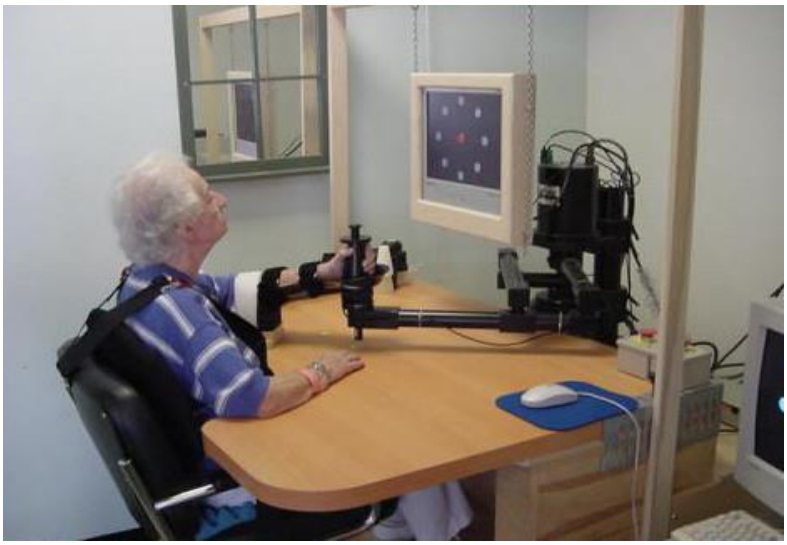


Figure 2.5.1: MIT Manus.

Image retrieved from:
www.jneuroengrehab.com/content/1/1/5

functional recovery gains in subjects with acute [32] and chronic hemiparesis [36], [37], [38], [39].

Another robotic system designed for upper-extremity rehabilitation through massed practice methods is the Assisted Rehabilitation Measurement (ARM) Guide (see **Figure 2.5.2**). This device emphasizes repetition of reaching motions along a linear track with assistive or resistive forces. Studies conducted by Kahn *et al.* [40] did not reveal significant differences between results obtained with robotic assistive and resistive forces employed by this device and the free, unassisted methods used in conventional therapy for the training of reaching motions.

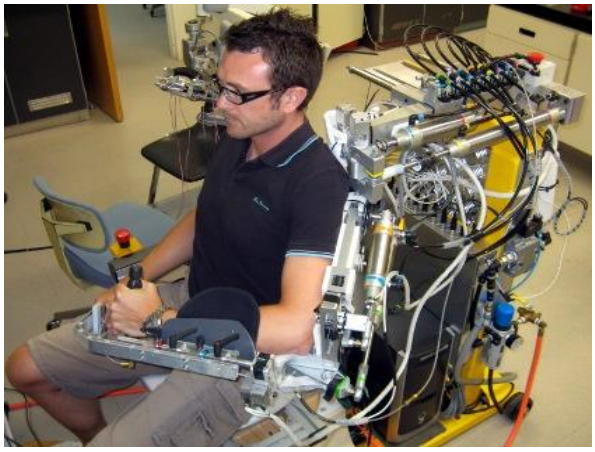


Figure 2.5.2: Assisted Rehabilitation Measurement (ARM) Guide.

Image retrieved from: http://biorobotics.eng.uci.edu/armrehab

The results obtained when evaluating massed practice devices, such as the ARM Guide, in terms of skill retention over time, suggested the implementation of other methods, such as bilateral training, in the development of new robotic devices for upper-extremity rehabilitation. The Mirror Image Motion Enabler (MIME) (see **Figure 2.5.3**) was the

Figure 2.5.3: Mirror Image Movement Enabler (MIME).

Image reprinted from: Lum *et al.* [42]

first robotic rehabilitation system that explored bilateral training for upper-extremity stroke rehabilitation, focusing specifically on the practice of reaching motions. This device is coupled to the user's unimpaired arm, allowing three-dimensional motions that are assisted or resisted through force feedback. Studies conducted by Lum *et al.* revealed that both chronic [41] and subacute [42], [43] stroke patients treated with this device demonstrated increases in reaching, strength and proximal movement. According to both studies, certain aspects of upper-extremity recovery appear to occur at a faster rate for patients using MIME during the initial period of the rehabilitation process. The long-term effects, however, do not seem to be significantly different than the ones obtained through conventional therapy methods. A follow-up research conducted in 2004 [44] with this same system evaluated the effectiveness of utilizing surface electromyogram (sEMG) for improved muscle activation patterns from patients with post-stroke hemiparesis. These studies revealed promising results showing consistency with related pre-existing research [45], [46] on training patients toward motor recovery. These outcomes suggested the potential incorporation of sEMG in this particular device.

Another robotic system that uses bilateral training for upper-extremity stroke rehabilitation is the Bi-Manu-Track (see **Figure 2.5.4**). This device focuses specifically on flexion and extension of the wrist, as well as forearm pronation and supination. In a study conducted by Hesse *et al.* [47], a large

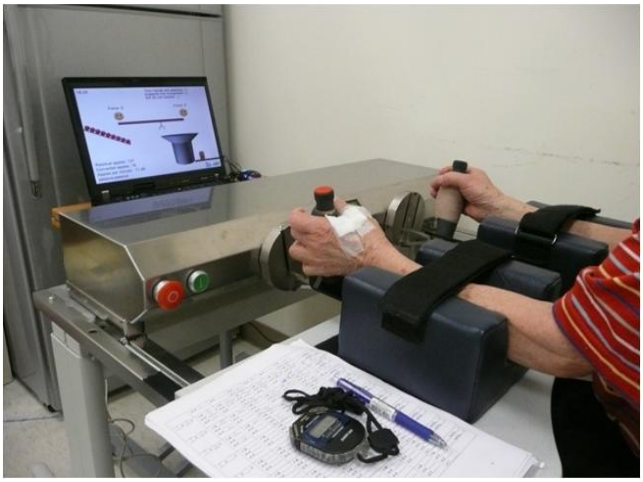


Figure 2.5.4: Bi-Manu-Track.

Image retrieved from:
http://nursing.stanbridge.edu/?p=19334

number of acute stroke patients who were treated with this device demonstrated a greater improvement in the Fugl-Meyer[1] scale [48] (for over three months after the treatment) than the other subjects who were treated with electrical stimulation as commonly done with conventional therapy.

Figure 2.5.5: SaboFlex System.

Image retrieved from:
http://www.saebo.com/products/saeboflex/

Some of the robotic systems developed for upper-extremity rehabilitation take the form of compact, wearable devices, such as gloves, to minimize stress on the hand, fingers, and joints during therapy [49]. One of these wearable gloves is the SaeboFlex (see **Figure 2.5.5**), which is a form of unpowered orthotic that emphasizes training of the finger, hand, and wrist. This device uses adjustable springs to assist, resist, or stabilize the movement of fingers during therapeutic exercises. Studies conducted by Farell *et al.* [50] revealed that use of this device could significantly improve wrist extension but not flexion. The results did not demonstrate improvements in finger flexion and/or extension.

Another wearable device developed for rehabilitation of the hand and wrist is the Hand Tutor (see **Figure 2.5.6**). This ergonomic glove, which comes with software dedicated for physical therapy rehabilitation, emphasizes recovery of speed, as well as passive and active

[1] Scale for assessment of physical performance regarding: motor, balance, sensation, joint range of motion, and pain [48].

range of motion of the wrist and fingers. The device tracks the patient's hand motions during game exercises and provides feedback that serves to improve their motor functionality [51].

Figure 2.5.6: Hand Tutor.

Image retrieved from:
http://www.fysiomed.cz/eng/medicaltechnologies/biofeedback/

2.6 Previous Work Leading to this Project

2.6.1 Rehabilitative Robotic Glove

The Rehabilitative Robotic Glove (see **Figure 2.6.1**) developed by Delph *et al.* [1] at WPI successfully demonstrated the feasibility of actuating the hand through the use of cables and servomotors. In addition to this accomplishment, the team developed a method for gathering, filtering, and processing sEMG data. Although never fully implemented, the data collected through sEMG could then be used to control the hand's extension or flexion, allowing the user to grasp objects. In addition, the device could also be controlled with switches or programmed positions.

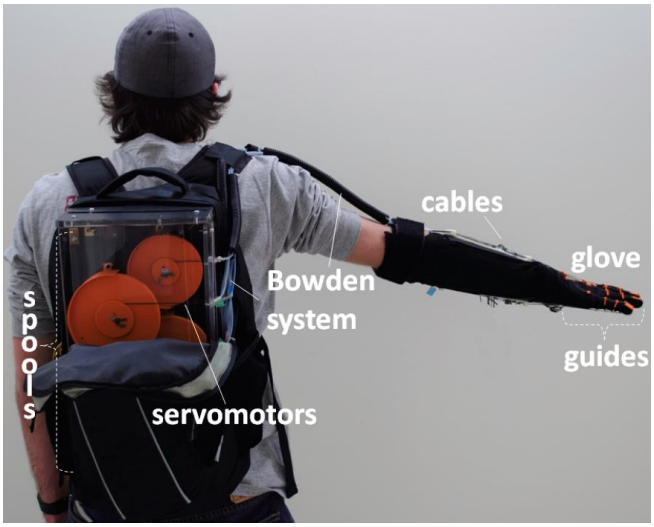


Figure 2.6.1: Rehabilitative Robotic Glove [1]

The mechanical construction of the glove itself included a collection of attachment points at the ends of each of the fingers and a series of cable guides for the Kevlar cables used to actuate the hand. In order to compensate for the varying dimensions of users, a tensioning system for the cables was placed on the user's forearm. This system allowed the effective length of cables actuating the hand to be reduced. Finally, through the use of polyethylene surgical tubing, the team was able to create a Bowden cable system that effectively allowed them to select any location for the servomotors and spools.

The servomotors were controlled with a combination of pulse-width modulation (PWM) signals and current limiters. The PWM signals were used to control the positions of the servomotors, while the current limiters were used to control each of the servomotors' torque outputs. These servomotors were attached to spools that had two different diameters to account for the different amount of cable required to extend and flex the fingers.

Two bipolar electrode-amplifiers were used to detect hand extension and flexion respectively. Another electrode was placed over the bony part of the elbow to serve as a reference for the two electrode-amplifiers. The signal was first amplified at the electrode-

amplifiers then filtered through an analog high pass filter and an analog low pass filter. The signal was sampled at a rate of 2 kHz with a MSP430 microcontroller. Finally, the sample data were digitally filtered. A circular buffer was used to calculate the signal moving average absolute value, which was then used to determine the user's intent on extending or flexing the hand.

2.6.2 Soft Robotic Exo-Muscular Arm Brace

The Soft Robotic Exo-Muscular Arm Brace (see **Figure 2.6.2**) developed by Brauckmann *et al.* [2] at WPI served as a platform for testing different actuation methods, experimenting with varying exoskeletal designs for the elbow, and improving sEMG data acquisition and processing. Similar to the Rehabilitative Robot Glove project, sEMG data were collected; however, the data were never used to actuate the device.

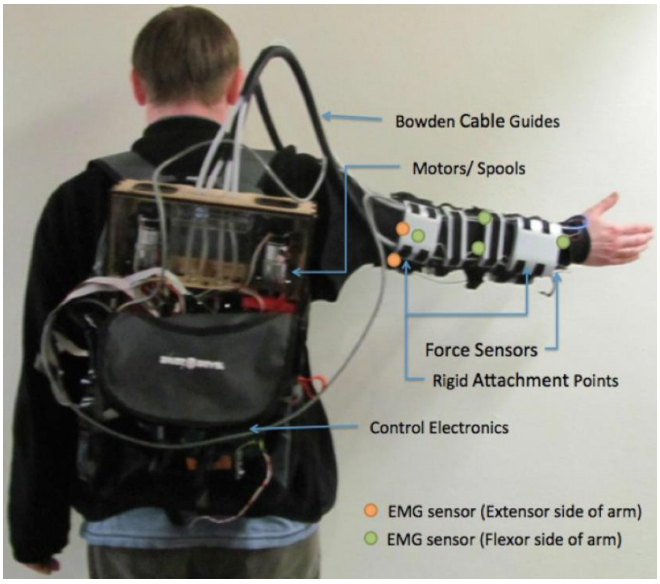


Figure 2.6.2: Soft Robotic Exo-Muscular Arm Brace [2]

3. PROJECT STRATEGY

3.1 Client Statement

The purpose of this project was to develop an exomuscular sleeve to be used by upper-limb impaired post-stroke patients for rehabilitation and assistive purposes. The device emphasizes flexion and extension of the fingers and elbow, as well as pronation and supination of the wrist. The client statement, as developed by the design team reads as follows:

> Redesign the soft robotic glove [1] and the exomuscular arm brace [2] developed by previous MQP teams, along with a newly developed wrist rotation structure into a unified system for upper-extremity hemiparetic stroke survivor therapy that is viable for use in both clinical and home settings.

3.2 Primary Objectives

The primary objectives taken into account for the development of this robotic system were to design a device that would be safe, effective, lightweight, adjustable, intuitive, affordable, and portable. The order of these objectives were determined using a pairwise comparison chart. These objectives were ranked as follows:

1. Safe
2. Effective
3. Lightweight
4. Adjustable
5. Ease of use
6. Affordable
7. Portable

This ranking was used as a set of dominant guidelines for the project, and their ranked priority helped to guide the many design iterations in a direction most beneficial to the project.

Ensuring the user's safety was the most crucial characteristic the device had to meet to avoid exposing the user to any form of physical harm. Such safety features as tracking cable displacement and force-limiting clutches to account for system failures, were considered in the design of this system. A safe system utilizing these features would give the operator complete control over the system's operations, while not exposing the user to potential injury or undue discomfort, during normal operations or in a failure scenario. An effective device would be one suitable for both rehabilitative and assistive purposes during the completion of specific motor tasks assigned to the patient, or in daily life. Since the device was intended to be used by stroke survivors on a daily basis it was important to make it lightweight. By restricting all powered parts to a separate platform, and by putting as few components on the arm as possible, the arm could be kept sufficiently lightweight as to not be burdensome on the user moving their arm, thus ensuring effectiveness as a mobile device and the user's comfort.

The adjustable capabilities of the system, i.e. ensuring that the device could be worn by users with different anatomical measurements and at different stages of the rehabilitation, were also taken into consideration in an attempt to reach out to a greater user population. Furthermore, designing an intuitive computer interface that could be easily used by both post-stroke patients and physical therapists would endorse the proper operation and application of the different therapeutic features, as well as the marketable properties of the device. Developing an affordable system was also considered throughout the design process in order to

ensure the minimum possible cost of rehabilitation when compared to conventional post-stroke physical therapy. A final purchasing and operating cost lower than that of a comparable regimen of physical therapy would certainly enlarge the number of stroke survivors in need of rehabilitation and/or assistance that could benefit from this robotic device. Lastly, producing a portable system that could be used in domestic settings or even by wheelchair users was an ideal objective to accomplish upon the completion of this design project.

3.3 Project Constraints

The constraints for the design were directly derived from the goals in the client statement, but limited in scope due to the timeline of the project. For the device to be considered an effective rehabilitative tool, it must be ergonomically adjustable to minimize the patient's discomfort during daily use. Since the device was intended to include portability features, the entire apparatus must be independently powered and considerably lightweight to facilitate its continuous use. Designing an assistive system, one which could supplement its user's own limited strength, would impact the overall portability of the device. Regarding the force requirements and range of motion, the device would meet assistive requirements, but the prototype size would compromise its portability. Therefore, the device would provide aid for the completion of daily activities and demonstrate its proof-of-concept as an assistive portable device. True portability, such as what would be expected in a final, marketable design would be a matter of time and research costs, which are unfortunately beyond the scope of this project timeline.

Temperature was another important constraint that arose from the need to guarantee the patient's safety when using this device. In order to avoid skin irritation and temperature-

related soreness, this sleeve could not expose the user's upper extremity to more than 100° F. Though a relatively small concern as the circuit boards and motors would be the only sources of heat on the device, heat dissipation was still very much worth accounting for. Lastly, the manufacturing of this prototype robotic system could not exceed the initial budget awarded to the team from the different academic departments that the student developers were affiliated with. This initial budget consisted of $1,440: $160 per each of the seven students who compose the team. This allowance restricted the parts and materials that could be used for the project.

3.4 Approach

The long-term and most essential goal of this project was to develop a functional exomuscular sleeve that justified its intended use by upper-limb impaired stroke survivors for both rehabilitative and assistive purposes. Mechanically and electrically, the system would apply resistive and assistive forces as part of its rehabilitative attributes, including force limiting and control and associated control algorithms. To ensure user's safety, smart fail-safes in both hardware and software were implemented. In addition, an integrated control system was required for the existing exoskeletal glove and arm brace. The design and development of such system was divided into the following seven stages: research, design phase, integration of glove and arm brace, prototyping/testing, design finalization, and documentation.

3.4.1 Research

Research was conducted to understand the specific needs of post-stroke survivors with upper-extremity hemiparesis, as well as the different stages of the conventional rehabilitation process most of these patients are exposed to. To gather some of that initial information, an occupational therapist was interviewed by the design team. The topics discussed covered the

impairments of stroke in the upper extremity, the average frequency and duration of the treatments, the types of upper-extremity motions stroke survivors tend to lack most, the different exercises that post-stroke patients are instructed to execute during rehabilitation, as well as the advantages and disadvantages of using assistive versus resistive forces for the different rehabilitative exercises. From the information gathered different research topics were formulated and divided among the team members.

3.4.2 Design Phase

By researching existing systems, the project team was able to establish the current state of technology in the field and innovate where necessary. Following the design constraints as set by the client statement, the team continually improved upon previous designs to arrive at a product which met the design constraints stated previously in Section 3.3. For organizational purposes, the group worked in two sub-teams: one mechanical (dealing with the device and its physical components), and one electrical (focusing on the control system and sensing).

The mechanical sub-team assigned each of its three members a specific part of the device to design and improve upon. By continually making changes to designs, bringing them to the sub-team for review and criticism, and returning to make further changes, well refined designs were created for each component on the device. As the project progressed, these revisions began to include physical prototypes to test for fit and functionality of the various components, ensuring the viability of each design.

Similarly, the electrical sub-team divided the development of the different printed circuit boards required for the system as well the software component among its four

members. Sub-team meetings with and without the advisors were scheduled every week to ensure that each member of the electrical sub-team was progressing appropriately in the design and development of the circuit boards. These sub-team meetings also served to confirm that the design of all the electrical components followed a consistent pattern and did not conflict with one another.

Furthermore, full team meetings with and without the advisors were scheduled every week to ensure regular communication among both sub-teams. In these meetings, the sub-teams presented the progress made so far to one another and discussed the pros and cons of the different design iterations from a mechanical, electrical and software point of view, thus facilitating the sustainability of the entire system in development.

3.4.3 Integration of Glove and Arm Brace

In keeping with the iterative nature of the project, this team was one of many working toward a comprehensive rehabilitative system. Previous teams had successfully developed functioning prototypes of two of the subsystems: a glove for the actuation of the fingers, and an arm brace for actuation of the elbow. A significant part of the task at hand was to combine these two devices, along with additional subsystems developed by the team, into a unified product. This complete device featured three subsystems: hand, wrist, and elbow. Operation of the system requires a singular power source, a common actuation system, and a unified control system. This ultimately was lowered in priority for the prototype in favor of successfully demonstrating the functionality of the device, but the prior assessment would still apply for any final product.

In addition to simply combining these previous systems, several additions were to be made as well. In order to increase the awareness of the control system, flex and presure sensors were added to the first knuckle and fingertips of each finger, respectively, and modifications to the glove had to be made in order to accommodate these. Additionally, the components making up the previous elbow brace were to be condensed and reduced in size to reduce weight and increase user comfort. Finally, both the glove and elbow were functionally integrated into a wrist device designed by the mechanical sub-team to allow for active control of pronation and supination of the wrist, and passive support of flexion and extension of the wrist.

This wrist device had to be made to integrate seamlessly into the existing Bowden system established by the other two components, which included the motors used to manipulate the cables. A particular challenge posed by the combination of two previous projects involved combining the two motor backpacks they each utilized independently. In order for the device to meet the portability specification, the final product would need to encompass only one motor backpack containing all the motors necessary to actuate the fingers, elbow and wrist, as well as several circuit boards for the control system.

3.4.4 Prototyping and Testing

Through the iterative design process, solutions were devised, prototyped, and tested. Mechanical components were subjected to tests of strength and durability, ensuring their capabilities and solidifying their design. Electrical control system components were subjected to testing for correct signal processing, data analysis, accurate interpretation of data,

and proper implementation for desired functionality. Safety was considered throughout all testing, and was prioritized while determining the validity of any working designs.

4. DESIGN ITERATIONS

The purpose of this project was to design and develop an exomuscular sleeve to assist post-stroke patients with their rehabilitation. A major requirement for this project was the development of an assistive sleeve that could mimic the natural behavior of human muscles and tendons of the upper limb. In order to accomplish this task, several soft robotics design choices were considered for the support and actuation of the different structures of this exomuscular sleeve, i.e. fingers, wrist, and elbow, to ensure the development of an easily wearable device with an intuitive user interface.

4.1 Mechanical System

4.1.1 System Overview

Figure 4.1.1 shows a high-level sketch of the placement for all of the brace parts on the arm. For actuating the flexion and extension of the elbow, Bowden cables are run similar to the bicep and triceps. For actuating the pronation and supination of the wrist, cables are wound about the forearm between the distal elbow brace and the proximal wrist brace segments. By tensioning the cables wound in the same direction, the twp brace parts are rotated opposite of each other. A flexible metal plate is installed between the proximal and distal wrist brace segments to support the flexion and extension of the wrist. Actuating the flexion and extension of the fingers is achieved in a similar manner as actuating the elbow.

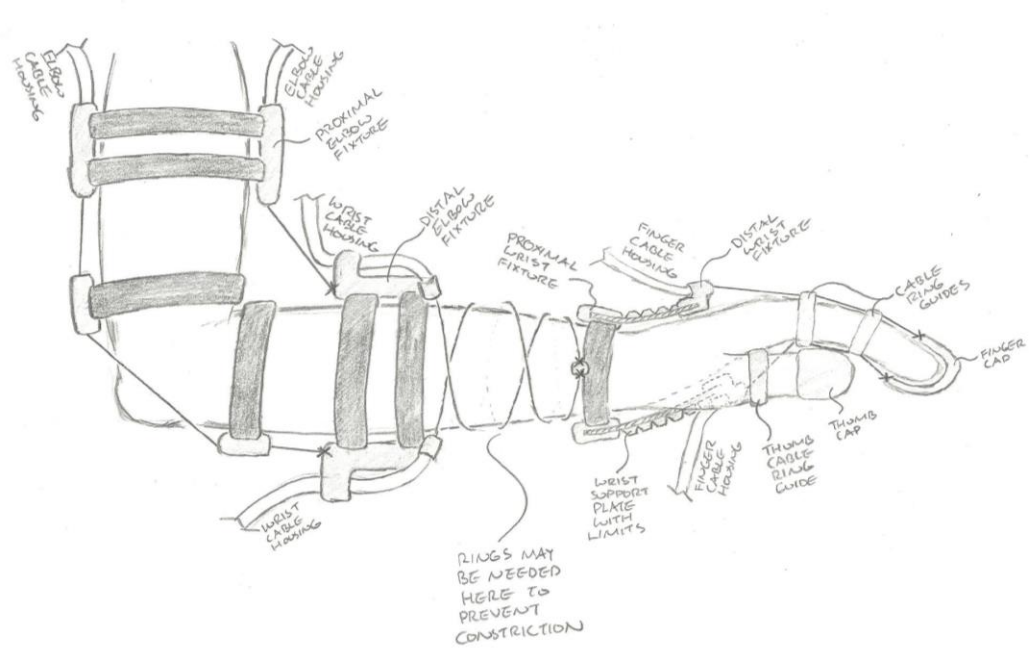


Figure 4.1.1: Concept Art for Brace Placement

Figure 4.1.2 is a more detailed sketch that shows placement of sensors and wire routing with respect to brace parts on the hand. In order to estimate the deflection of the fingers, a flex sensor is placed over the first knuckle of each finger. Pressure sensors are attached to each fingertip to allow for estimation of grip force. In order to allow for easier donning and doffing of the glove, the sensor board and cables are allowed to detach using an interface device placed on the back of the hand.

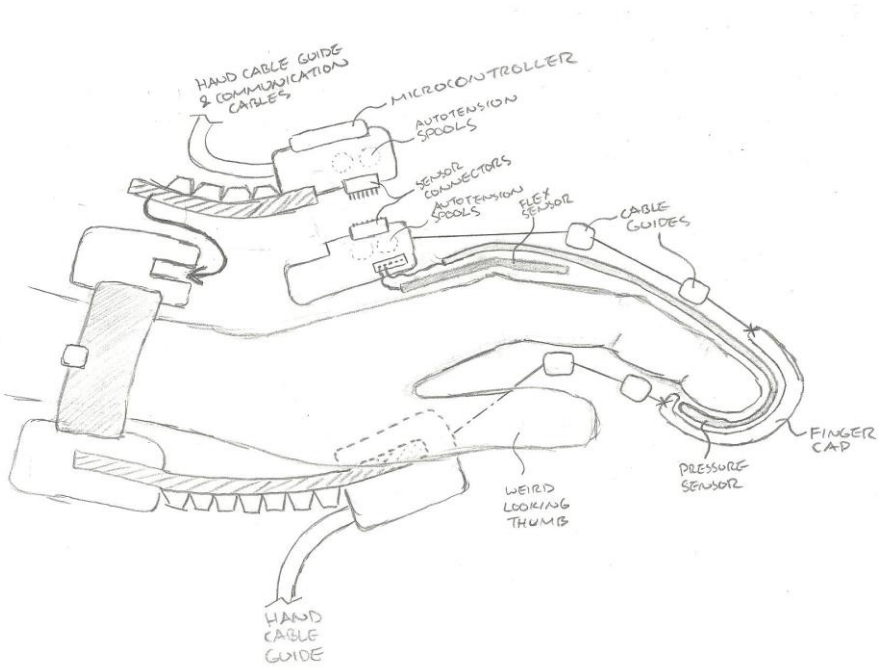


Figure 4.1.2: Sensor Placement on Hand

4.1.2 Cable Drive

Based on the research conducted by the previous two MQP groups [1]- [2] and as suggested by the graduate student advisors, the team decided to continue to use cables strung between the joints and a series of motors to actuate the device. Referred to as a Bowden system, this design is used in many common devices, such as in the brakes on a bicycle, and is a proven way to transmit a force between two points without exerting a force on any components in the middle. This system is accomplished by pulling on an inner cable relative to an outer tube it sits within, causing the force along the length of the system to cancel while the force is experienced at both ends of the cable. In addition, the mechanical team evaluated the mechanical design of the previous two projects and decided to make several modifications based on their research and the requirements set by ranked objectives.

One design change was the system used to tension the cables when fitting the device to a new user. The Rehabilitative Robotic Glove developed by Delph *et al.* [1] used a series of knots and a brace to take up the extra slack in cable to account for people of different size. This technique, however, made the system a little more bulky than necessary, and the brace occupied space that would be necessary for other components of the sleeve. In addition, the system was not easy to adjust. The Soft Robotic Exo-Muscular Arm Brace as developed by Brauckmann *et al.* [2] resolved this issue by using one motor per cable. This configuration allowed each cable to be tensioned individually. However, this method proved to be very bulky, because four motors were required to actuate the elbow instead of one.

The team decided instead that adjustable spools were the simplest way to account for the different lengths of cable required for people of different sizes. This design was accomplished by using two separate spools along a threaded rod that could be rotated independently when in an “open” position. When in a “closed” position, the spools would be locked and could not rotate independently of each other. Initially this system was designed by placing a series of radial triangular gear teeth extending outward from the center of each spool, on the facing sides of each of the two spools. These teeth would allow the two spools to be pressed together and be unable to rotate independently, but loosened to allow the teeth to slip past each other to a new position. The principal is demonstrated below in **Figure 4.1.3**.

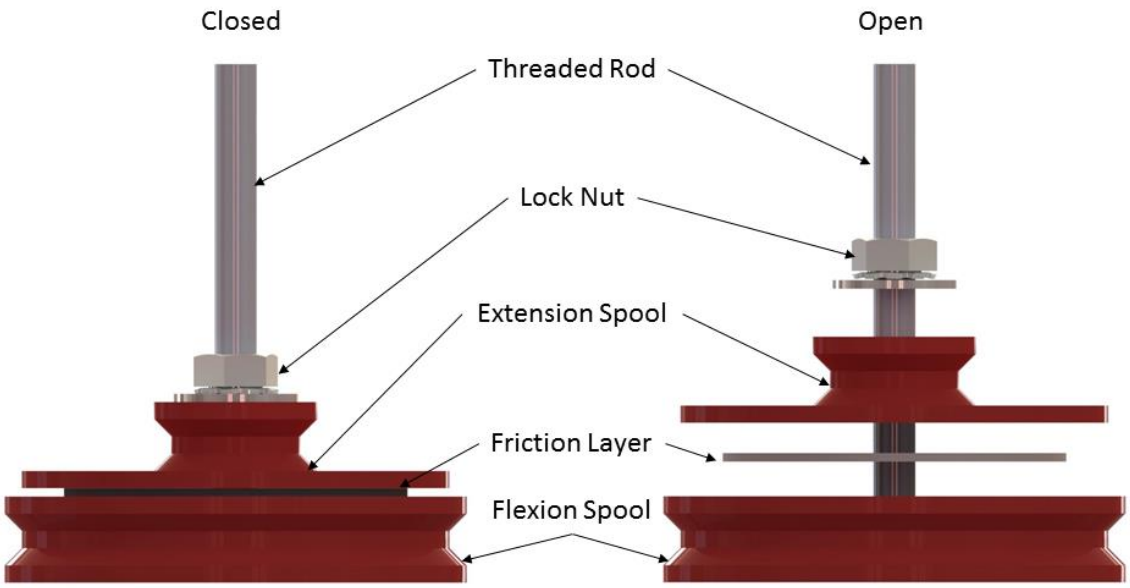


Figure 4.1.3: Open and Closed Spool Configurations

Due to the ratio of cable pulled along the top and bottom of each finger differing when flexing and extending the finger, one spool had to be much larger than the other, on average at a ratio of 2.6:1. This condition caused a problem in manufacturing as the teeth became very small towards the center of each spool. The spools were to be 3D-printed due to the relatively low cost and unique shape of the parts, but even the best 3D-printers have finite precision, and the teeth towards the center were simply too small and tended to blend together into a solid face in the first prototype.

In response to the first prototype failing to meet the design, the method was modified to have no teeth near the center of the spool. Ideally this would fix the problem of the printer to distinguish one tooth from another. Since one spool was so small however, a flange was added to one face of the small spool to match the diameter of the large spool. On these faces a

ring of teeth was added to fill the outermost 0.75". Though the issue seen in the first prototype did not occur, a secondary issue which could not be observed in the first prototype was discovered. Due to the small triangular shape of each tooth, the tip of each tooth tended to be a single extruded line of plastic from the printer. Since the material exits the printer nozzle as almost a liquid, the material tended to slump and deform, which made each tooth a slightly different shape, causing them to not fit together at all. An image of this prototype can be seen below in **Figure 4.1.4.** Only the toothed face is seen in the figure, this was to conserve money and time while prototyping the lock mechanism by not printing the full thickness of the spools every time.

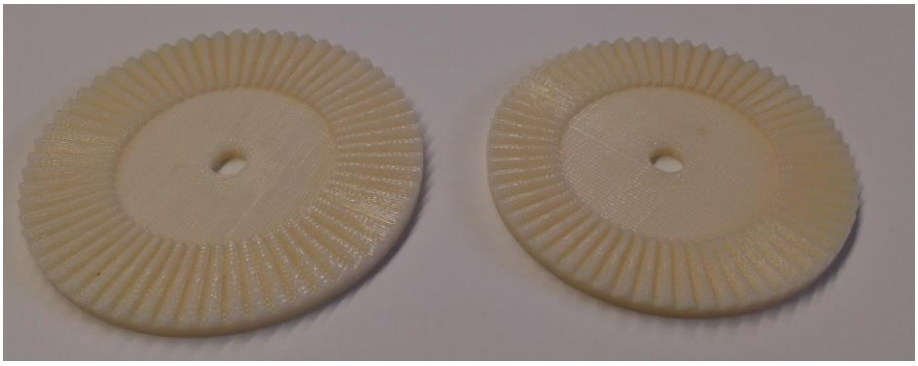


Figure 4.1.4: Toothed Spools

A third prototype was created by removing the top .025" of each tooth with the intention of removing the singular layer material that had caused the problem in the second prototype, while still maintaining the triangular shape of each tooth. This solution was not successful and the same problem persisted. Finally, the toothed design was abandoned all together for the much simpler design utilized in the final design. This design simply featured the two spools being compressed together with a layer of rubber between. This rubber friction layer prevented the spools from sliding with even a modest amount of pressure. The pressure

was applied by a lock nut, with an accompanying washer, on the central threaded rod which was firmly glued into the large flexion spool. By loosening the nut, the spools could be separated and the small extension spool could be rotated on the threaded rod, which it was not glued to. This allowed the slack to be drawn up when fitting the sleeve to a user.

4.1.3 Series Elastics

Following in the footsteps of the Soft Robotic Exo-Muscular Arm Brace project, the team decided to continue the use of series elastic actuators for actuating the elbow and wrist. The Soft Robotic Exo-Muscular Arm Brace employed springs at the cable attachment points to the brace segments to achieve the elasticity necessary for a series elastic actuator. The deflection of the spring was determined using a magnet and magnetic potentiometer. The deflection could then be used to determine the approximate force the cable was applying to the brace segment. The measured force, in turn, was used to ensure that an excess of force and torque was not applied to the user.

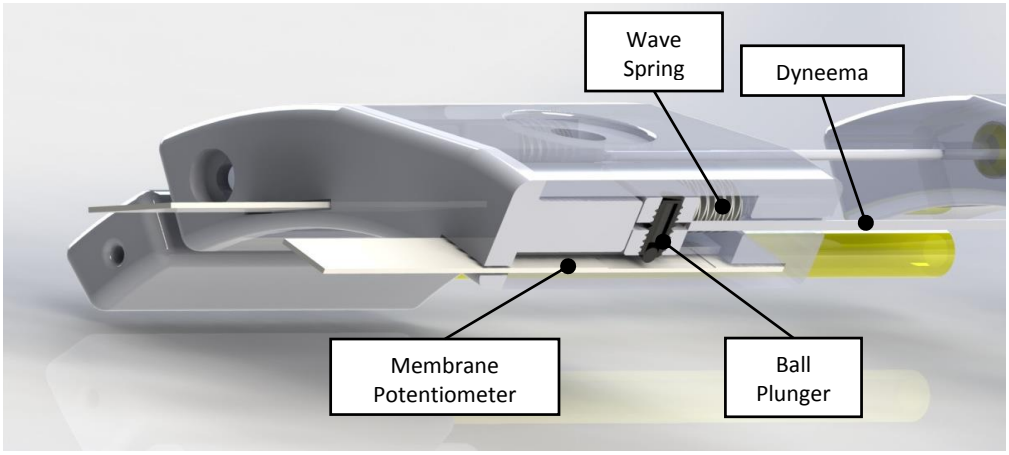


Figure 4.1.5: Ball and Plunger with Membrane Potentiometer Force Sensor

The team decided to use membrane potentiometers and a ball plunger instead of the magnetic potentiometer because of its lower profile as seed in **Figure 4.1.5**. The more compact

design aided in reducing the size of the brace parts, thereby reducing the weight of the overall system and increasing the mobility of the elbow and wrist.

4.1.4 Elbow Brace Design

One of the major concerns of designing the elbow brace parts was slimming down the profile of the braces. The reduced size of the brace parts provides two benefits. First, decreasing the size of the part minimizes the amount of weight on the extremities, thereby reducing the power requirements for actuation. Secondly, and more importantly, decreasing the size of the rigid part increases the ratio of hard to flexible parts, thereby making the brace more conforming for different users. In order to ensure ease of donning and doffing of the brace parts, each of the segments were designed to be attached to a Neoprene sleeve. The flexibility of the Neoprene sleeve allows for some flexibility in final placement of each of the brace parts. The sleeve also allows patients to don and doff all of the arm brace parts at once.

4.1.4.1 Elbow Brace Design from 2012-2013 MQP

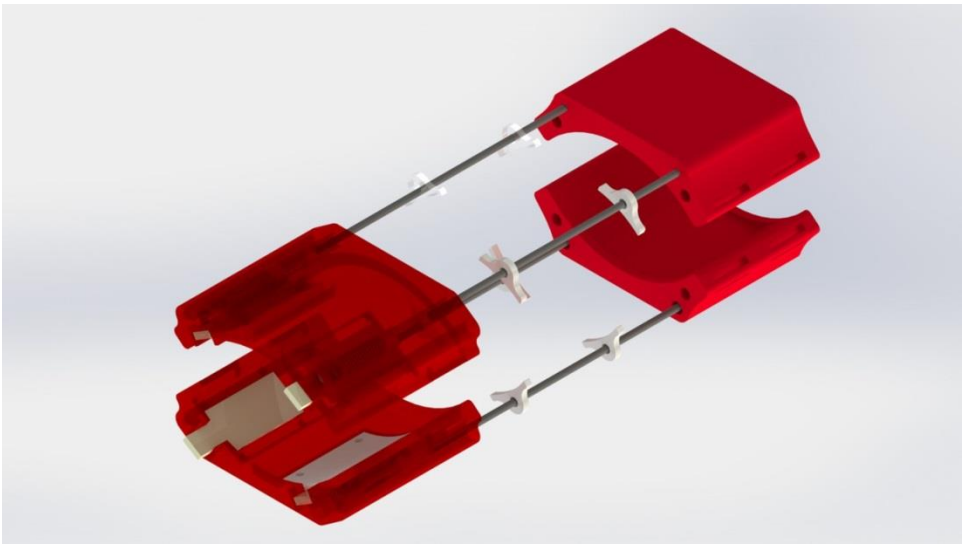


Figure 4.1.6: Elbow Brace Design from 2012-2013 MQP

Figure 4.1.6 shows a rendering for elbow brace parts that was used by Brauckmann *et al.* [2]. In this design, the Kevlar cables pass through the proximal side of the distal brace (left set of red blocks). A knot at the end of the Kevlar cable prevents it from sliding out of the device. The cable can thus be put into tension, pulling back on a cylindrical block with a magnet attached. The cylindrical block then compresses a spring and also actuates a magnetic potentiometer, thus allowing the device to determine the position of the block.

While testing actuation with the assembled device, the team discovered that some of the smaller guides would compress together during arm extension, causing discomfort due to pinching. The team attempted to address this problem by adjusting how the brace parts are attached to the sleeve to prevent the translation of the parts.

4.1.4.2 Elbow Brace Design Revision 1

The first revision of the design from the previous year was an attempt to segment each of the brace parts in order to increase the conformity of the brace.

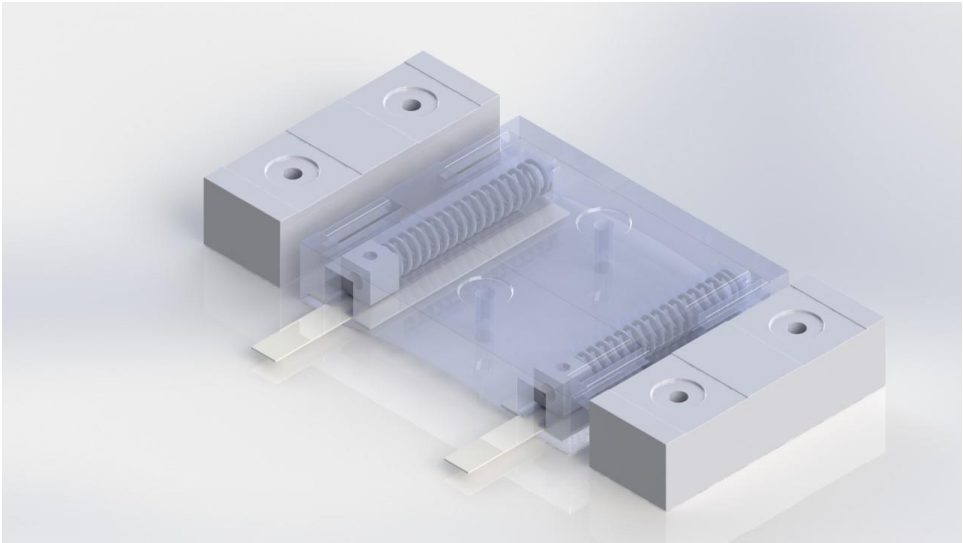


Figure 4.1.7: Elbow Brace Revision 1

This revision, as seen in **Figure 4.1.7**, includes the use of ball plungers and Spectra Symbol SoftPot membrane potentiometers for sensing the force exerted on the cables. The membrane potentiometers offered a lower profile sensor package, while the ball plunger blocks remained approximately the same size as the blocks used to connect the Kevlar cables to the magnets. The additional blocks on the sides were placeholders for Bowden guide endpoints for the cables leading to wrist pronation and supination actuation. The first revision was more compliant than the previous design but also proved much more bulky.

4.1.4.3 Elbow Brace Design Revision 2

The second revision of the elbow brace is lower-profile than the first. Instead of starting with a rectangular extrusion, the main body of the brace is instead an arced surface.

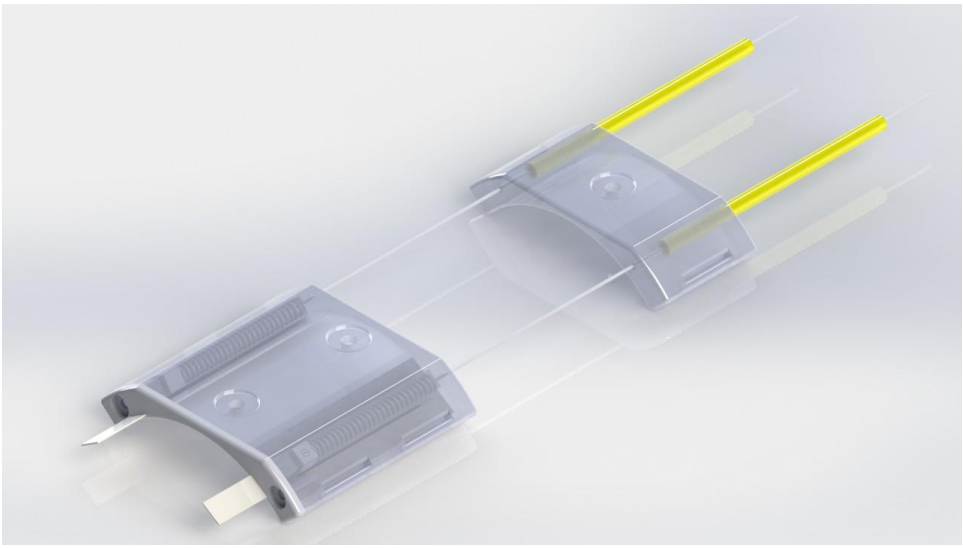


Figure 4.1.8: Elbow Brace Revision 2

As is visible in **Figure 4.1.8**, the basic construction of the brace is quite similar to the model developed by Brauckmann *et al.* [2]. The endpoints of the cables are tied to the ball plungers, which are placed inside small blocks. A square shape was chosen for the channel,

because that prevented the blocks from rotating. Tensioning the cables caused the springs in the lower part of the arm brace to compress; the displacement would be measured by the membrane potentiometers in contact with the ball plungers.

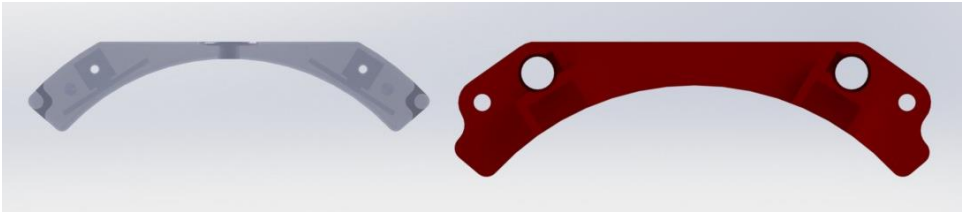


Figure 4.1.9: Comparison of Revision 2 (left) and Original (right) (Front View)

Figure 4.1.9 shows a front view comparison of the second revision of the elbow brace (left) with the original model (right). As is visible, the updated version is slightly thinner and not as wide. It is important to note, however, that the updated version does not have extra holes for the Bowden system for either the hand or wrist actuation. The thickness of the material near the center of the second revision is also too thin. This problem was discovered when the brace part was broken in half by flexing is by hand along the length of the part. Although a metal brace part would withstand the forces of actuation, a plastic version would not.

4.1.4.4 Elbow Brace Design Revision 3

After 3D printing the Revision 2 elbow brace, the team determined that the cables in the brace were placed too far from the center of the brace. The distance between the cables and the center of the brace is problematic when the arm is at full extension. By placing the cables at the extremity of the arcs, the cable tension nearly acts though the elbow's axis of rotation, thereby decreasing the effective torque output of the motors to the cables. Ideally, the cables should be placed as far away from the center of joint rotation.

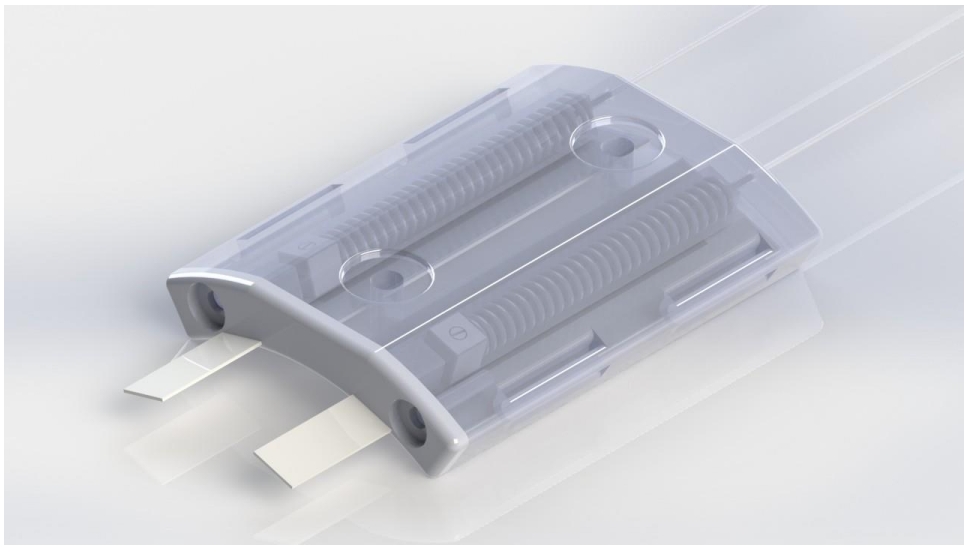


Figure 4.1.10: Elbow Brace Revision 3

Figure 4.1.10 shows a close render of Revision 3 of the elbow brace. The cables are closer to the center of the brace. When the elbow is at full extension, the cables are thus farther away from the center of joint rotation, thereby maximizing the amount of torque the motors can exert on the elbow.

Figure 4.1.11: Front View Comparison of Rev. 3 at the Top, Rev. 2 in the Middle, and Original in the Bottom

Bringing the cables closer to the center of the brace part also thickened the part, thereby increasing its strength near the center. **Figure 4.1.11** shows the third revision compared to the second and original versions of the brace. The third revision is flatter and narrower than the first two parts.

4.1.4.5 Elbow Brace Design Revision 4

In order to further decrease the size of the brace parts, Professor Fischer recommended using wave springs instead of normal springs.

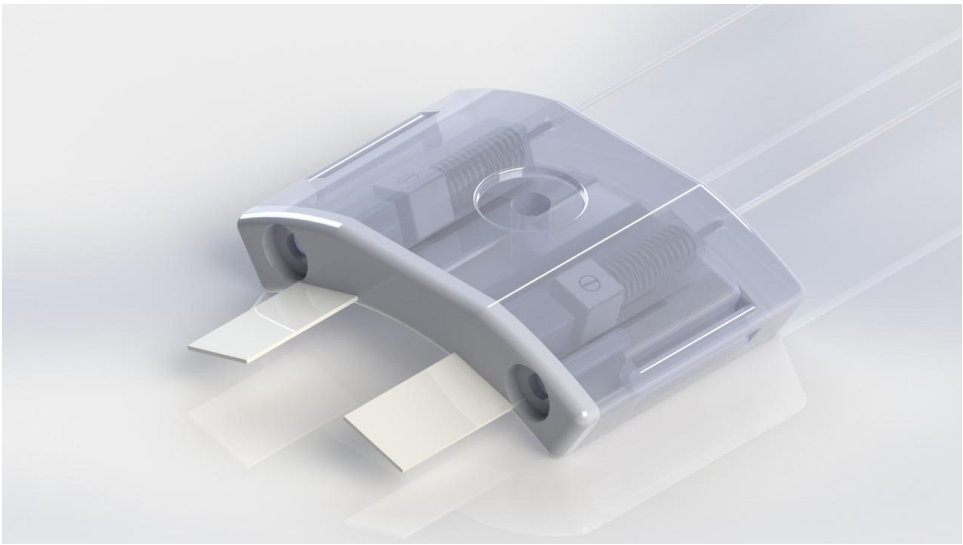


Figure 4.1.12: Elbow Brace Revision 4

Figure 4.1.12 shows a close render of the fourth revision of the elbow brace. In this version, the size of the brace part is about half as long at the previous revision. In order to fit the SoftPot membrane potentiometers in the brace, the team had to order custom sized membrane potentiometers from Spectra Symbol.

4.1.4.6 Elbow Brace Design Revision 5

The fifth revision of the elbow brace design addressed an issue with the cables going across the outside of the elbow. If the arm is flexed at the elbow, the cables will take the shortest path, going directly from the proximal part of the brace to the distal part of the brace. In order to force the cable to travel about the elbow, two additional smaller brace parts are placed just above and below the elbow.

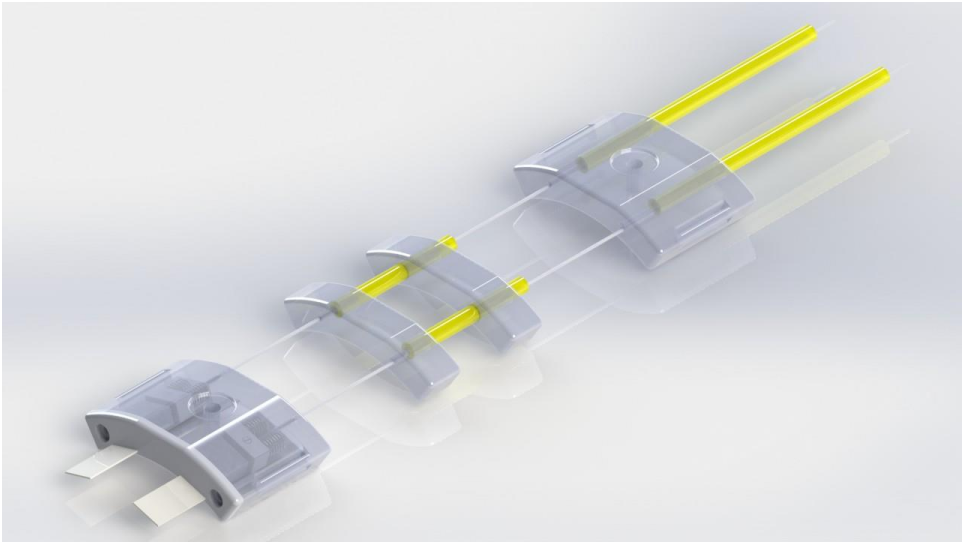


Figure 4.1.13: Elbow Brace Revision 5

Figure 4.1.13 shows the additional parts for guiding the cables around the elbow. One end of the shorter segment of the Bowden housing is first roughened using 80 grit sandpaper then glued with J-B Weld: Original Cold Weld Formula Steel Reinforced Epoxy into the smaller distal brace. The other end, however, is allowed to freely translate within the smaller proximal brace, thereby preventing the Bowden housing from flexing outwards when the arm is extended at the elbow. The placement of the brace parts can be seen in

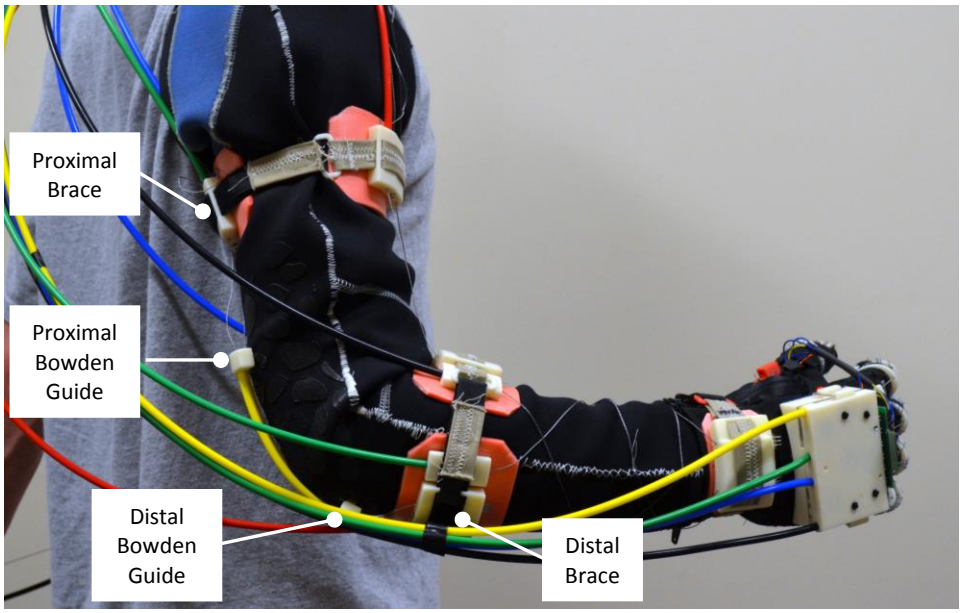


Figure 4.1.14: Brace Part Placement

Figure 4.1.15: Elbow Brace Revision 5 (Close)

In addition to adding the smaller brace parts for cable management about the elbow, additional wave springs were added to the series elastic sensors. The design in Revision 4 only allowed for one wave spring in the device, and the 19.98 lb/in spring constant of the wave

spring was not high enough for the expected loads on the part. Having only one spring effectively means that the series elastic sensor would frequently reach its limits for force sensing. By placing two wave springs in parallel, the effective spring constant on each actuator is doubled to approximately 40 lb/in (see **Figure 4.1.15**).

4.1.4.7 Elbow Brace Design Revision 6

The sixth revision of the elbow brace reduced the amount of free travel the ball plunger blocks have within the brace by extending the rectangular extrusions in the cap. The rectangular extrusions prevent the ball plunger block from travelling past the extension limit of the springs, thereby eliminating readings from the potentiometer past the maximum extension of the spring. In addition to lengthening the rectangular extrusions, the width and height of the extruded rectangles were increased; however, this change ended up causing the cap to not fit on the end of the brace properly.

4.1.4.8 Elbow Brace Design Revision 7

The seventh revision of the brace was created after revision 6 was 3D printed. In this version, the slots for the membrane potentiometers are widened and flattened, improving fit. The width and height of the extrusions on the cap were also decreased for improved fit. The assembly and performance of this final revision is addressed in Section 5.1.3 Elbow Brace Design.

4.1.5 Wrist Brace Design

The design of the wrist brace is borrowed from the design of the elbow brace. As visible in **Figure 4.1.16**, the brace part is extended. In the additional extruded section of the wrist brace, a slot was created for the shim stock to pass through.

Figure 4.1.16: Wrist Brace Segment (Close)

Instead of having the Dyneema cables run parallel with the length of the forearm, the Dyneema is wrapped around the forearm, thereby allowing for the pronation and supination motions.

Figure 4.1.17: Wrist Ring Cable Guide

Because the Dyneema is wrapped around the forearm, a method for preventing the Dyneema from constricting the forearm was implemented. **Figure 4.1.17** shows the CAD model of a pair of arcs that form a circular ring when put together. Each segment of the ring is attached to the Neoprene sleeve independently of each other. The segments of the ring allow the two arcs to separate when the user is donning the sleeve. The x-shaped grooves in the ring serve as guides for the Dyneema. The notches and protrusions at the ends of each of the arc segments serve as locators, allowing the arc segments to properly align when compressed. After testing these rings, the team determined that the neoprene sleeve was sufficient for limiting constriction.

4.1.6 Glove Design

In addition to providing the actuation needs of an assistive device, a system designed for hemiparesis patients must be easy to put on for patients with limited dexterity. Various

features of the system were re-designed with an emphasis on substantially improving the process of donning and doffing the device.

The glove design was adapted from the previous project work with Bowden cable actuated finger guides adhered to a stretchy minimal weight glove. Anchored above the fingernail and on the fingertips, cables run through cable guides on top and below the patient's fingers to mimic the flexion and extension motion of the tendons in one's finger. As shown in **Figure 4.1.18** and **Figure 4.1.19**, the cables are constrained by guides on each digit of the fingers.

Figure 4.1.18: Glove Designed by Delph et al. [1]

Figure 4.1.19: Previous Finger Guides, Designed by Delph et al. [1]

One of the drawbacks of this design is that when the flexion cables are pulled to close the hand, the guide stretches the fabric outward instead of transferring tension to the fingertip.

In order to ensure that the glove was still customizable and solved the problem with a pliable glove, the team explored the idea of solid rings. The idea was to generate a set of user specific rings instead of two piece guides so that the extension and flexion guides were connected. The Solidworks models for the finger rings would be linked to an Excel spreadsheet so that a full set of rings could be made by entering a patient's finger diameter into a table. The solidworks "Design Table" feature allows users to have the various dimensions used to define a part be linked to cells in a spreadsheet, and cells linked to each other via background calculations. This table would automatically calculate the dimensions of the entire ring based on only the width measurement of the knuckles between the ring and the fingertip on the appropriate finger. Please refer to Section 5.1.1: Glove Design and **Table 5.1.1:** Finger Ring Design Table for additional information.

The exact calculations for the proper sizing underwent several slight changes as the exact thickness of the glove, sensors, and sensor pocket was not known, but a conservative estimate was used and found to be appropriate based on tests, and ultimately was the correct size once the glove was more completely assembled. The calculation proceeds by reading the entered input for the finger width, increases it by a constant amount (to account for the thickness of the glove material), sets the new number as the inner diameter of the ring, and draws the rest of the ring from there. Initially the calculation added 0.075" to the diameter, but this value was found to be too small and constricted the wearer's finger. Next the value was multiplied by 1.1, but this approach was unreliable as this assumed that the thickness of the glove was proportional to the width of the finger, which was untrue. Finally adding a constant

of 0.15” was found to be the exact amount necessary, and indeed is the calculation used in the final prototype. A basic model of the rings can be seen below in **Figure 4.1.20**.

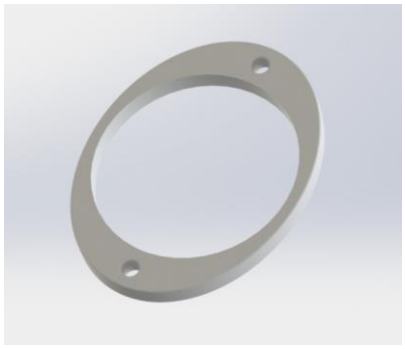


Figure 4.1.20: Solid Ring Finger Guides

In many cases, hemiparetic stroke survivors have limited control over their fingers or stiff joints that would make donning a glove-device challenging. The glove needed a way for a therapist to assist the patient’s fingers as they were donning the glove. The first glove iteration to tackle this constrain featured an “L” shaped zipper sewn along the pinky-side seam of the glove and across the top of the knuckles as represented in **Figure 4.1.21**.

Figure 4.1.21: Glove with L-Zipper Layout

The glove with an “L” shaped zipper allowed the patient to slide their hand in from the side, but did not give room for someone assisting the donning process to access the patient’s index and middle finger. The second glove iteration featured two zippers along the side seams as represented in **Figure 4.1.22** so that the top of the glove could be hinged up during the donning process. The outside zipper ran along the pinky seam while the inside zipper ran along the seam above the thumb.

Figure 4.1.22: Glove with Two Zipper Layout

Several iterations for donning glove were considered while assessing the effort needed to move the hand and fingers during the donning process. Looking at the device from a higher standpoint, there are three major parts that a patient would need to thread their arm through: the elbow anchors, the wrist anchors and wrist flexion limiter, and finally the finger rings on the glove. Attaching the above listed parts to a neoprene wetsuit sleeve and attached glove allowed the patient to thread their arm through every part until the sleeve was completely on their arm and hand. This procedure would be easy for a healthy individual with full control and high dexterity.

In order to simplify the device donning process and allow for a therapist to assist individuals with limited control over their fingertips, a two-step donning process was devised.

Instead of permanently fixing the wrist of the glove to the end of the neoprene sleeve, the bottom of the glove would be attached to the sleeve and the top of the glove would fold up.

In the two step donning process the patient would insert there hand into the sleeve with the glove bent down. In the second step the patient would insert their fingers into the glove and then zip the sides of the glove closed. The zippered sides of the glove also allows a second individual the ability to help guide the patient's fingers into the glove.

In order for the top half of the glove to be folded up during the donning process the Bowden cables for extending the pointer through pinky fingers must be disconnected. The detaching mechanism went through several iterations trying to optimize the ease of use and keep the thickness of the mechanism slim and close to the surface of the glove. The first designs entertained were placing each on a track with a disconnect joint on the track. Alternatively, the team decided in using a system where the extension cables were wrapped around small spools. The connection would then be made by connecting the axles of these pulleys. The torque on the end affecter would originate from the tension from the Bowden cable leaving the backpack wrap around one pulley on connected by a detachable axle to another pulley before being threaded to the end of the finger.

The first major design for the detaching mechanism consisted of small pulleys with torsional springs on the same axis as the pulley. The two halves of the detaching mechanism would be mirrored except that the lower half would have exposed cables running through the cable guides to the fingertips instead of connecting directly into Bowden cables. Exploring the coaxial pulley idea, the team realized that when the mechanism is detached, there needed to

be a component to keep the pulleys and cables in tension so that the axles would line up when donning the glove. Constant force springs were chosen to provide a constant torque on the pulleys.

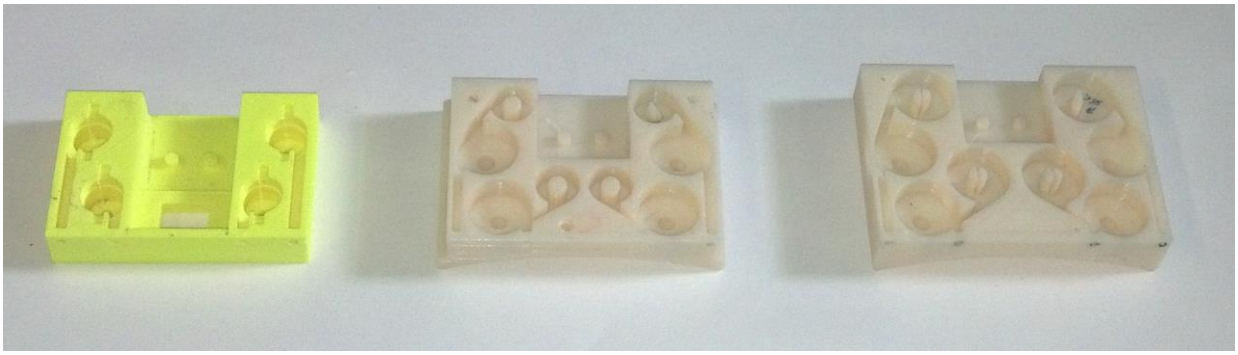


Figure 4.1.23: Three Iterations of the Detaching Mechanism in Chronological Order from Left to Right

This first iteration (see leftmost component in **Figure 4.1.23**) stacked the rotating elements vertically, and therefore increased the necessary height of the mechanism. One of the original goals for the project was to create a rehabilitative device that was as slim and the least inhibiting as possible. With this in mind the second major iteration of the detaching mechanism (see middle component in **Figure 4.1.23**) placed the tensioning springs on a separate parallel access to keep the thickness relatively shallow and used a counter wrapped cable to provide tension on the pulleys.

The third major iteration (see rightmost component in **Figure 4.1.23**) had the same pulley layout but included an acrylic shield on both halves of the detaching mechanism to keep all components in place. The heads of the socket cap screws also served as locating pins when connecting the two halves together during the donning process.

Additionally, A method for supporting the wrist in flexion and extension was explored. Establishing, that the wrist portion of the team's device is primarily used to support hand gestures, the wrist was supported as opposed to being actuated like the fingers. The wrist support device would anchor to the detachable mechanism and wrist pronoation module.

To assure reliability and minimize the number of moving parts the wrist design incorporated an exoskeletal beam with hard stops to limit the angle the wrist can bend. On the top of the wrist the thin metal beam provides spring like support for the wrist. Affixed to the surface of the metal beams are rows of trapezoids cut to a specified angle as seen in **Figure 5.1.14** in the final design chapter. The angle for these trapezoids was computed based on analyzing a picture of a human wrist in both their most extended and flexed states. This will allow the wrist to bend at a minimal angle, as mentioned in the design iteration chapter, with increasing resistance (provided by the surface mounted metal beam). Once the wrist reaches its limit angle, the trapezoids will meet and provide the hard stops.

4.1.7 Motors, Servomotors, and Encoders

The Rehabilitative Robotic Glove team used servomotors for actuating the device. Through the use of current limiters and PWM signals, the team was able to effectively control the position and torque outputted by the servomotors. However, the servomotors were incapable of making full revolutions, thus the spools designed for the project had to be large, able to retract enough cable for hand flexion and extension over a range of approximately 200 degrees. This, in turn, made it extremely difficult to make the system compact.

The following year, the Soft Robotic Exo-Muscular Arm Brace team decided on using stepper motors. Stepper motors can fully rotate, thus eliminating the need for extremely large spools. Despite the minimized torque, the stepper motors were not capable of putting out enough torque to actuate the elbow. Thus, additional motors were used to supplement the stepper motors. The benefit of using stepper motors, however, was that they provided excellent position control.

Based on the experiences of the previous two teams and research conducted by Chris Nycz, the team decided to use regular DC motors in conjunction with clutches and quadrature encoders. The motors selected were capable of outputting at least the minimum force required, and in most cases much more, thereby allowing users to not only move their appendage but also to manipulate small objects. The motors achieved this torque by having initial speeds of several thousand rpm's, and using custom fitted gearboxes to reduce the speed to a much more reasonable level, all while multiplying the torque significantly. This did have the unfortunate side effect of removing any back-drivability that the motors may have had. For an additional level of control, as well as preserving the safety of the system, a model of electromagnetic clutches were chosen by Chris Nycz to use between the motors and spools. These clutches would allow precise control over the force being transmitted to the spools by the motors, as well as provide a convenient quick-release mechanism as when the clutches are unpowered, the motors and spools are effectively disconnected. The encoders were used to track the positions of the spools, and had to be connected directly to the spools instead of the more traditional location on the motors, and the use of clutches meant that the motion of one was not absolutely tied to the other.

4.1.8 Finger Motor

4.1.8.1 Motor Selection

In order to find a satisfactory median between the space inefficient servomotors and the heavy, underpowered stepper motors, the team made a careful selection from the catalog of Maxon Motors. The specifications required were established during the previous projects, and so were carried through to this project. The requirements were: back driveable, weight less that 1kg, speed of 50 rpm's, torque of .65 Nm, peak current draw of 1.5A, and operating voltage of 12V.

The motors initially chosen, when combined with their custom-fitted gearboxes would meet almost all requirements set by the design statement: lightweight, powerful, quiet, and small. Unfortunately the one design specification not met was low cost, and so a comparable motor found on eBay.com, a Maxon Motor with a third-party gearbox attached was significantly and almost identical, and certainly sufficient for the needs of the project. For details on the motor chosen, please see Section 5.1.4.1: Finger Motor Units.

4.1.8.2 Finger Housing Revision 1

One challenge faced by the design team was the layout of the backpack that would contain the motors. The two previous MQP teams, by the nature of their projects being separate, had housed their motors in separate backpacks. Though the glove and brace had been in separate locations along the user's arm, the motors had been in the same location. Finding an orientation that would allow for five motor setups for the fingers and two for the

elbow and wrist motors would prove to be challenging. The first design revolved around avoiding all the complications that arose in previous projects from the motors being built into the actual backpack. The first major decision revolved around making each motor into its own modular unit containing every piece of the motor train. Ideally these units would be easily removable for maintenance and adjustment. The first iteration of the actual design involved having the cables from each motor unit being routed through the backpack structure using a system of pulleys to the upper left corner where they would join with the plastic tubing to become a Bowden system. Though the finer details of this design were never planned, the benefits and drawbacks were analyzed and it was determined that a better design was needed.

4.1.8.3 Finger Motor Housing Revision 2

The second design iteration kept the modular design of the first concept design, but sought a better method of managing the cables from the spools to the Bowden cables. To accomplish this, the modular design was taken a step farther to include the mounting location for the mounting cables with the rest of the unit. A model of the CAD assembly can be seen below in **Figure 4.1.24**. The motor train features a large central block to which all components are attached, as well as several clamps which are used to secure them. There is also a sliding mechanism used for attaching and removing the block to and from the actuation platform. It consists of a baseplate that will be permanently fixed to the actuation platform, and the motor block simply slides in and out, held in primarily by gravity. This revision features no locking mechanism for this as it was a last-minute design change.

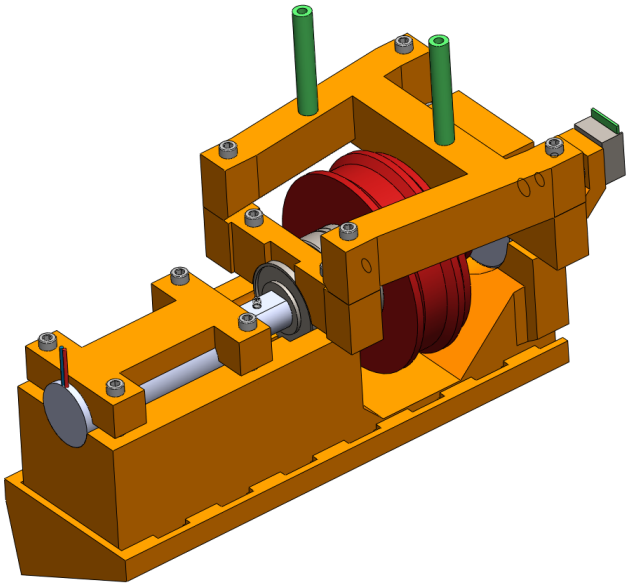


Figure 4.1.24: Finger Motor Train V2

The cables can be seen protruding from a cross-brace across the top, holding them firmly in place and causing each motor train to be absolutely independent of the others, and the backpack itself. The Bowden cables would still be attached to the sleeve, but since they only attach to the motors and sleeve they do not bind the motors to the backpack. With this design, aside from the wiring to power the motors, clutches, and encoders, each motor train may be freely removed from the backpack. The location of the cross-brace had a design failure causing it to block the primary access point to the lock nut used for clamping the spools together. To remedy this, the entire assembly was tilted 25° so that when several are stacked next to each other, they could be accessed from the exposed side.

4.1.8.4 Finger Motor Housing Revision 3

The final motor train changes little from the second revision. Most of the changes were brought about by creating a physical prototype of the motor train and making small changes from that. The greatest change came as lowering the surfaces that the heads of a few of the

clamp screws rested against. Due to a slight error in modeling, the screws in the model were slightly longer that the actual screws used in the assembly. A relatively simple fix, a slight removal of material was all that was necessary in the model. The clamp used for attaching the clutch has to be slightly reshaped to account for the wiring port being slightly larger than anticipated. Again, a simple change.

One problem that did go unnoticed for a long time was the interference between the spools and the cable mount, the piece across the top that the Bowden cables fit into. Its shape and position on the motor train caused it to interfere with the spools, essentially locking the small extension spool in place and preventing it from sliding along the threaded rod, effectively disabling the spool adjustment mechanism. The fix for this was to modify the shape of the cable mount so it had clearance over the spools and was not in the way. Finally, a large portion of the motor block and base was hollowed out. This, while reducing the weight by a trivial amount, reduced the material used in manufacturing by a significant amount, which served to reduce the cost significantly. This revision of the motor train can be seen below in **Figure 4.1.25**. This revision is discussed in greater detail in the Final Design chapter.

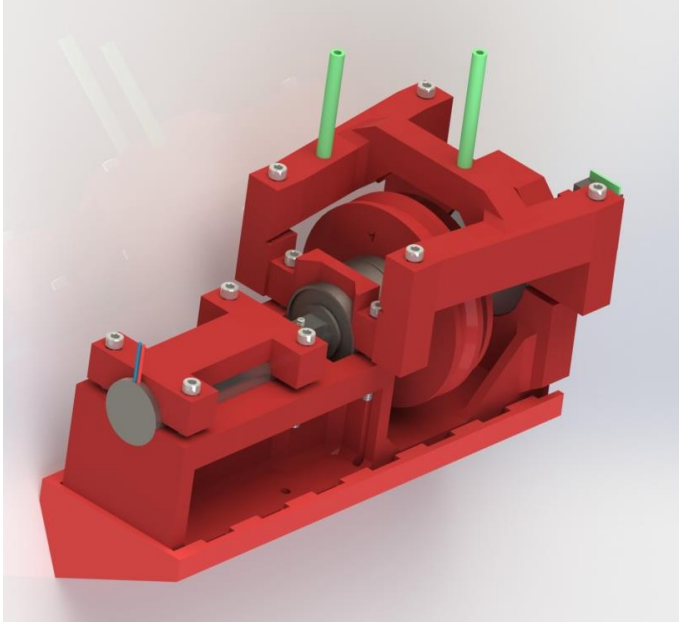


Figure 4.1.25: Finger Motor Train V3

4.1.9 Elbow and Wrist Motor

4.1.9.1 Motor Selection

The first step of finding appropriate motors was determining the expected torque output requirements of the motors. Quick calculations estimated a minimum of approximately 2 Nm of torque from the motor for overcoming the mass of the forearm when flexing the arm from full extension.

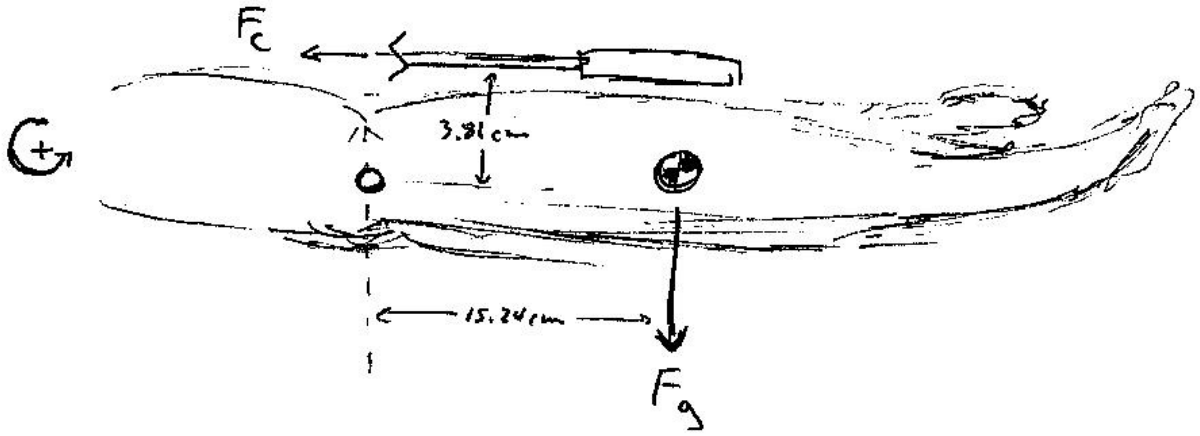


Figure 4.1.26: Motor Torque Estimation

In order to determine the approximate torque requirements for actuating the arm, the average mass of the forearm and hand was assumed to be 1.9 kg [52]. Estimates were made for the distance between the elbow's axis of rotation and the forearm's center of gravity and the distance between the elbow's axis of rotation and the cable tension, as visible in **Figure 4.1.26**. Assuming that the arm is at full extension, parallel to the earth, the force of gravity due to the mass of the forearm is calculated as follows:

$$F_g = 9.8\frac{m}{s^2} * 1.9\ kg = 18.6\ N$$

Next, the moment through the elbow's axis of rotation, notated as M_0, is assumed to be zero. The angular velocity of the forearm is also assumed to be zero. Therefore, the cable tension, F_c, required to maintain equilibrium may be solved as follows:

$$\sum M_0 = F_c * 3.81 - F_g * 15.24 = F_c * 3.81 - 18.6 * 15.24 = 0$$

$$F_c = 74.4\ N$$

Assuming a spool radius of 1” (2.54 cm), the torque output required from the motor is calculated as follows:

$$\tau_M = 74.4 * \frac{2.54}{100} = 1.9\ Nm$$

At $437.70 a piece, the team quickly discovered that these specifications required Maxon gearmotors that exceeded the budget of the project. As a result, the team decided on using cordless drill motors. Cordless drills are designed to provide high torque, while minimizing weight and maximizing efficiency. The motors for actuating the wrist and elbow have similar requirements. The first cordless drill considered was the Black & Decker SS12C. With a rated torque output of 130 in-lb (14.69 Nm) this drill would have easily performed the task of actuating the arm. However, the cordless drills selected for this project were purchased from Harbor Freight, item #68239. With a rated torque output of 84 in-lb (9.49 Nm) Harbor Freight drill proved sufficient for the application. The team decided on modifying the Harbor Freight cordless drills for two reasons: ease of modification and price.

4.1.9.2 Gearmotor Modification

The Harbor Freight gearmotor did not have two different speeds, thus the gearmotor could be removed from the cordless drill housing as one unit. The gearmotor assembly extracted from the Harbor Freight cordless drill had a slip clutch built into the gearbox.

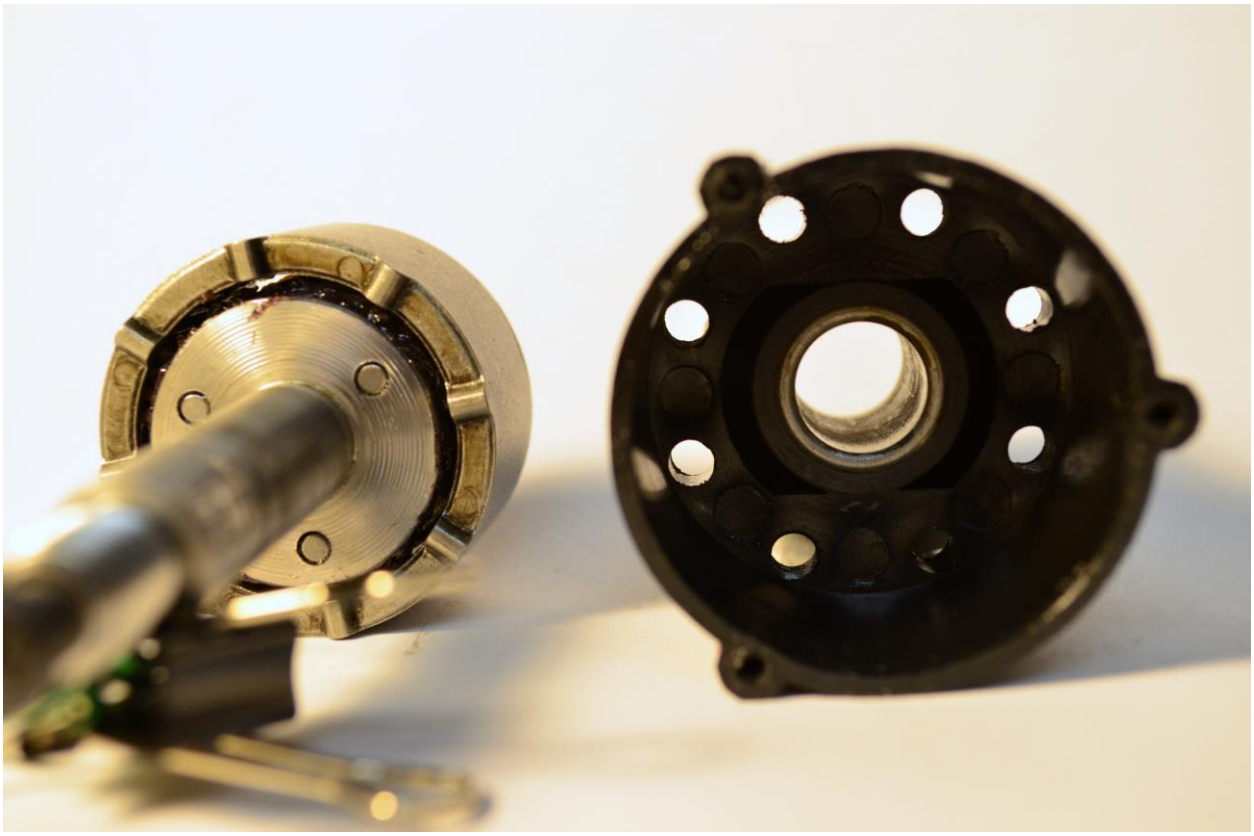


Figure 4.1.27: Disassembled Gearbox

When the output shaft of the cordless drill reached the cordless drill's set torque, the ring gear of the planetary gearbox would slip within the plastic housing. The ribs on the surface of the ring gear as shown in **Figure 4.1.27** centered themselves in the holes of the plastic housing when the clutch was not slipping.

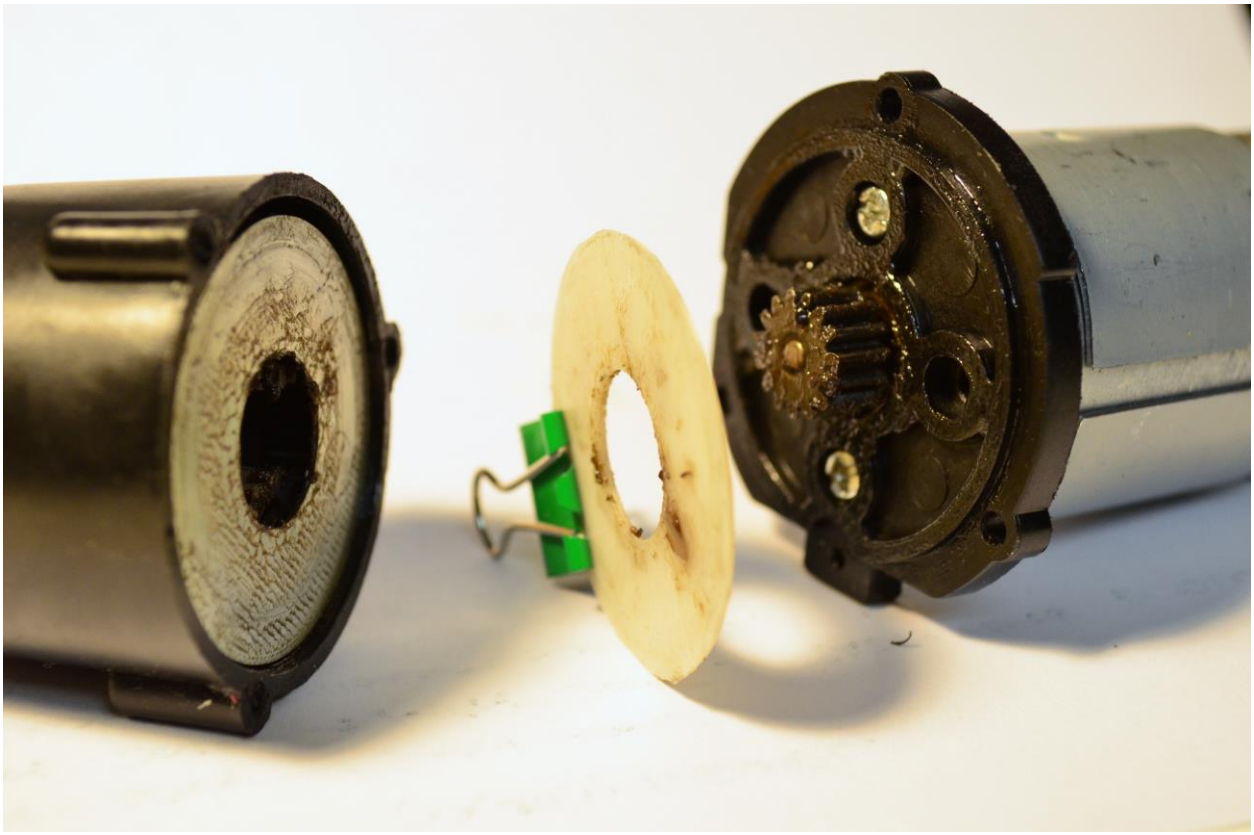


Figure 4.1.28: Plastic Insert in Gearmotor

In order to mitigate the ring gear slippage, a 0.05” thick plastic spacer was 3D printed and installed between the washer and motor head as shown in **Figure 4.1.28**. When assembled, the spacer would create enough force between the motor head, ring gear, and plastic housing to prevent the ring gear from rotating.

4.1.9.3 Elbow and Wrist Motor Housing Revision 1

The first concern for the elbow motor housing is reducing the size of the device. Because the team had originally planned to fit this device in a backpack, the primary consideration was reducing the size of the actuator.

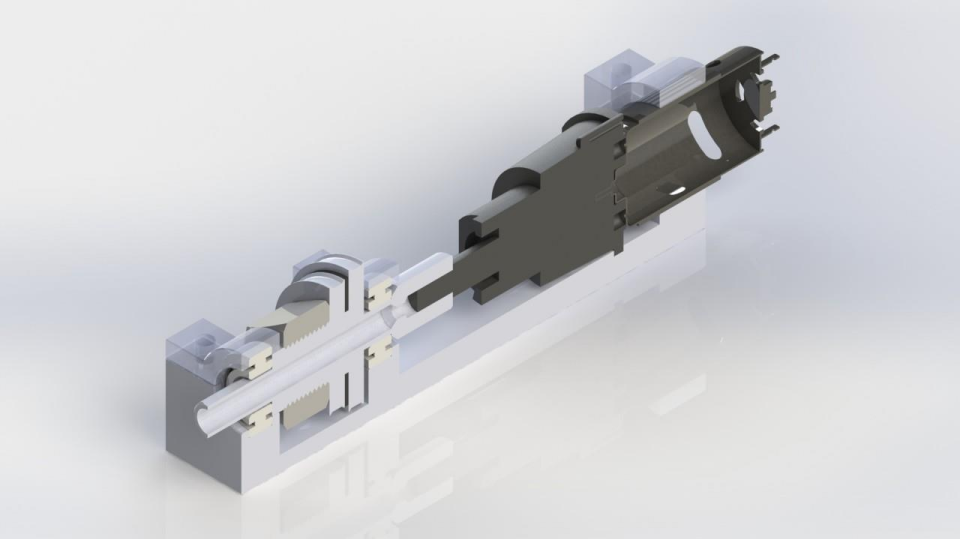


Figure 4.1.29: Elbow Motor Housing Revision 1 (Section View)

Figure 4.1.29 shows a section view of the model. The shaft on the end of the gear motor has a right-hand threaded exterior and a left-hand threaded interior screw. In order to take advantage of the threads, a hollow shaft was designed to thread onto the outside of the motor shaft. A left-hand screw could then be passed through the shaft and installed, thereby effectively attaching the shaft to the gearmotor. Each of the two bearings had grooved circular slots cut into the base for placement. Although compact, this design would have been extremely difficult to machine, thus the design was quickly altered in favor of machining concerns as opposed to size or weight.

4.1.9.4 Elbow and Wrist Motor Housing Revision 2

With machinability in mind, the second revision of the elbow motor housing removed many of the features that would have been difficult to machine. Instead, more standard off-the-shelf parts were selected for construction.

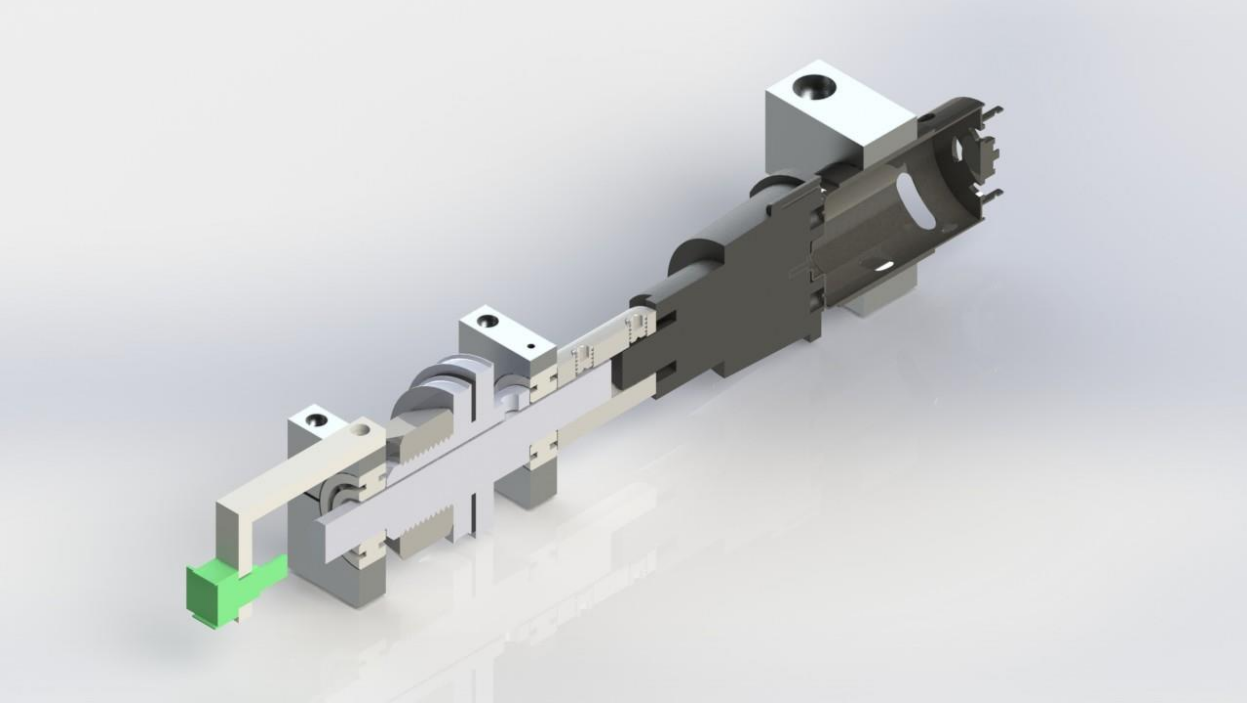


Figure 4.1.30: Elbow Motor Housing Revision 2 (Section View)

While the second revision of the part was being developed, the team had decided to mount all of the actuators to one plate of aluminum. As a result, the flat base that connected all of the retaining blocks in the model was removed, as visible in **Figure 4.1.30**. Instead of machining a shaft for the part, a standard 0.5" diameter D-shaft was selected. The D-shaft is attached to the gearmotor using a 0.5" to 0.5" coupler. In addition, the retaining blocks for the bearings do not have a groove. Instead, the bearings are held in place solely with friction. The CUI C14 quadrature encoder is attached to the final bearing retaining block with an L-shaped 3D printed part.

4.1.9.5 Elbow and Wrist Motor Housing Revision 3

After the team decided to have a single plate of aluminum as the base for all of the motors, the design for the elbow and wrist motor housing had to be reevaluated.

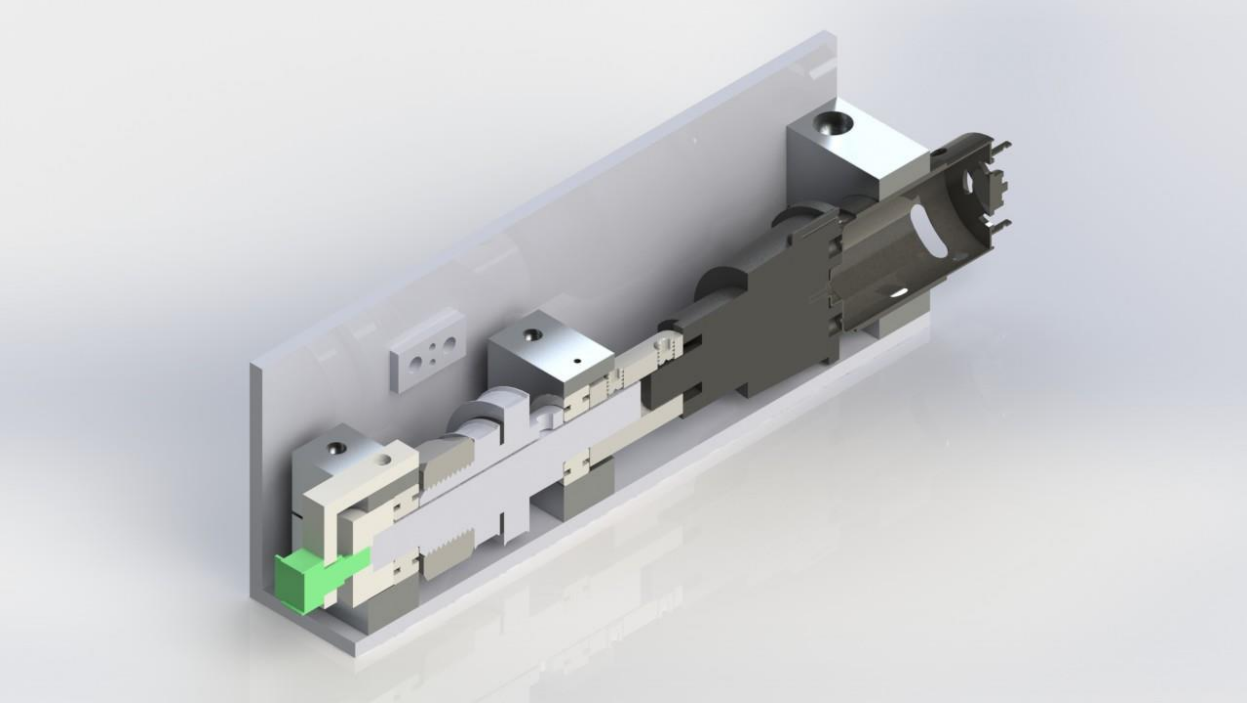


Figure 4.1.31: Elbow Motor Housing Revision 3 (Section View)

The third iteration of the motor housing design took into consideration what resources were readily available. The machine shop on campus often has scrap material, and there were a number of 2” by 3” and 3/4” thick blocks of aluminum. These could be easily machined to create the bearing retaining blocks for the bearings and motor. As visible in **Figure 4.1.31**, an L-shaped aluminum channel was added to the model to serve as the base for all of the parts. In addition, the side of the aluminum plate serves as the termination points for the Bowden cables.

4.1.9.6 Elbow and Wrist Motor Housing Revision 4

The fourth revision of the elbow and wrist motor housing design includes some modifications to strengthen the part and remove some of the excess material.

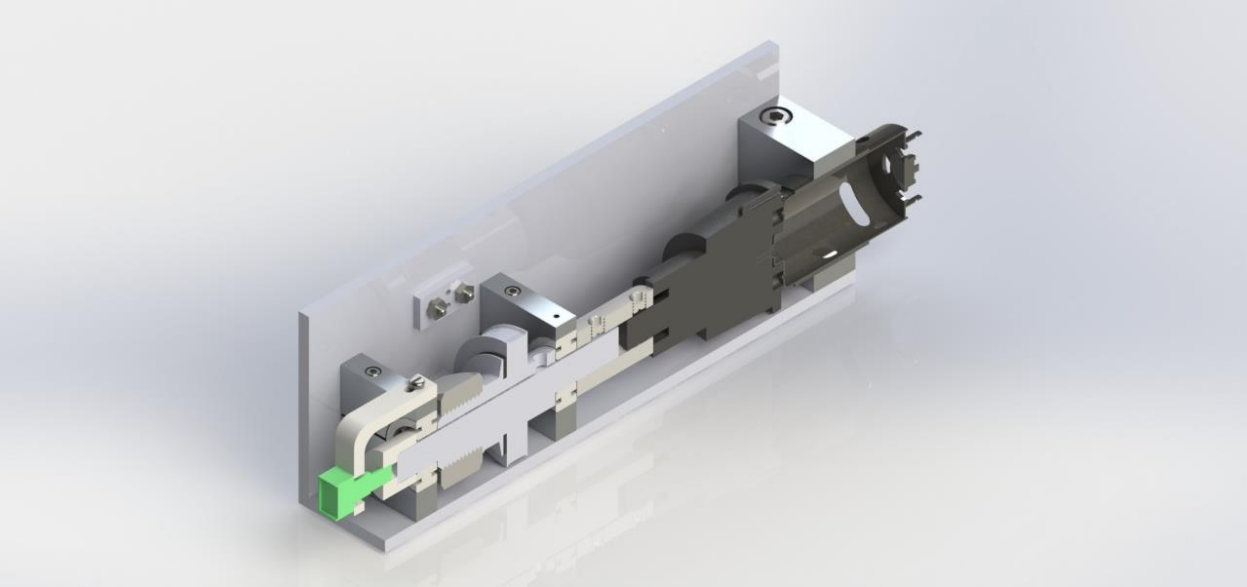


Figure 4.1.32: Elbow Motor Housing Revision 4 (Section View)

Because the side wall of the motor housing is likely to experience high forces due to the Bowden system, the bearing retaining blocks were made wider, such that they are flush against the side of the L-channel, as visible in **Figure 4.1.32**. The blocks thus provide extra support to the L-channel, preventing it from folding inwards. In addition, the cost of buying 3/8" thick stock for the bearing retainers was deemed inconsequential compared to the weight reduction, thus 3/8" thick blocks are used for the bearing retainer blocks as opposed to 3/4" thick blocks. In addition, instead of 3D printing a straight L-shaped bracket, the part was modified from a sharp right angle to an arced piece to minimize warping.

The size of the smaller spool was also reduced to account for the different extension and retraction lengths required for extending and flexing the elbow. In order to maintain a relatively high surface area between the large and small spools, a flange was added to the smaller spool, as seen in **Figure 4.1.32**.

4.2 Electrical System Architecture

Two main design choices were considered for the electrical system architecture. The first one consisted of a more simple design with only one microcontroller. The second option evaluated consisted of several microcontrollers communicating with one another. Both options would be sufficient to handle sensor data acquisition and motor control. When compartmentalizing the system with several microcontrollers, the approach of how many controllers to use and what task would they be executing came under discussion.

One of the advantages of the single controller approach was that it did not require data acquisition synchronization amongst multiple controllers. Furthermore, no communication protocols were required and the data did not need to be sent to/from other controllers since everything was already centralized. However, with only one controller the options for future expansion of the system were very limited because most I/O ports on the microcontroller would be used. Sensors are spread throughout the body with some being on the hand and others near the motors. If all the sensors were to connect to a single controller, the analog sensor lines would be longer since the microcontroller cannot be located near all of the sensors. This would also allow other signals to interfere with the signal being measured. Therefore, placing the controllers within close proximity of the sensors being measured ensures accurate sensor measurement.

The multiple microcontroller approach consisted of three different circuit boards, which were to be placed on different areas of the glove and the actuation platform, each with their own microcontroller. The approach was to have three controllers: Main, Motor, and Sensor circuit boards. One of the main advantages of this multiple-controllers approach was the ability

to split up the computation tasks. Each microcontroller would be dedicated to executing one function as opposed to one microcontroller doing several functions, such as pressure and flex sensor data acquisition and processing, as well as motor control. Separating the motor control onto another board as opposed to motor control processes occurring within a single controller allowed for a faster feedback control loop where the motor controller would not have to interface with many other boards and the computer at the same time. This setup allowed for the Main board to act as a hub between the other boards and as an interface to the computer. Moreover, software development and debugging could be divided up amongst the different developers of this project in a much easier and less conflicting manner. A possible problem that could occur with this multiple-controllers design choice was that since each controller had its own oscillator driving it, the oscillators would never be exactly the same and therefore data acquisition timing would drift over time. Nevertheless, this possible drawback could be solved by having the Main controller send a signal to the other controllers to synchronize the data acquisition. Therefore, the electrical sub-team selected the multiple controllers option as our high-level electrical system structure, using the PIC32MX795F5F512L microcontroller from Microchip in each of the different boards.

4.2.1 Sensors

One of the main components required for the development of this exomuscular sleeve were a set of various sensors that allowed collecting sufficient data to control the performance of the device for therapeutic purposes. In order to properly actuate the fingers for exercises such as closing and opening the hand, it was important to ensure that the angles of flexion and extension did not exceed safety thresholds that could cause pain to the users. Beside safety of

the patients, measuring angles can also provides data for position based control. Thes data can also be used by patients and therapists to track the progress. Flex sensors were considered a viable option to measure the angles of flexion/extension of the fingers.

Typical therapeutic exercises usually include grasping motions in order to improve the patient's abilities to perform daily tasks, such as grabbing a glass of water. Pressure sensors were evaluated as an option to detect pressure at the fingertips, thus allowing to determine the quality of the grasping tasks performed by the users. Additionally, the design team considered providing visual feedback to the patients that could serve to indicate the completion of specific tasks. Light emitting diodes (LEDs) were considered suitable to indicate the patient's degree of motion completion using a predetermined color scale. **Sections 4.2.1.1** and **4.2.1.2** provide a detailed evaluation of the aforementioned sensors.

4.2.1.1 Flex Sensors and Force Sensitive Resistors

An important component of upper limb rehabilitation is to determine the mobility of the fingers. In order to obtain such data, flex sensors could be placed on each finger to measure the angle of flexion or extension and ensure the motor actuations are not reaching extreme angles that could harm the users. Furthermore, force sensitive resistors (FSR) could be placed on each fingertip to measure the force exerted when grabbing objects.

The signal received when reading these sensor data would need to be amplified. Instrumentation Amplifiers were determined to be the best choice to achieve this signal amplification since they are characterized by having very high input impedance, which would not affect the output or gain. Two types of instrumentation amplifier variants were evaluated:

those using dual power supplies and those using Rail-to-Rail Amplifiers. When experimenting with dual power supply instrumentation amplifiers, it was determined that they could not achieve the desired signal gain between 0 and 5 Volts. Furthermore, these types of amplifiers had a tendency to clip at the lower end at approximately 0.9 Volts and at the upper end at approximately 4.4 Volts (when using supplies of 0 and 5 Volts). On the other hand, Rail-to-Rail Amplifiers were more prone to reach a signal very close to the rails. When tested with 0 and 5-Volt supplies, these amplifiers were able to swing between 0 and up to 4.5 Volts. These characteristics can be observed in the graphs shown below in **Figure 4.2.2** and **Figure 4.2.3**, which display the result of simulations done with NI Multisim version 12 (by National Instruments Electronics) with one of the sensor circuits (see **Figure 4.2.1**).

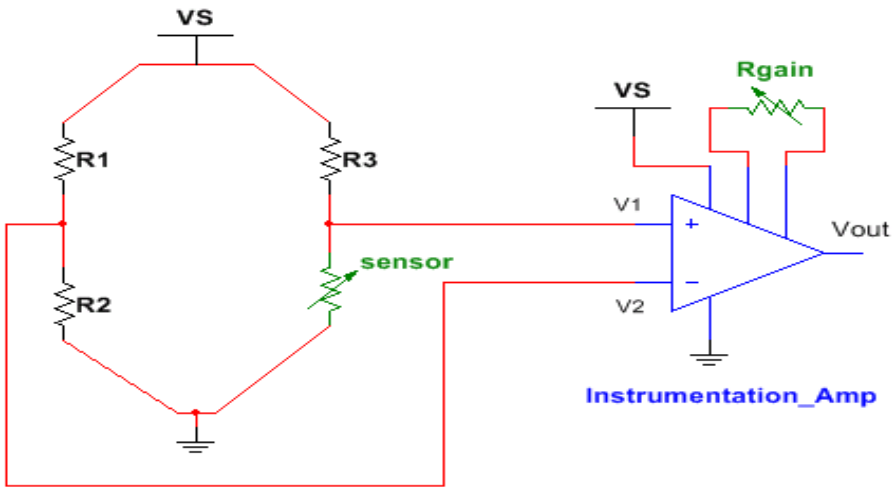


Figure 4.2.1: Wheatstone Bridge circuit used for simulation.

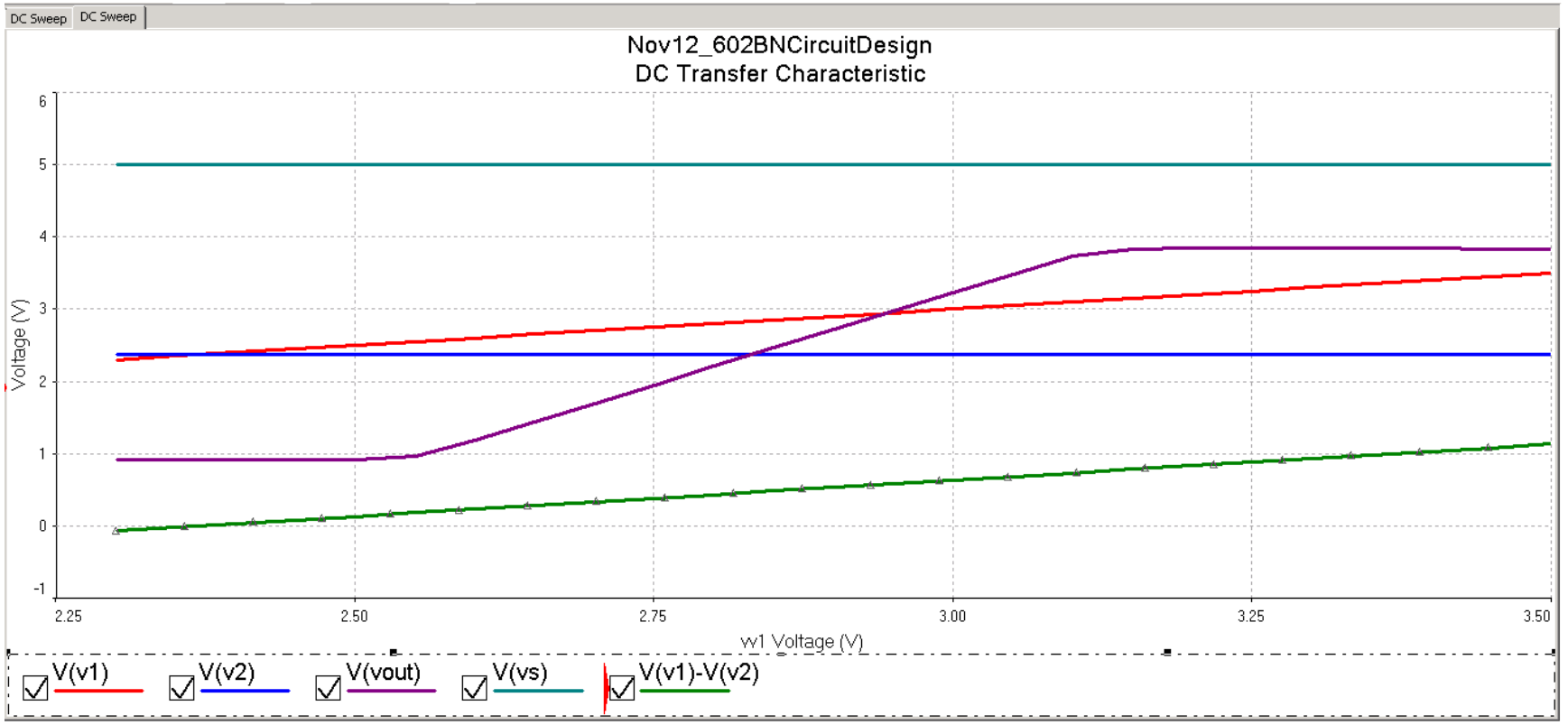


Figure 4.2.2: Graph showing Vout clipping well below Vs and Ground when using Bipolar Instrumentation Amplifier (AD602BN).

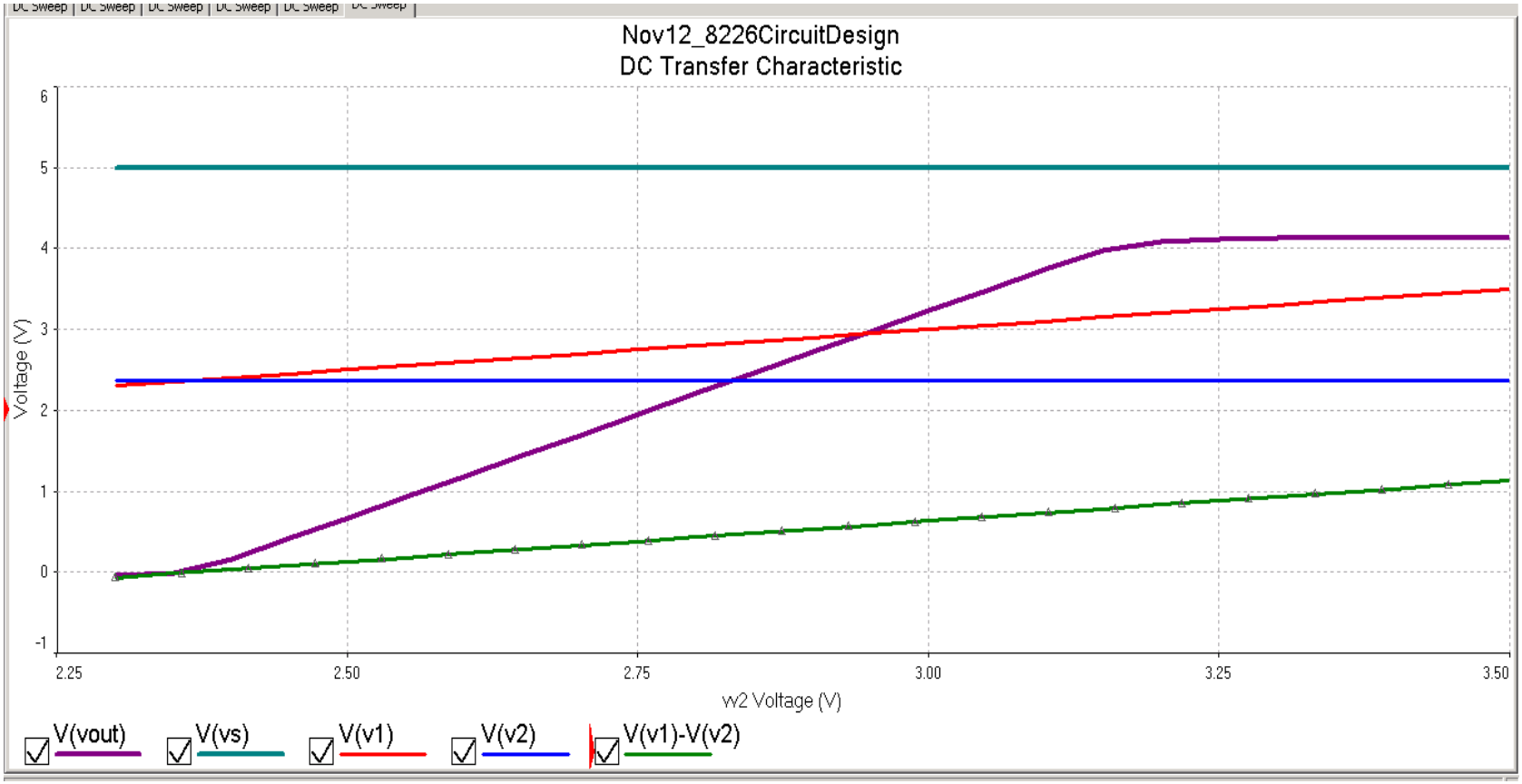


Figure 4.2.3: Graph showing Vout clipping closer to rails when using Rail-to-Rail Instrumentation Amplifier (AD602BN).

4.2.1.2 Visual Feedback through Light Emitting Diodes

Several design ideas were considered to give haptic or visual feedback to the user, signaling they had accomplished a movement goal, such as placing small vibrating motors or light emitting diodes (LED) around the device. Vibrating motors would allow the user to actually feel the feedback as the motors vibrated against their skin. However, this option might not be successful when treating post-stroke patients with very limited sensation on their upper limb. Furthermore, the vibrating motors could interfere with the EMG data collection, which would not be suitable for future iterations of this system.

Two choices were evaluated regarding the placement of the LEDs: on top of the fingernails or on top of the knuckles on the hand. When considering the placement of the LEDs on the fingernails it was noted that the users might not be able to receive the visual feedback when grabbing objects since their hand would be closed and the ends of their fingers could be out of sight. Since most tasks for upper extremity rehabilitation involve grasping actions, this LED placement would not be ideal. However, arranging the LEDs on top of the knuckles on the hand would be much more practical since users would be able to receive visual feedback even when grabbing objects. The only case considered in which users would not be able to perceive visual feedback from the LEDs on the knuckles would be when the palm of their hand was facing them. Nevertheless, in this case the user could visually see their hand closing, so the LEDs would not be necessary.

Two main choices were considered regarding the selection of the proper LEDs, such as passive or active (optoelectronic) LEDs. When evaluating the bipolar LEDs, it was determined that these have too many digital output lines, i.e. 3 wires per LED, which

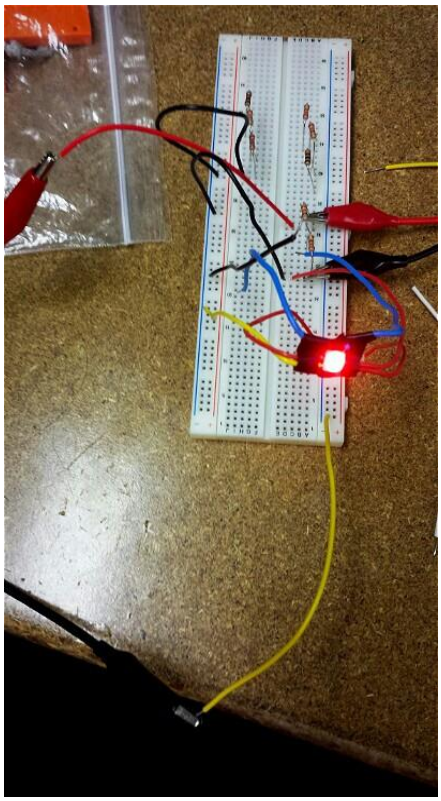


Figure 4.2.4: Bipolar LED Testing on Prototyping Board.

would require using 3-to-1 multiplexers than directly selecting an LED from one serial communication line. LEDs with integrated IC can be daisy-chained and selected by serial bits from serial communication lines on the Sensor board (see **Figure 5.2.8**). These LEDs were tested on a prototyping board using an Arduino Uno microcontroller as shown in **Figure 4.2.4**. To conduct such testing the LEDs were surface-mounted, which introduced some difficulties when soldering them onto the prototyping board. On the other hand, the control LEDs displayed below in **Figure 4.2.5** cost the same as the bipolar LEDs ($ 0.50 for the WS2812 by WordSemi) and included built-in Integrated Circuits. The ICs require SPI data from the microcontroller to individually control and select color and output, respectively.

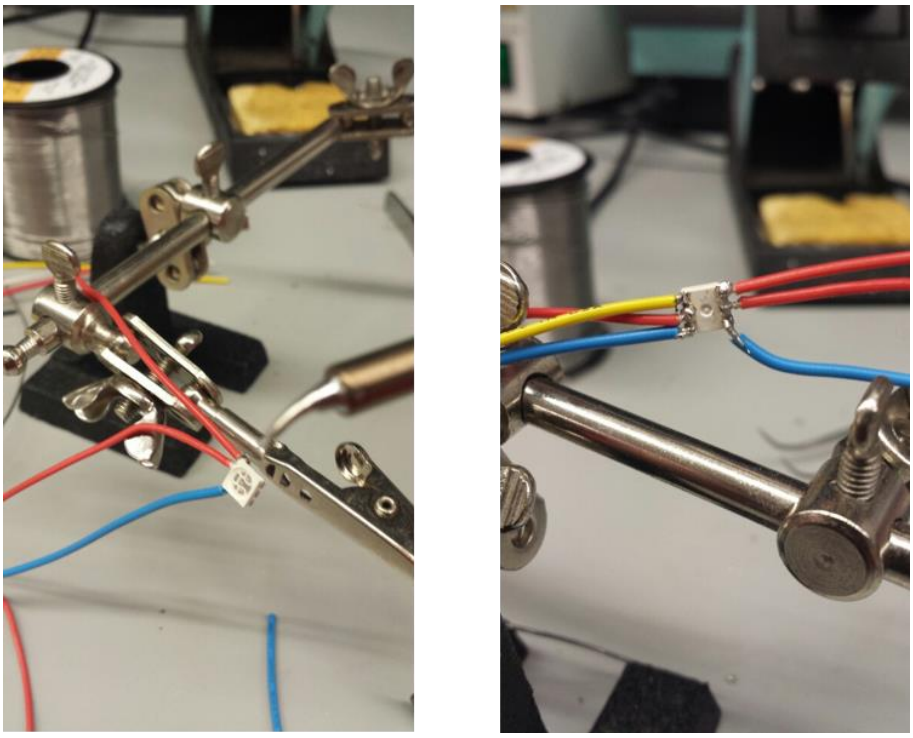


Figure 4.2.5: Control LED with Built-in Integrated Circuit.

4.2.2 Actuation Control Mechanisms

A total of 7 motors were used to implement the actuation system of this integrated exomuscular sleeve; 5 for actuating individual fingers, 1 for wrist actuation and 1 for elbow actuation. Since the wrist and elbow motors required different operating voltages than the finger motors, it was necessary to have regulator circuits for the various voltages required on the motor board. In addition, each one of the 7 motors was connected to an encoder to provide position information as a control feedback, sending 2-channel outputs to decoder chips (details offered in **Section 4.2.2.5**). Motors were also connected to 7 motor drivers for control independence. **Section 4.2.2.2** provides detailed information about the functionality of the motor drivers.

Within the motor driver circuitry, a main control feedback, current limiting circuit was inserted. The current limiting circuit setup was suggested by the advisors since it was successfully used in the previous iterations of this MQP [2]. This current limiting circuit facilitated the current control for the integrated actuation system (details offered in **Sections 4.2.2.1** and **4.2.2.4**). The motor drivers were fed PWM signals from an external PWM driver. A selected 16-output PWM driver (Texas Instruments, TLC5940) sources different PWM signals to all 7 motor drivers and 5 clutches also included in the actuation mechanisms of this system (refer to **Section 4.2.2.3** for more details). Other control inputs included were 8 series elastic actuators (SEAs). By utilizing two op amp ICs (low power quad op-amps), each IC including 4 internally compensated op amps for required analog signal conditioning, the SEAs could be able to deliver a suitable range of voltage outputs (details on signal conditioning and SEAs are offered in **Section 4.2.2.6**).

4.2.2.1 Motor Control

The design team for this project decided to use a current sensing circuit to implement the motor control algorithm. Several methods were evaluated, such as voltage or position control using transistors or encoders, respectively. Research conducted in this area along with the recommendations obtained from the advisors of this project, indicated selecting the current limiting design choice since it is a common way to control DC motors, which has been thoroughly tested. Furthermore, the current measurements could be used to obtain torque estomates and provide control feedback.

4.2.2.2 Motor Drivers

Several DC motor drivers were evaluated, including the L298 chip from STMicroelectronics. Some of the requirements taken into account before making the selection of the most suitable motor driver were the output current, the number of full H-bridges, and the implementation methods. The output current of the system required a maximum between 2 and 4 Amps to operate the motors previously selected. Having a full H-bridge (see **Figure 4.2.6**) was highly considered since the system required fully backdrivable (achieved by electromagnetic clutches on finger motors and turn off powers of elbow and wrist motors) DC motors for its proper performance during the different rehabilitative tasks assigned to the patients. To avoid the need for a separate current sensing circuit, it was suitable to select a driver that included current sensing capabilities.

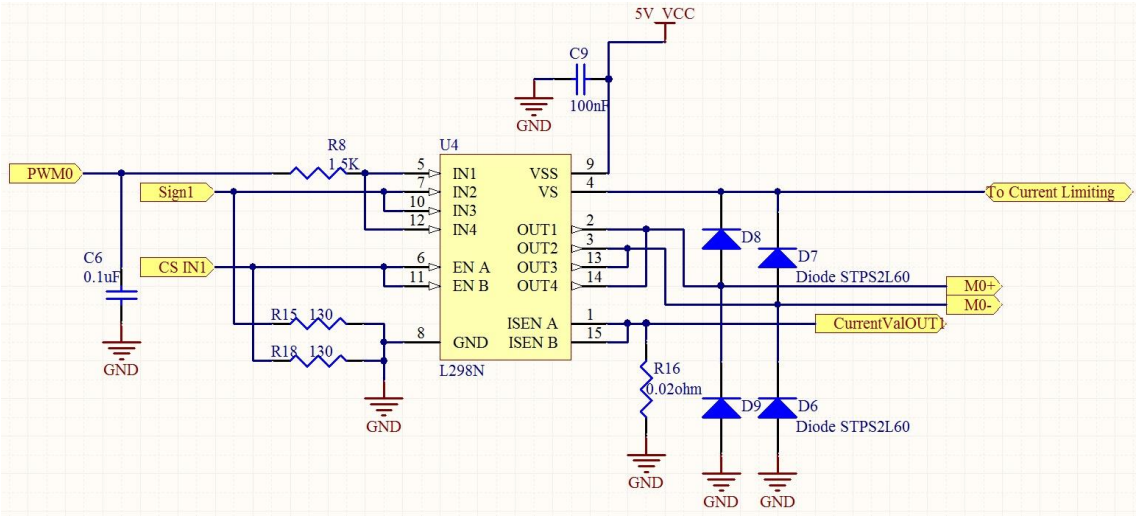


Figure 4.2.6: Motor Driver L298 (4A Dual H-bridge)

4.2.2.3 Pulse Width Modulation Drivers

Additional pulse width modulation (PWM) inputs were required for the motor drivers due to the lack of designated PWM outputs from the selected microcontroller. The system required a total of 12 PWM signals; 7 for motors and 5 for the clutches that assist the finger motors. Each clutch required a PWM input with a NPN transistor as switch for clutch control circuit (see **Figure 4.2.7**).

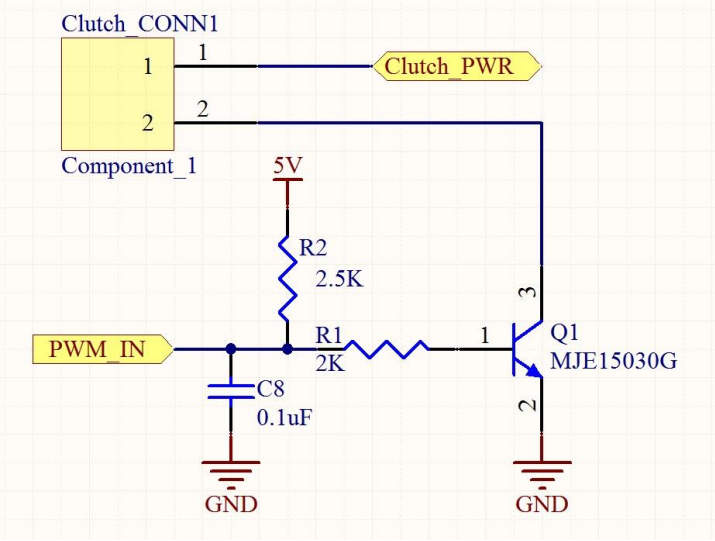


Figure 4.2.7: Clutch-NPN-PWM Input Circuit Schematic Diagram

However the microcontroller only had a maximum of 10 PWM outputs from 5 Output Compare pins. Since 2 additional PWM supplies were needed, the team decided to source all PWM signals from an external PWM driver. Among the main criteria taken into account to select the PWM drivers were: maximum PWM duty cycle, reasonably high data transfer rate, and simple communication interface. The TLC5940 PWM driver chip from (Texas Instruments Inc.) can generate a PWM signal from 0% to 100% duty cycle. The control update loop between the Motor board and the motors was designed to run with a frequency of at least 1 KHz.

Therefore, the frequency supplied to drive and communicate with the selected PWM driver (TLC5940) would have to be high enough to satisfy this update rate. The TI TLC5940 datasheet specifies the calculation of the frequencies as shown below:

$$f_{(GSCLK)} = 4096 \times f_{(update)}$$

$$f_{(SCLK)} = 193 \times f_{(update)}$$

where $f_{(GSCLK)}$ is the minimal frequency to drive the peripherals (in this case, motors and clutches), $f_{(SCLK)}$ is the minimal frequency to drive SPI communication, and $f_{(update)}$ is the desired update rate for the entire system. The maximum SCLK and GSCLK frequencies specified for the TLC5940 are 30 MHz. Using the equations above for these frequencies results in the minimum SCLK and GSCLK frequencies required to achieved the desired 1 KHz frequency. The results of these computations yield $f_{(GSCLK)}$ = 4,096 KHz and $f_{(SCLK)}$ = 193 KHz. The PIC32 operates at 80 MHz, but it can be stepped down to anywhere within the range of 193 KHz to 30 MHz. The selected 16-bit output-channel PWM driver chip (TLC5940) can be seen below in **Figure 4.2.8**.

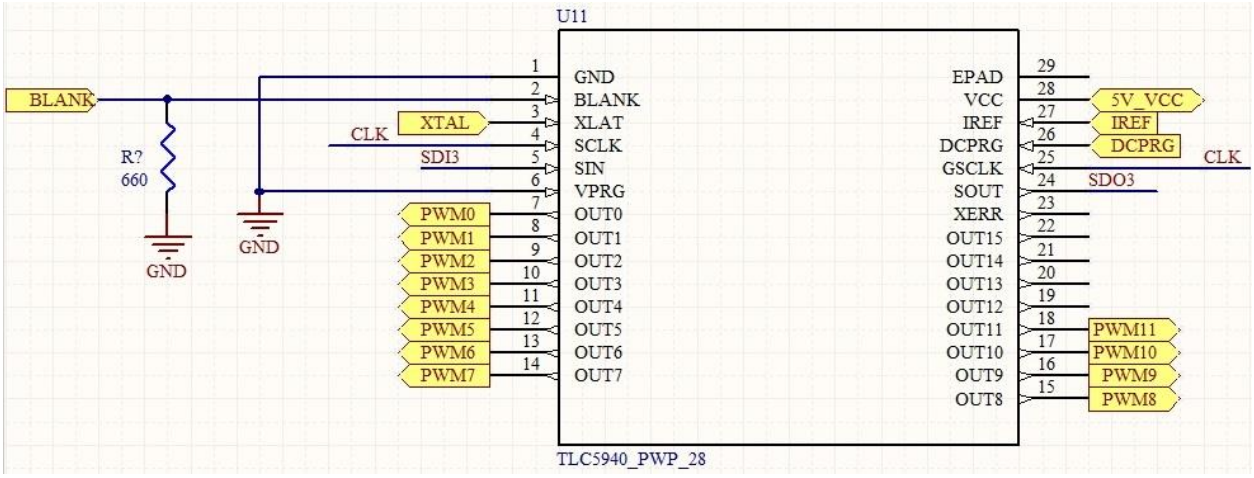


Figure 4.2.8: 16-Output-Channel PWM Driver TI TCL5940

4.2.2.4 Current Limiting Circuit

Consistent with the design choice of having four separate circuit boards, one of them was strictly dedicated to control the seven motors included in this system, i.e. five for the fingers, one for the wrist, and one for the elbow. The main components of the current control system are the current limiting and current measuring circuits. Current limiting is accomplished through the use of a current limiting circuit, which limits the amount of current flowing to the H-Bridge as shown in **Figure 4.2.9**. The current limit is controlled by a digital potentiometer. The greater the resistance, the more current is inhibited. Current control is accomplished by setting the current through a current limiting circuit and measuring the current used by the motor through a current sense circuit.

To implement this motor control board, current limiting was required as a safety factor and to provide feedback for force control. Research conducted regarding the design choices evaluated by previous projects leading to the development of this device suggested using a

zener diode with 4 transistors to regulate current and a digital potentiometer to control the current threshold from the microcontroller. The current limiting circuit implemented as suggested by previous MQP teams [2] would be connected in between the positive voltage supply of the motor (+V_b) and a transistor on positive motor power (P-FET). The current limiting circuit can be seen below in **Figure 4.2.9**.

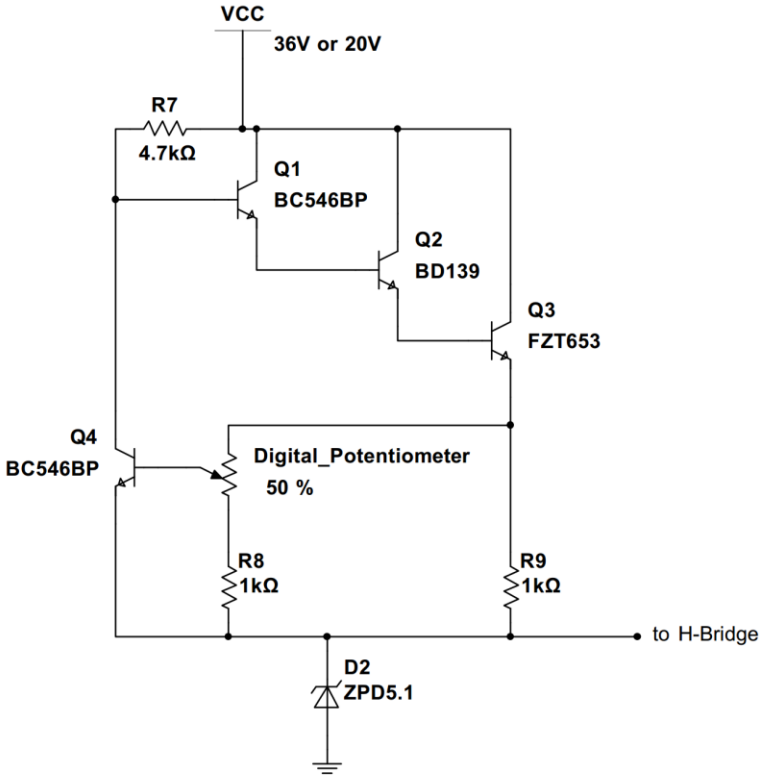


Figure 4.2.9: Current Limiting Circuit

4.2.2.5 Encoders and Decoders

Quadrature decoders were considered ideal to accomplish bidirectional position control since these do not require additional potentiometers. There were a few parameters that needed to be evaluated before making a proper selection, such as operating frequency, counter resolution, and communication protocol with the microcontroller. The parameters were chosen

based on functional objectives along with the integrated system architecture. A few different decoders were researched and picked with reasonably high operating frequencies, high counter data resolution, and parallel outputs in order to ensure dedicating the least possible amount of I/O pins from the microcontroller to the decoder interface. Quadrature encoders and decoders were selected based on the established parameters. There were relatively fewer options for quadrature encoders with high resolution, which yielded the selection of optical encoders. For decoder chips, one of the high-end quadrature decoder ICs with 14 MHz maximum clock frequency was picked. It had 8-channel output, allowing to be chained in parallel from a decoder to another in order to reduce the number of input pins for the PIC32 microcontroller.

4.2.2.6 Series Elastic Actuators

With the design of this exomuscular sleeve it was very important that the system had the capability to measure the torque exerted by users as they perform different exercises. Series Elastic Actuators (SEA) were considered as an option to translate this type of data into electrical feedback to be sent out to the system. Moreover, using SEAs could serve to limit the transient spikes in the torque exerted by the system to ensure the patient's safety. Series elastic actuators are a combination of potentiometers, springs, and actuators, where the displacement of the spring is used to determine the amount of force (Fa) exerted onto the potentiometer by applying Hooke's Law (see **Figure 4.2.10**).

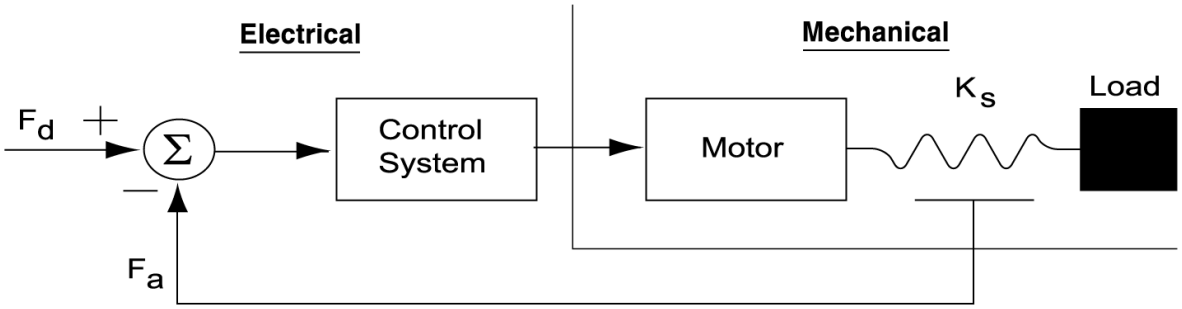


Figure 4.2.10: Functional Block Diagram of Series Elastic Actuators [53]

Four different SEAs could be placed on the elbow brace to ensure its flexion (two SEAs) and extension (the other two), while five other SEAs could be used on the wrist for measurement not for actuation, i.e. four to determine pronation and supination and one for flexion and extension. Preferably two SEAs are needed to accurately measure torques for each movement since reading torque from only one SEA cannot cover large movements of the forearm. Torque may differ from one end to the other on the side of the arm; therefore, two SEAs are used and located on both sides of each area to perform flexion and extension. By averaging torques measured from the two sideway ends of rotating plate, more accurate torque can be achieved. The analog data obtained from all SEAs could be processed on the Motor control board as control feedback to ensure that the system would perform nominally.

Two different options were considered when evaluating the type of potentiometers that could be used, i.e. magnetic potentiometers and soft membrane potentiometers. Magnetic potentiometers tend to have a wide range of operation and less actuation force (0.18 N pull force required). However, their price was almost twice the price of the soft membrane potentiometers ($22 for 10kΩ 50mm magnetic potentiometer and $13 for 10kΩ 50mm soft membrane potentiometer). Considering the number of sensors needed and budget allowed for

this project, the $9 difference was fairly significant (70% of the price of soft membrane potentiometer). Moreover, when evaluating soft membrane potentiometers it was found that their actuation force ranged from 0.6 to 1.5 N, which was significantly larger than that of magnetic potentiometers. This particular characteristic of soft membrane potentiometers was highly desirable since the force exerted by the users is likely to exceed the 0.6 N lower boundary and would not press any difficulties when measuring the displacement of the spring during each movement.

Moreover, the thickness of this type of potentiometer was approximately 1/7 the thickness of magnetic potentiometers, i.e. 0.02" and 0.138", respectively. These characteristics of soft membrane potentiometers would reduce not only the amount of space dedicated to placing these devices on the sleeve, but also the total weight of the system.

4.2.3 Analog to Digital Converters

The output of each sensor goes through the low pass filter seen below in **Figure 4.2.11**. The filter is a unity-gain 2nd order Butterworth Sallen-Key low pass filter. Since the signal input is a low-frequency mechanical data with favorable voltage range, there was no need of gain. Research conducted revealed very stable output and high performace in Sallen-Key topology filters [54]. A cutoff frequency of approximately 25 Hz was preferred because the changing rate of the input signal would not exceed 15 Hz since the motions performed by stroke patient do not typically reach this frequencies. This 25 Hz cutoff frequency would also keep data below the 25 Hz range. A 2nd order active filter was designed according to calculations suggested in [54]. The designed filter's frequency response can be seen below in **Figure 4.2.12**.

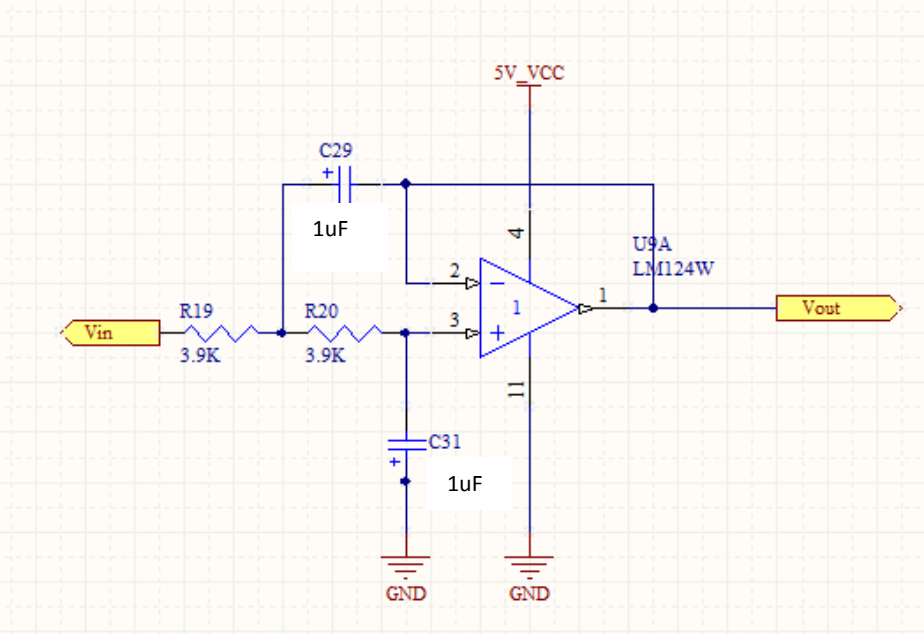


Figure 4.2.11: Low-Pass Filter for Flex and Pressure Sensors.

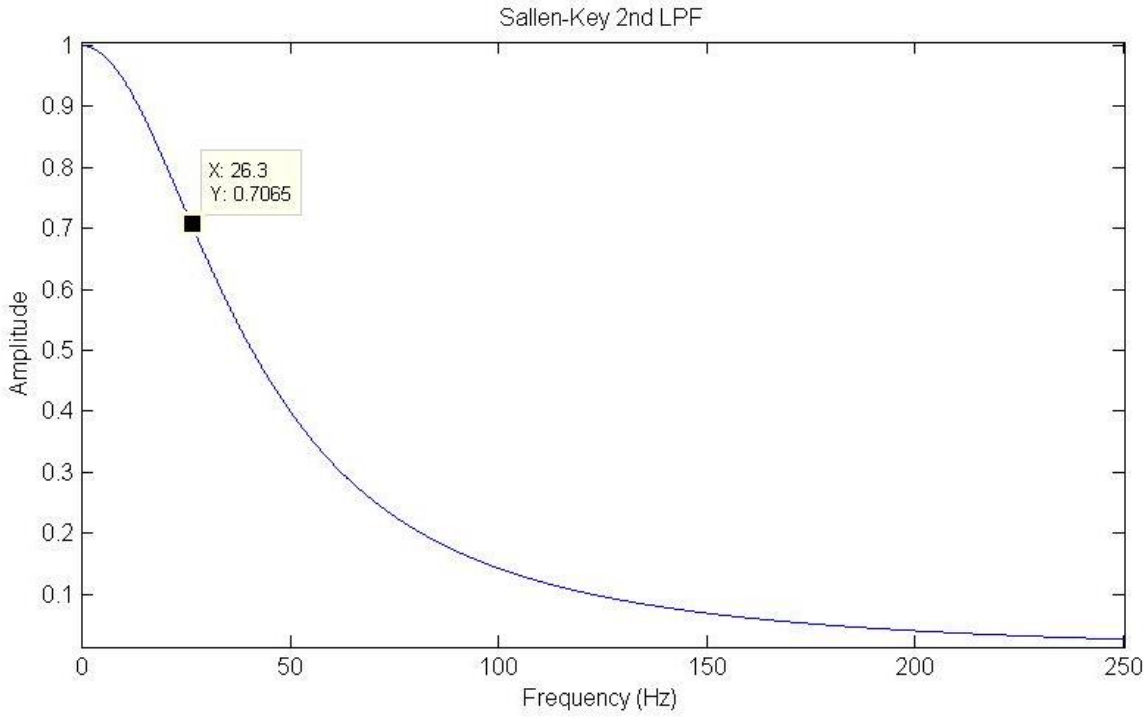


Figure 4.2.12: Magnitude Response of 2nd order Butterworth Sallen-Key Low Pass Filter

The output of this filter would be sampled via Analog to Digital Converters (ADC) and sent to the microcontroller for data processing. In order to read accurate signals for data processing, it was required to choose a suitable ADC. The initial option considered was using the internal ADC included in the PIC32 microcontroller. PIC32 microcontrollers have built-in 10 channel ADCs with a resolution of 10 bits. However, there were many more sensors included in this system, which suggested the option of using external ADCs with possibly higher resolutions. Another requirement taken into consideration when selecting the proper ADC to use was the power supply conflict since the sensors would be driven by 5V power whereas each microcontroller would be driven by 3V power. Therefore, the ADC had to be able to read analog signals ranging from 0 to 5V, while the clock signal and the chip select signal coming in from the microcontroller would only be 3V. The major concern with this setup was that the ADC might not register 3V chip select or clock signal as logic high. One of the solutions was to change the range of output of sensor circuits from 0-5V to 0-3V, and have a rail-to-rail ADC for the digital input lines. This option was not preferred since it limited the output range. Another solution to the problem was using an external ADC capable of referencing separate power source for digital and analog operation, such as the AD7927 chip from Analog Devices. This chip included eight additional channels with 12 bits of resolution. With these specifications in mind the AD7927 ADC chip was chosen and set up as displayed below in **Figure 4.2.13**.

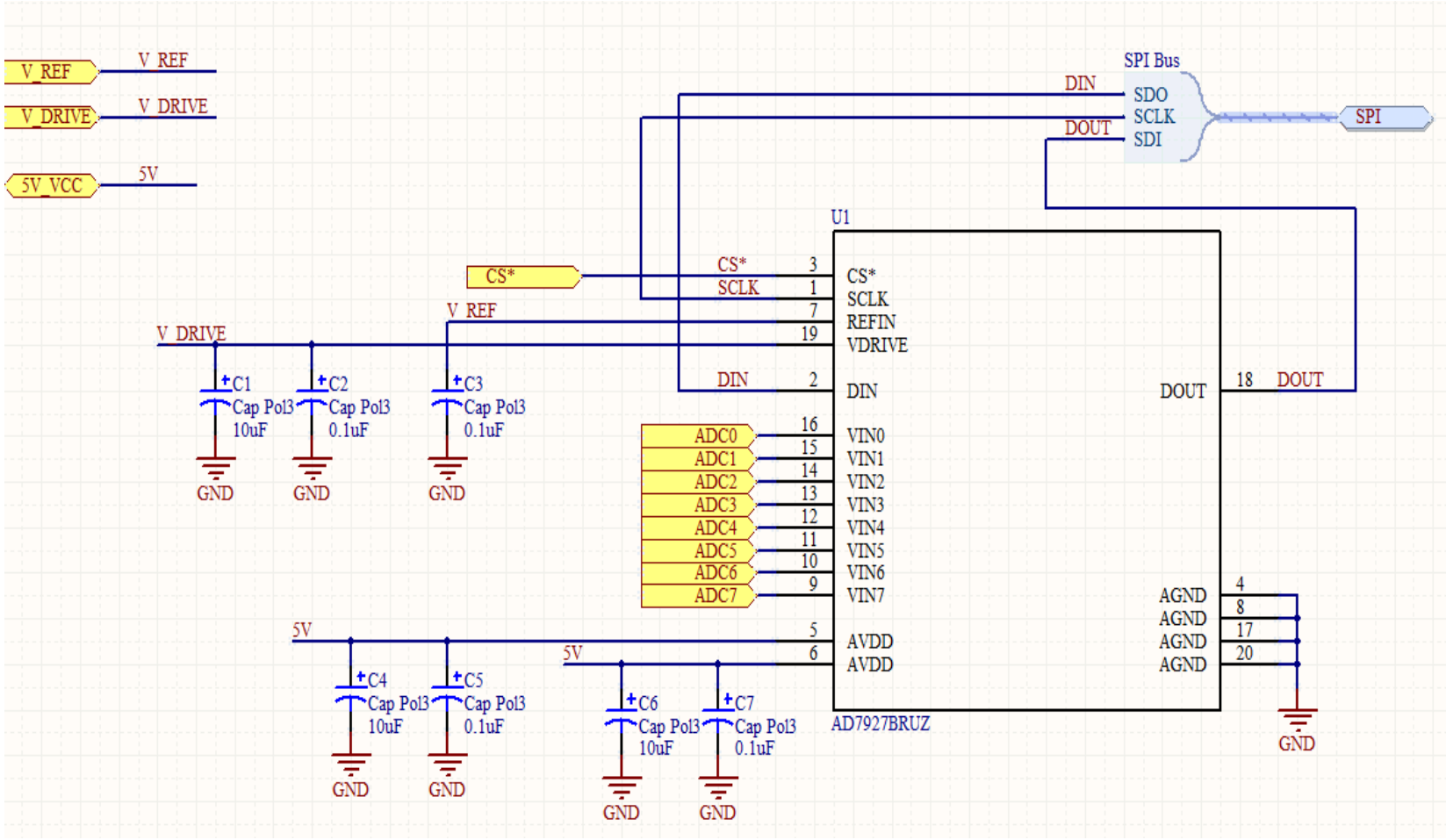


Figure 4.2.13: Schematic Setup for the Selected Analog to Digital Converter

4.3 Communication Protocols

Two different approaches were considered to send data between the boards: sending all the data in one large packet or split it into multiple packets. The single packet approach had the advantage of when the packet was received, all the data was there. There would be no need to keep track of how many packets were expected to compare it with how many packets were actually received. For instance, if the computer commanded the sensor board to publish the index finger data, with the single packet approach, one large packet was being sent over UART regardless of the amount of sensor data needed to be sent.

The multiple packet approach, however, was more conservative on the UART bandwidth because it would only send a packet with data from a single finger. This way, only data that was requested by the main board would be sent over UART. The downside to this option was that all data from one board would not be received at the same time since the total amount of data would be split up into multiple packets. On the receiving end, the packets would also have to be tracked to see which packets came in the same instance.

The approach implemented was the multiple packet approach. This was chosen because, as the overall system is expaned into the future when more boards are added and an increasing amount of data is being sent around, only packet data with useful information will be transferred. With the single packet approach, there is a lot of waste, meaining the entire packet isn’t being populated with useful data.

5. FINAL DESIGN

The final design consisted of an exomuscular sleeve covering the upper limb from the fingertips to the shoulder. The flexion and extension of the fingers is actuated through cable tension applied to the fingertips and measured with flex and pressure sensors connected to each finger. These cables run from the fingertips to the actuation platform where the motors and the spools are located. The final prototype of the exomuscular sleeve can be seen below in **Figure 5.1**. The entire system is controlled by a Main circuit board placed in the actuation platform, which communicates with three other boards monitoring the sensors of the fingers and the motor control. A high-level functional diagram of these circuit boards and their communication with the computer can be seen below in **Figure 5.2**.

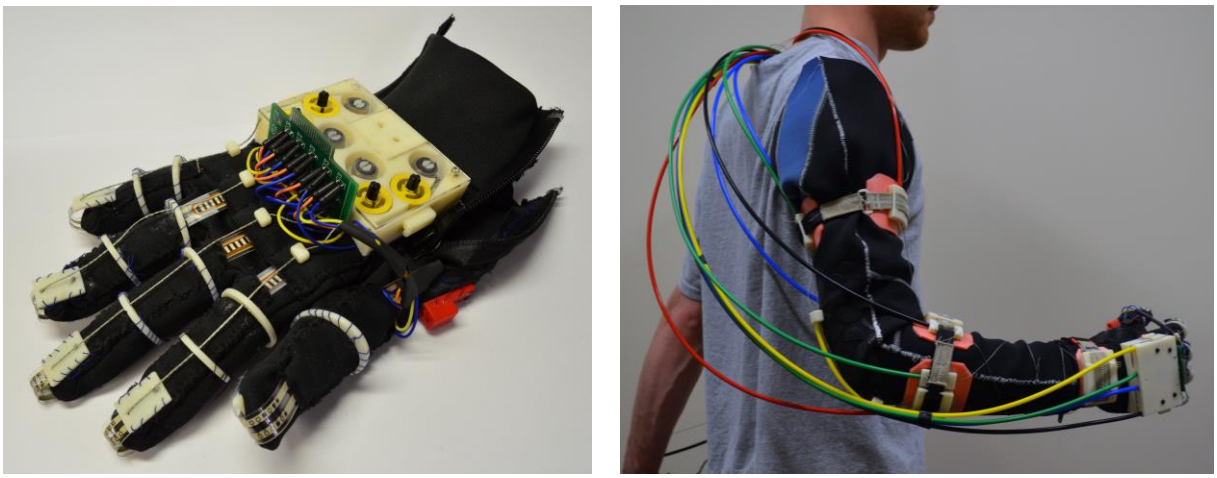


Figure 5.1: Exomuslcuature Sleeve Final Prototype

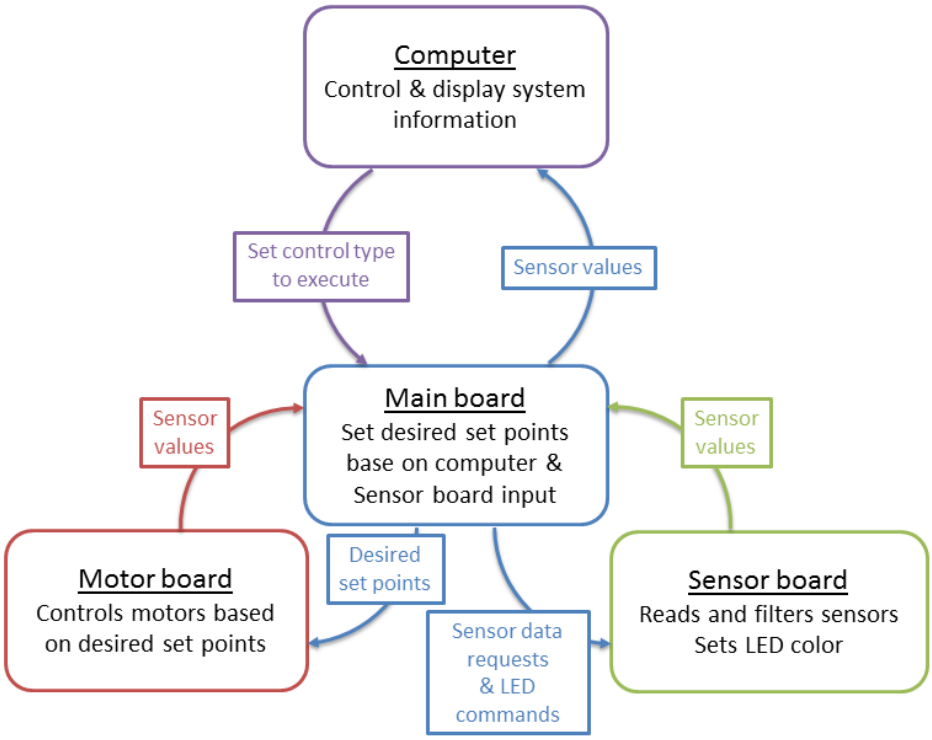


Figure 5.2: High-Level System Functional Diagram.

5.1 Mechanical Subsystem

For organizational purposes the prototype was divided into four segments: the glove, the wrist piece, the elbow brace, and the actuation platform (see **Figure 5.1.1**). Indeed each segment was designed independently and even by different team members. In the sections that follow the entire final design is reviewed categorically by these sections.

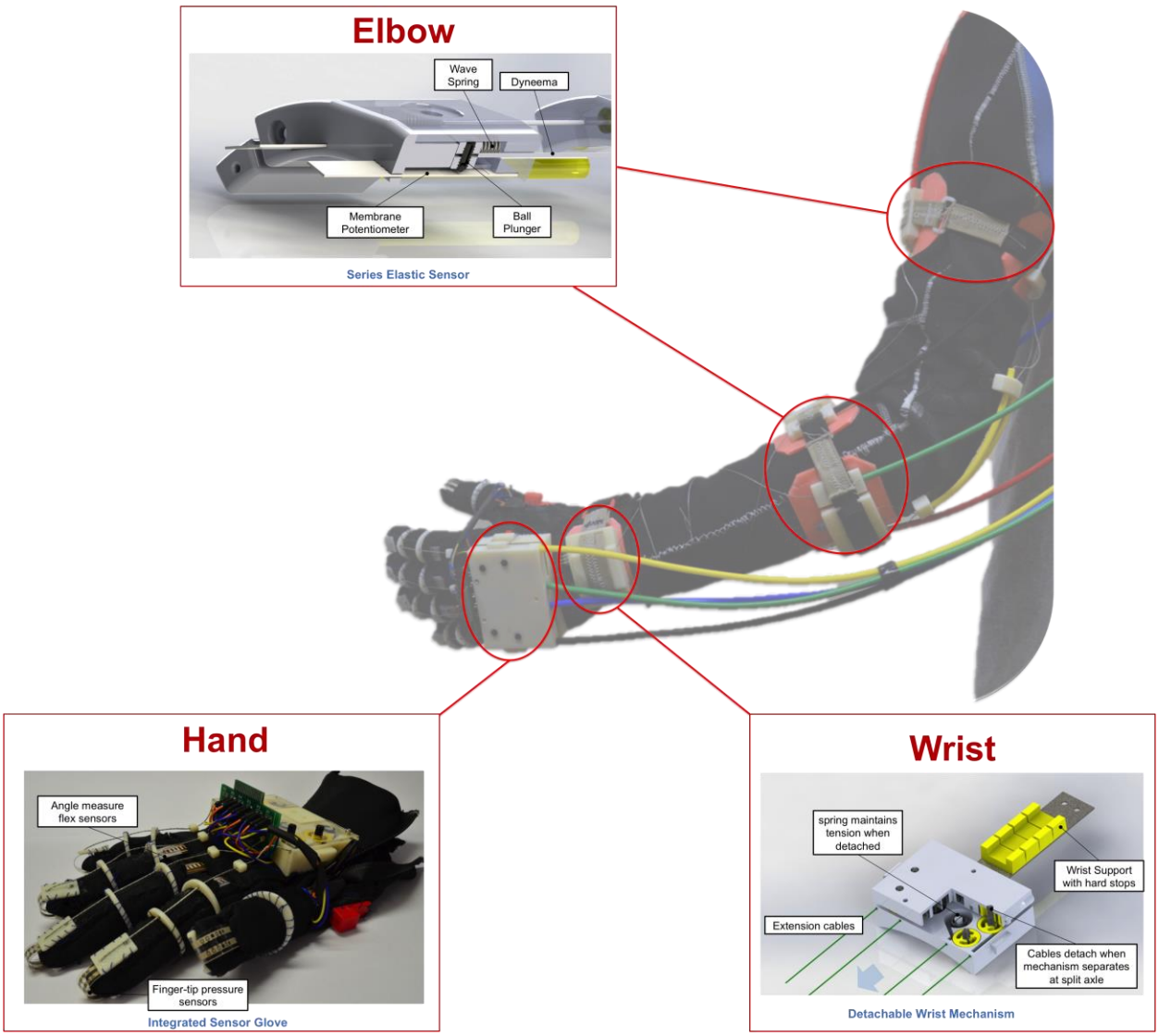


Figure 5.1.1: Mechanical System Overview

5.1.1 Glove Design

The glove was adapted from previous projects with modifications to support sensing and more advanced electrical components. The glove itself is a thin neoprene glove modified with two zippers running parallel from the base of the first and fourth knuckles to the top of the wrist, as seen in **Figure 5.1.2**, which allows for ease in putting the glove on a stroke patient suffering from decreased hand function. Along the back of each finger and sown into the glove

material is a pocket in which the flex sensors and leads for pressure sensors are be inserted, with their electrical connections available at the back of the hand. The flex sensors measure the angle of the first knuckle, and the pressure sensors measure any pressure present at the fingertip resulting from grasping objects or pressing on surfaces. The cables are attached to the glove via a series of two plastic rings fixed to each finger between the first and second knuckle the second and third knuckle, in addition to an end cap for the top and bottom of each finger.

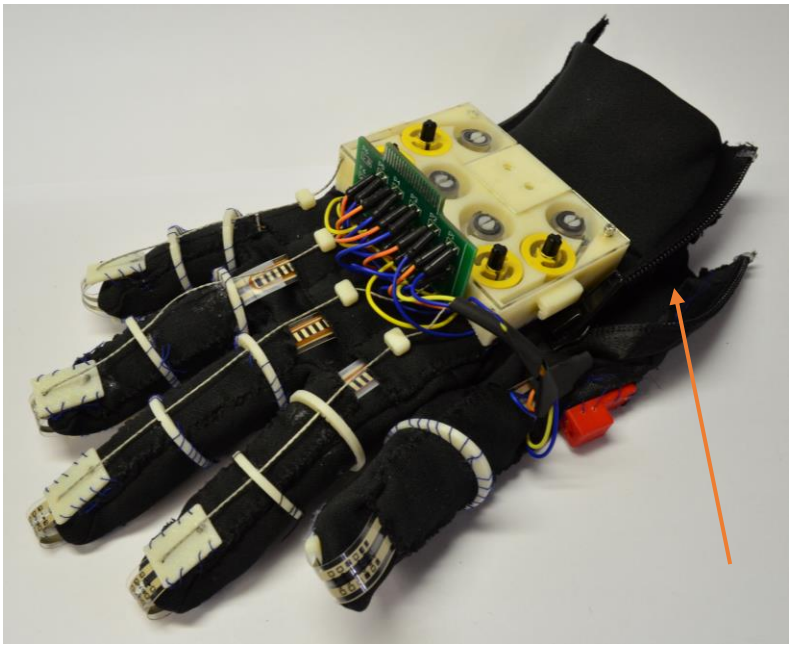


Figure 5.1.2: Glove with Two Zippers

The rings are an elliptical shape with a circular hole in the center. Centered in each of the resultant thicker areas are small holes. Through these holes the cable filament is threaded and glued. The rings are made of ABS plastic, and were created using a 3D printer. These rings are an advancement on designs in previous projects, which utilized two crescent shaped pieces which were glued to the top and bottom of the finger. In the previous design, being that the top and bottom cables were linked through the glove, rather than a solid piece of plastic, the glove

tended to stretch when tension was applied to the cables. This stretch caused much of the energy and mechanical advantage applied to the cables to be lost in the glove simply stretching and not moving the finger. The revised design solved this problem, while still being thin enough on the sides of the finger to avoid interference with the rings on other fingers. The final design for the finger rings may be seen below in **Figure 5.1.3**.

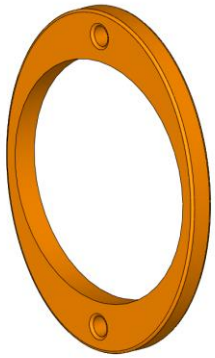


Figure 5.1.3: Plastic Ring Cable Guide

Designing the rings rigidly to solve the problem of the glove stretching caused another problem: the lack of the glove stretching. While inhibitive to the function of the device, the stretching of the glove between the two attachment points was what had previously maintained the adjustability of the design by allowing the glove to stretch around users with different sized fingers. To maintain compliance with this design goal the rings were said to be adjustable in manufacturability. Prior to this conclusion, the rings were planned to be fitted to the dimensions of person wearing the glove to keep them from being too tight or too loose. This was done by taking a series of measurements of the width of the knuckles of the person who would be wearing the glove. These measurements were then entered into a spreadsheet which governed the dimensions of the Solidworks model of the rings. Through a series of calculations the entire model was built based each knuckle measurement, and an entire hand's

set of rings were created for an individual automatically. Since the measurements are simple to take, and the rings are simple to produce using a 3D printer, it was established that any patient receiving this product could have a custom-fitted handmade rather simply, preserving the design's compliance with the adjustability design goal. The design table used for calculations can be seen below in **Table 5.1.1**. By entering the measurements taken into the column labeled "Finger Diameter" a series of relations calculates the proper size for each other dimension, which are then inserted into the model which created the properly sized ring for each finger, which are represented by each row in the table. The dimensions within the finger ring model which correspond to the table can be seen below in **Figure 5.1.4**. Dimensions in the table are in inches.

Table 5.1.1: Finger Ring Design Table

	Finger Diameter	Finger Width@Sketch1	Keep at .18@Sketch1	Keep at .05@Sketch1	D1@Boss-Extrude1	$PRP@Text Value	.18/2+width/2@Sketch3	math@Sketch3	Letter Height@Sketch6
Index-Middle	0.67	0.82	0.18	0.05	0.09	A	0.5	0.5	0.455
Index-Proximal	0.78	0.93	0.18	0.05	0.09	B	0.555	0.555	0.51
Middle-Middle	0.67	0.82	0.18	0.05	0.09	C	0.5	0.5	0.455
Middle-Proximal	0.75	0.9	0.18	0.05	0.09	D	0.54	0.54	0.495
Ring-Middle	0.67	0.82	0.18	0.05	0.09	E	0.5	0.5	0.455
Ring-Proximal	0.73	0.88	0.18	0.05	0.09	F	0.53	0.53	0.485
Little-Middle	0.63	0.78	0.18	0.05	0.09	G	0.48	0.48	0.435
Little-Proximal	0.67	0.82	0.18	0.05	0.09	H	0.5	0.5	0.455
Thumb	0.9	1.05	0.18	0.05	0.09	I	0.615	0.615	0.57

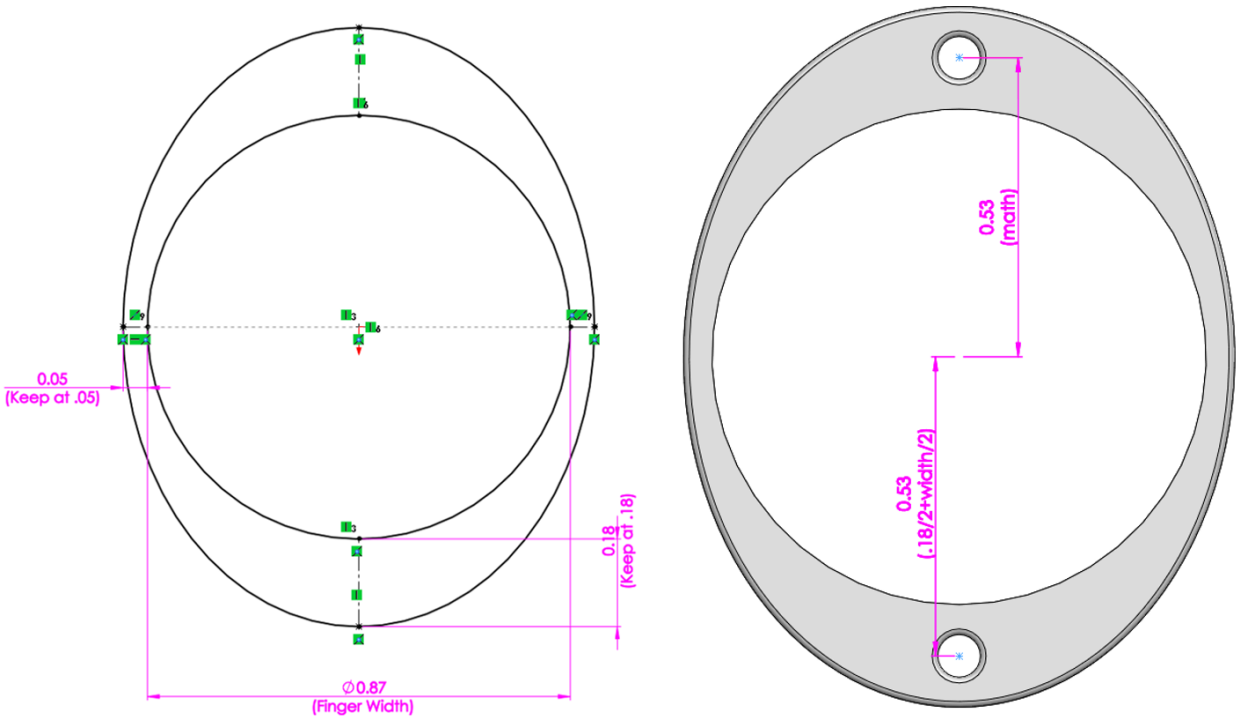


Figure 5.1.4: Finger Ring Dimensions

At the underside of each fingertip there is a clamshell mechanism which serves as the end cap for the cable running along the bottom of the finger, as well as forming a housing for the pressure sensors located at each fingertip. This device features a flat piece, which presents itself as the gripping surface of the fingertip and deflects when subject to a force from the user grasping an object. On the underside of this piece, and butting the pressure sensor, is a small protrusion which will interact with the pressure sensor. This protrusion creates a consistent and clean signal in the pressure sensor, as well as protects it from any damage that it would incur were it the main interacting surface. This flat piece joins with a platform to form a clamshell shape encasing the end of the pressure sensor. The base piece features a flat inlet the same size and shape as the pressure sensor, and an opening with a curved edge which allows the leads from the sensor to extend around the fingertip and back into the sewn groove of the

glove. At the point where the arm and base join there is a small trench which the end of the cable is inserted into and glued in the same procedure that the arm and base are glued together, solidly fusing all three together. Both pieces of the fingertip design were made of ABS plastic with a 3D printer. The final design can be seen below in **Figure 5.1.5**, **Figure 5.1.6**, and **Figure 5.1.7**.

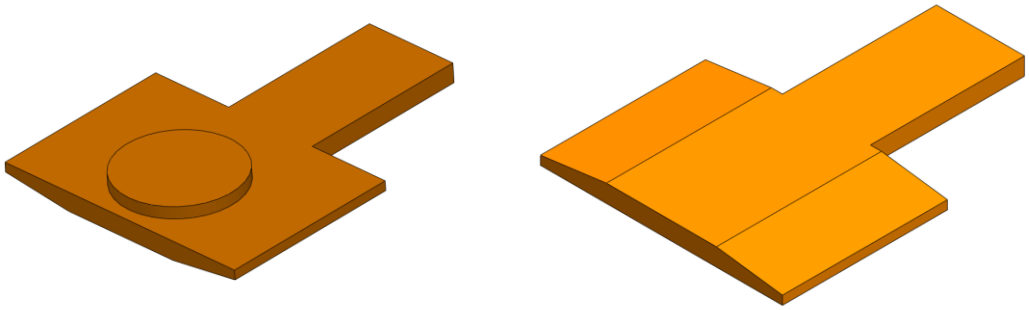


Figure 5.1.5: Fingertip-Arm

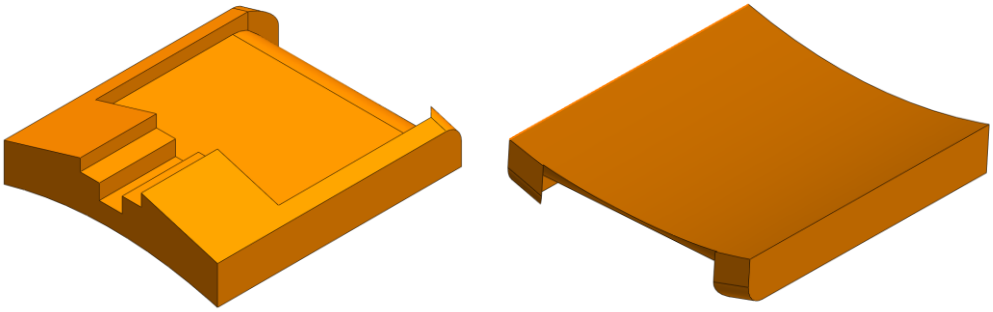


Figure 5.1.6: Fingertip-Base

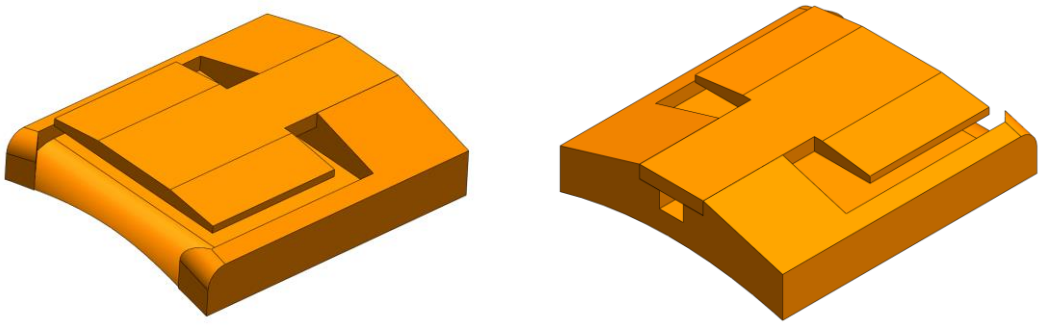


Figure 5.1.7: Fingertip-Complete

As for the back of the fingertip, the design used in earlier projects served fine and needed no improvement. It is simply a slightly curved plastic piece with a hole at each end. The cable was threaded through both holes and glued, before the whole piece was glued to the glove. This piece was made from ABS plastic in a 3D printer as well. The design for this piece can be seen below in **Figure 5.1.8**. Finally, an image of the completed fingertips of the glove can be seen below in **Figure 5.1.9.**

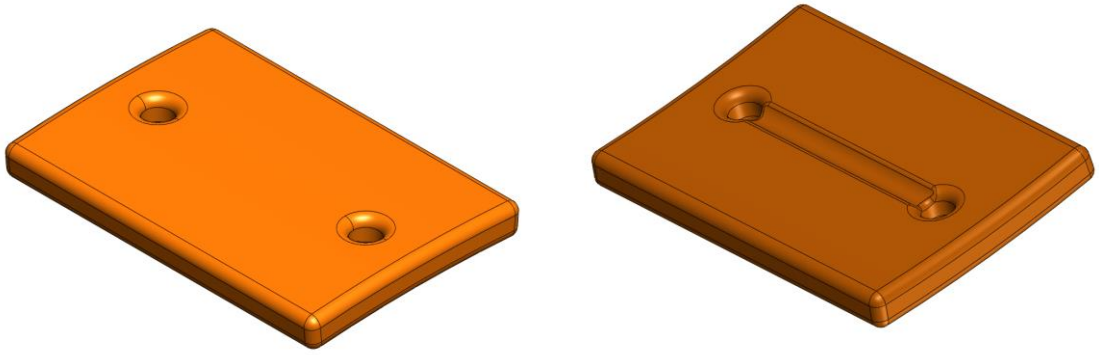


Figure 5.1.8: Fingertip Back

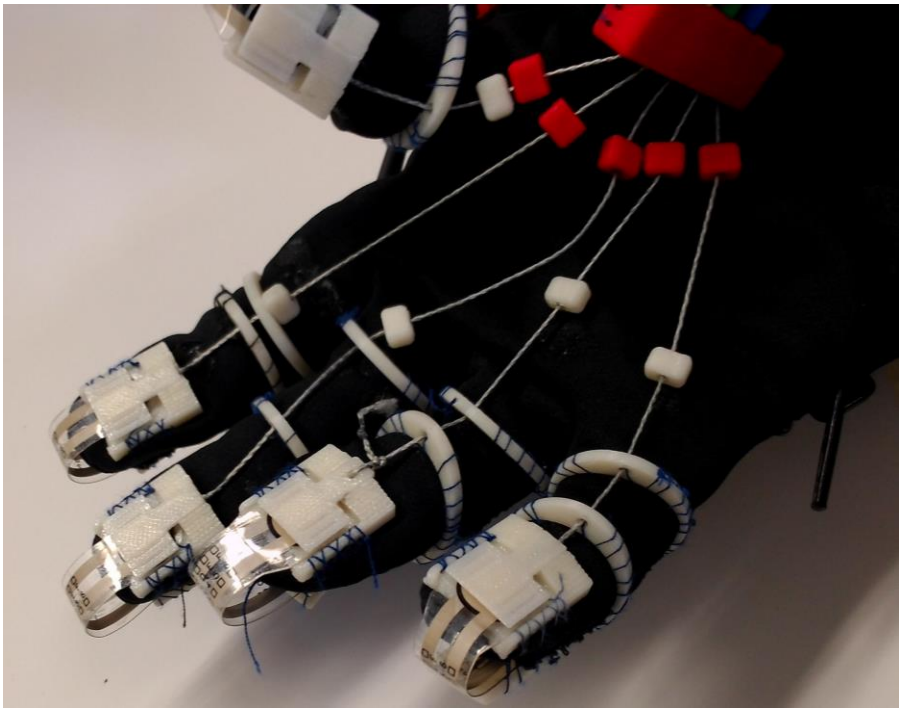


Figure 5.1.9: Completed Fingertips

The final iteration ensured the user could easily don the device while minimizing the number of mechanisms. The glove has zippers along both sides that opened the top half of the glove. Looking at the donning process at a higher level the process consist of a few simple steps. The user slips their arm through the sleeve while the glove is hanging out of the way. The top of the glove would unzip allowing the user's hand to slide into the glove. Finally, the quick release mechanism containing the upper metal wrist support, and the end of the finger extension cables running from the actuation platform would snap into place. Therefore the majority of the finger Bowden cables remain intact, and the four extension cables running on top of the hand would have a quick release mechanism. The bottom half of this device can be seen below in **Figure 5.1.10**.

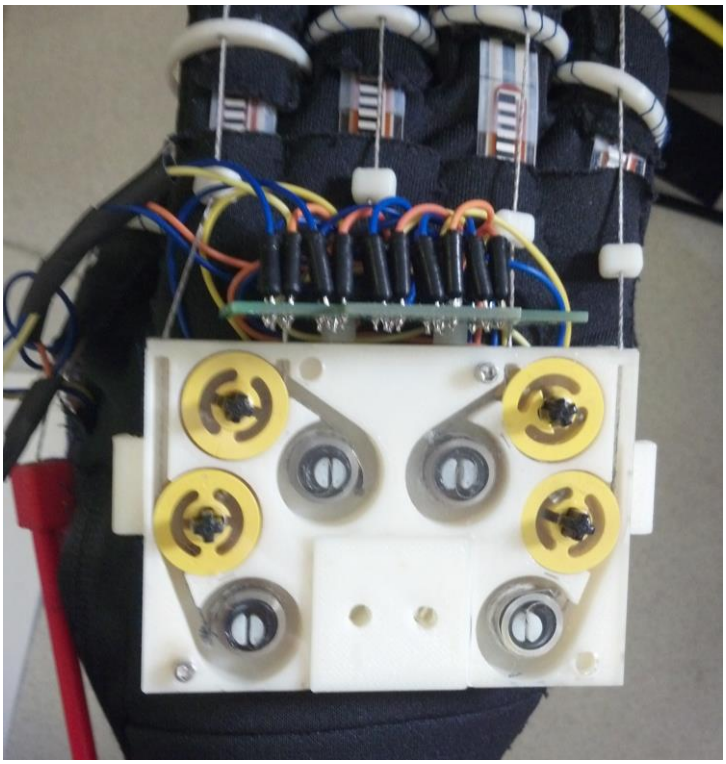


Figure 5.1.10: Attached Wrist Piece

Closer examination of the extension cable detaching mechanism reveals its inner workings. The cable anchored at the fingertips feed through the finger guides and wind onto the lower set of pulleys. The upper half of the mechanism would have an identical set of pulleys and a slot for receiving the axles from the lower pulleys. The upper pulley transfers the tension into the cable inside the connected Bowden cables leading back to the motor modules. This can be seen below in **Figure 5.1.11**.

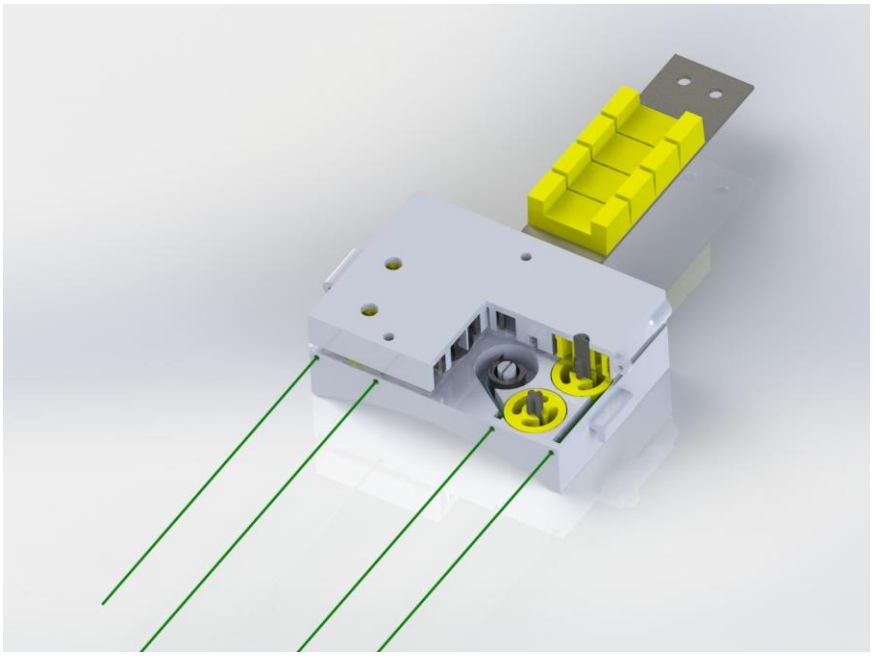


Figure 5.1.11: Upper Wrist Piece

Additionally, each pulley would be tension by a counter wrapped cable wound around a constant force spring. The tensioning is required when the mechanism is separated, and the pulleys are no longer connected to each other. The tension keeps the cables wrapped correctly on the pulleys. A model of just the pulley and spring system can be seen below in **Figure 5.1.12**.

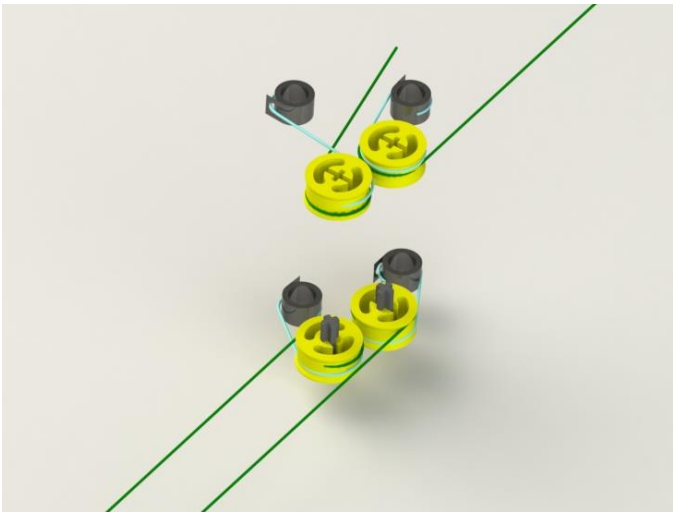


Figure 5.1.12: Pulley System

A view of the completed glove can be seen below in **Figure 5.1.13**.

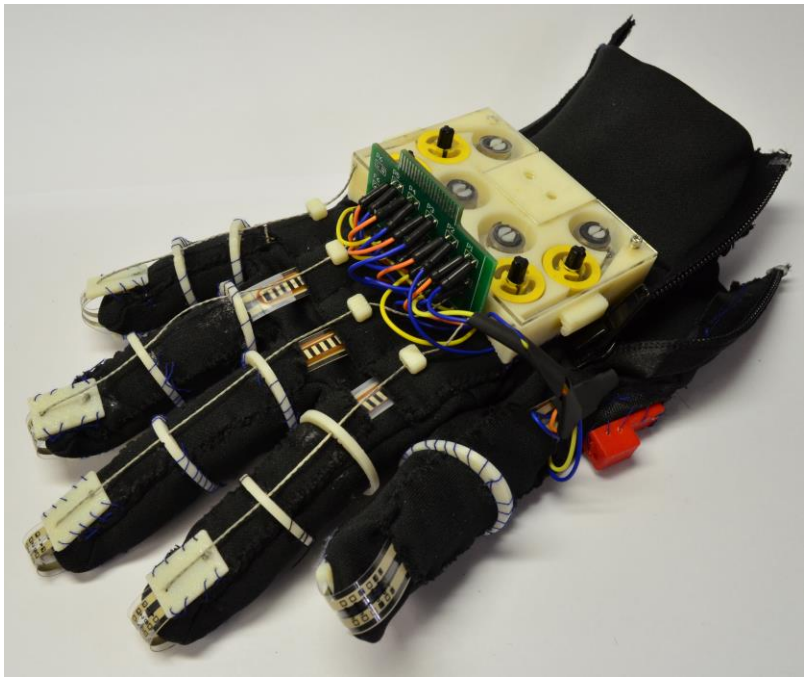


Figure 5.1.13: Final Glove

5.1.2 Wrist Design

Given that this device will be used in assistive applications, the wrist mobility of the human arm becomes significantly more important during rehabilitative motions. Analyzing many daily tasks where this assistive device could allow an individual to more autonomously perform chores, wrist supports the hand in grasping and carrying maneuvers. For example picking up an item that fell on the floor, the hand grasps the item while the wrist supports the hand in a neutral position to maintain grip.

Establishing, that the wrist portion of the team's device is primarily used to support hand gestures, the wrist was supported as opposed to being actuated like the fingers.

Needing some play for general movement the support must also prevent collapse under loading during rehabilitation or assistive tasks. To assure reliability and minimize the number of moving parts the wrist design incorporated an exoskeletal beam with hard stops to limit the

angle the wrist can bend. On the top of the wrist the thin metal beam provides spring like support for the wrist. Affixed to the surface of the metal beams are rows of trapezoids cut to a specified angle as seen below in **Figure 5.1.14** and **Figure 5.1.15**. This will allow the wrist to bend at a minimal angle, as mentioned in the design iteration chapter, with increasing resistance (provided by the surface mounted metal beam). Once the wrist reaches its limit angle, the trapezoids will meet and provide the hard stops.

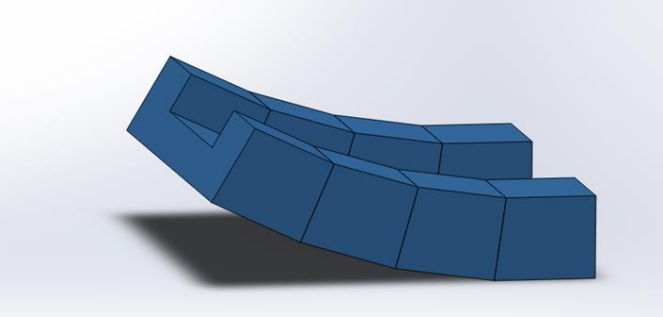


Figure 5.1.14: Row of Trapezoids Placed on Top Surface of the Wrist

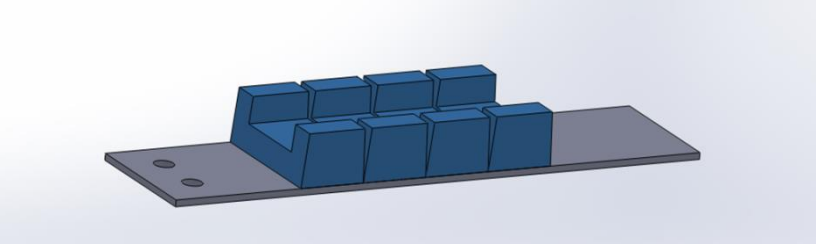


Figure 5.1.15: Row of Trapezoids Mounted onto the Metal Beam Attached to the Wrist

5.1.3 Elbow Brace Design

5.1.3.1 Stress Simulation

The material used for the elbow brace design is primarily ABSplus-P430 printed with the Stratasys Dimension 3D printer. Assuming a tensile strength of 39 MPa [55], SimulationExpress studies were perform using Solidworks.

Name		Type	Min	Max
Stress		VON: von Mises Stress	183.022 N/m^2 Node: 18365	1.45295e+006 N/m^2 Node: 20144

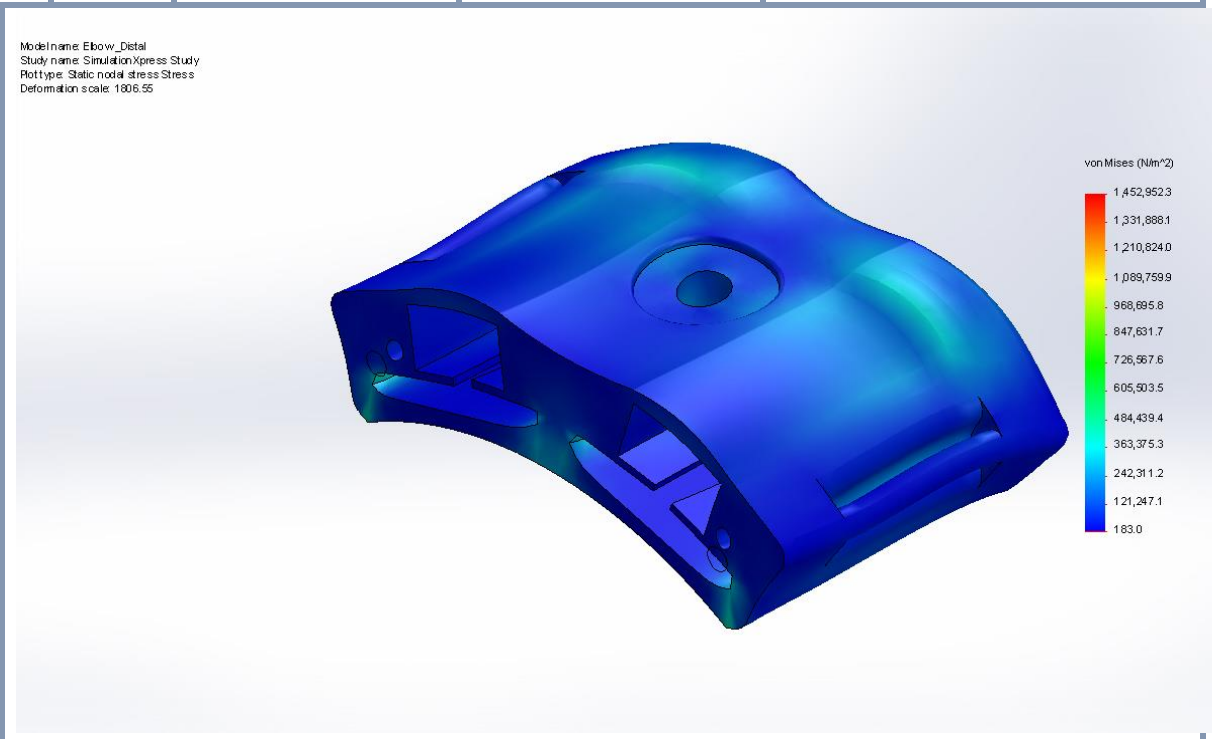


Figure 5.1.16: Lateral Stress Simulation

Figure 5.1.16 shows the results of the SimulationExpress study. In this simulation, the bottom curved surface of the part was considered fixed, while a total load of 75 N was applied to the interior back surface of the hollowed out space in the part. The force was applied in the

direction of the back of the part. This kind of loading is expected when the arm is fully extended. As is visible in **Figure 5.1.16**, the maximum stress reached is 1.4 MPa, which is 26 times less than the tensile strength of the material.

Name	Type	Min	Max
Stress	VON: von Mises Stress	50.2279 N/m^2 Node: 18373	1.85059e+006 N/m^2 Node: 8669

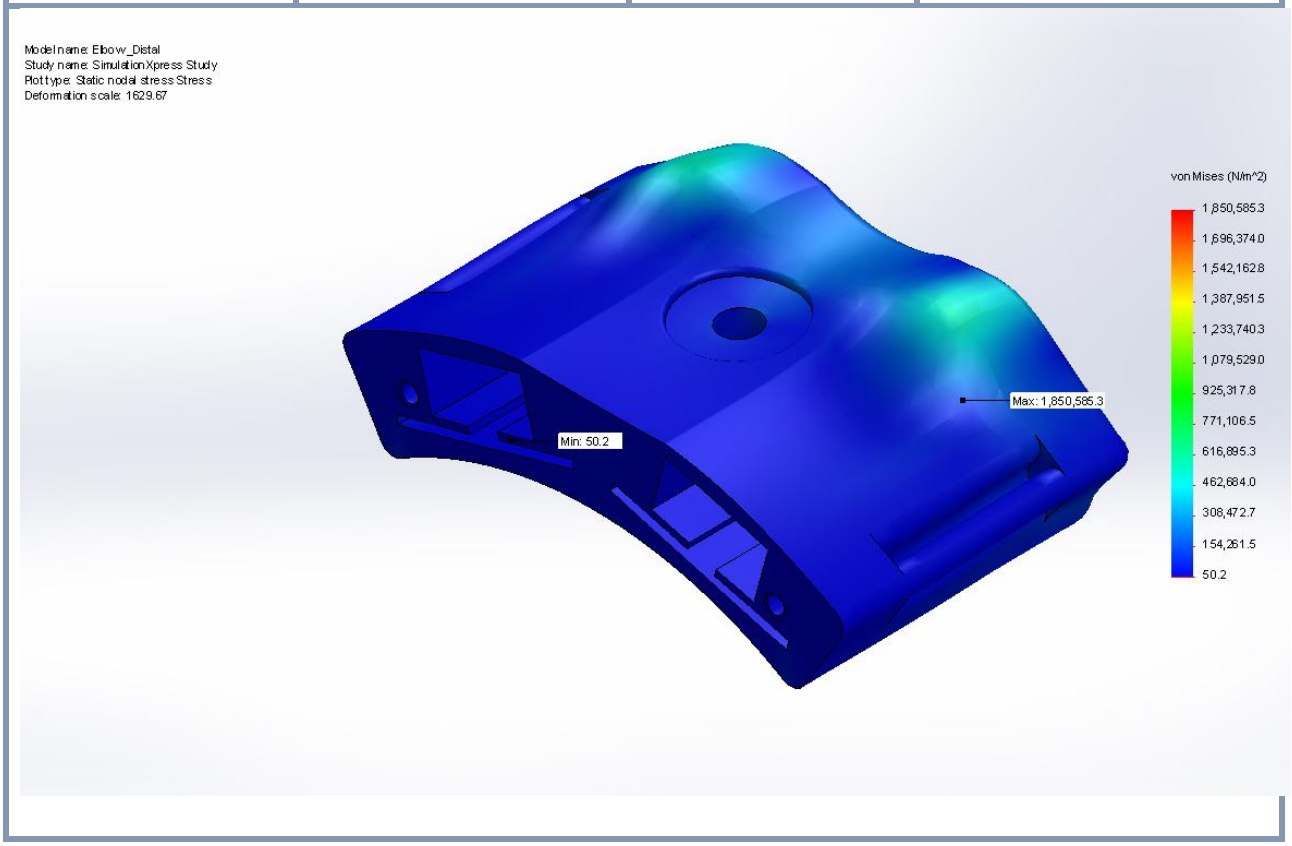


Figure 5.1.17: Vertical Stress Simulation

Figure 5.1.16 shows the results of another SimulationExpress study. In this simulation, the bottom curved surface of the part was considered fixed, while a total load of 75 N was applied to the holes through which the Dyneema pass. The force was applied in the direction of the top of the part. This kind of loading is expected when the arm is flexed at 90˚ while parallel

with the ground. As is visible in **Figure 5.1.17**, the maximum stress reached is 1.9 MPa, which is 18 times less than the tensile strength of the material.

5.1.3.2 Fabrication and Assembly

The majority of the brace parts were printed using ABSplus-P430 on the Dimension 3D printer. As a result, the surface finishes of the parts were not ideal for sliding parts. In addition, features generated in Solidworks, such as threaded holes made using the Hole Wizard, did not propagate through to the STL models and were thus not printed. Therefore, the ball plunger block required modification post-printing. The first step was sanding all sides of the ball plunger block. This not only improved the surface finish but also removed enough material from the sides of the ball plunger such that it fit within the track.

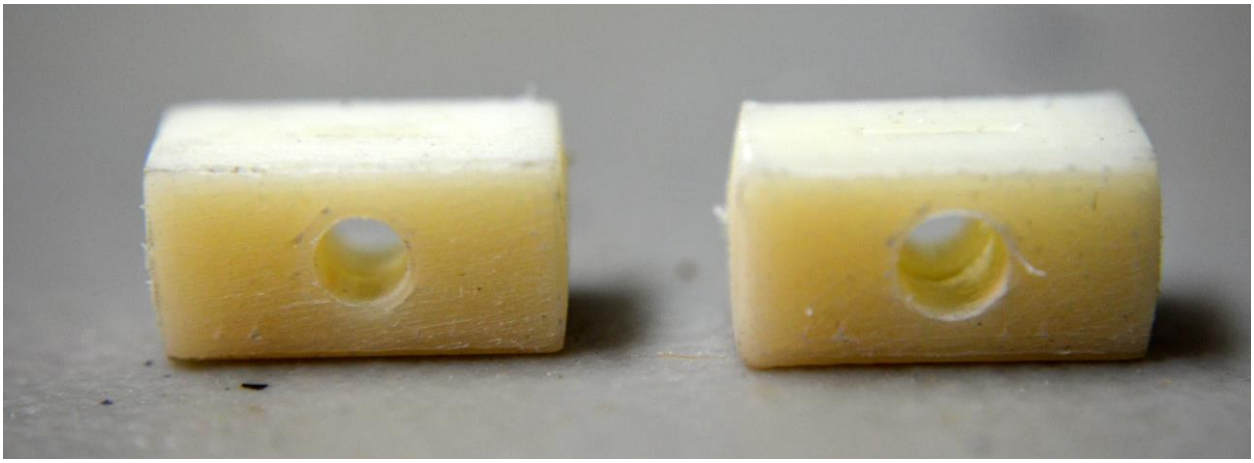


Figure 5.1.18: Modified Ball Plunger Blocks

As visible in **Figure 5.1.18**, in addition to sanding the faces, the hole in the ball plunger block must be enlarged to accommodate the ball plunger. One side of the block was drilled to a 1/8" diameter, while the other side was enlarged to 7/64". This allowed the side of the ball

plunger block with the larger hole to serve as a guide to more easily screw in the ball plunger. The narrower hole allowed the ball plunger to thread into the plastic.

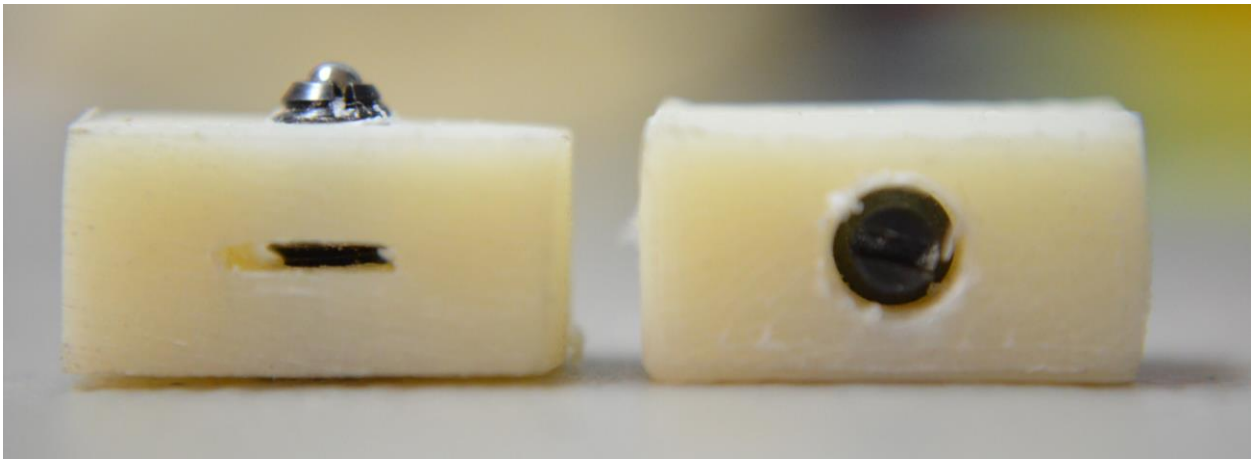


Figure 5.1.19: Ball Plunger Blocks with Ball Plungers Installed

Figure 5.1.19 shows two ball plungers installed in two different ball plunger blocks. The threading allows the distance between the end of the ball plunger and the bottom of the ball plunger block. The distance is used to set the force with which the ball plunger traces its path on the membrane potentiometer. Once all of the ball plunger blocks were assembled, Dyneema was tied to the ball plungers, and the blocks were installed in the brace parts.

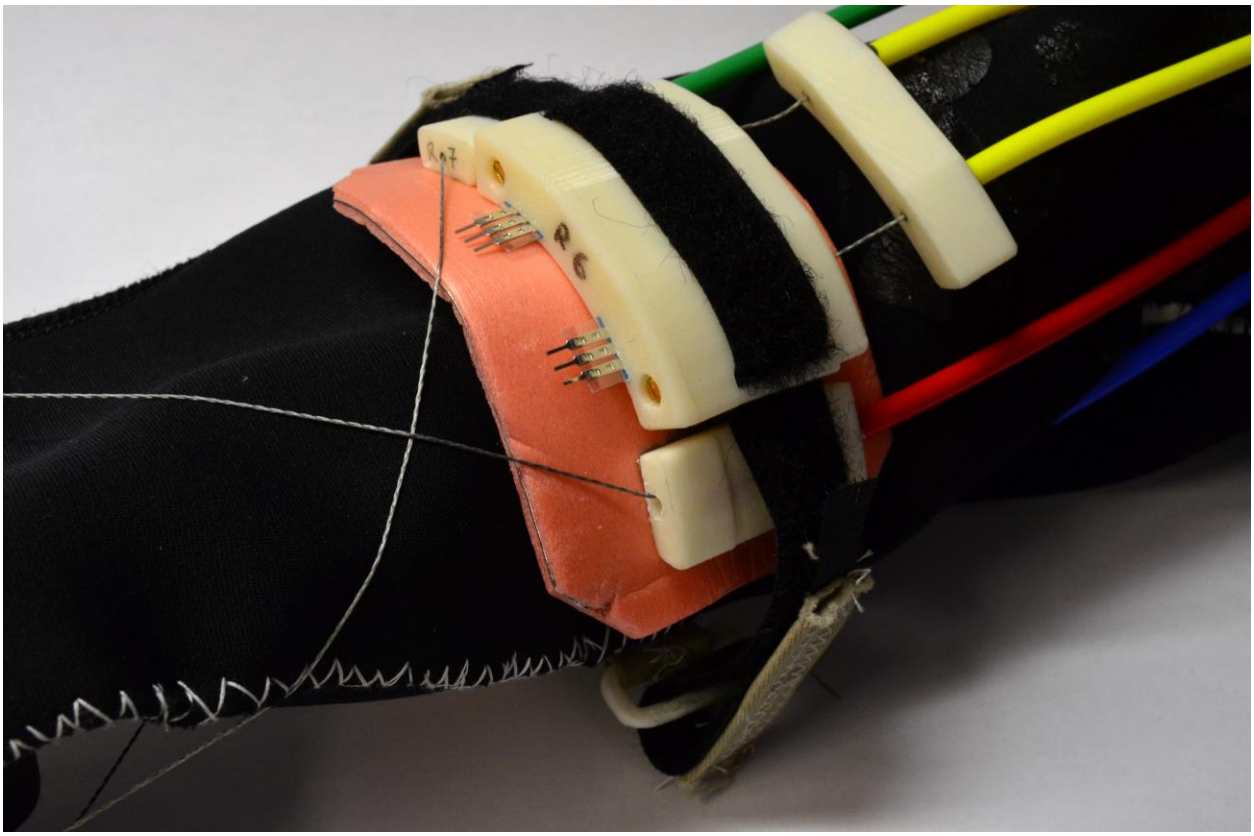


Figure 5.1.20: Assembled Brace Part

Figure 5.1.20 shows the brace parts attached with rivets to segments of padded aluminum splint. After the aluminum splint was attached, Velcro was added by passing strips through the cutouts in the brace parts. The Velcro loops strip (fuzzy Velcro) was passed through the slots to spread the loading of the Velcro strip across the entire brace part as opposed to concentrating the force on the relatively thin printed rods. The ends of the Velcro loops strips were then either sewn into plastic loops or sewn to pieces of Velcro hooks (rough Velcro strip). Once all of the brace parts were assembled, the segments were temporarily tacked onto the neoprene sleeve using Welder adhesive. When the locations of the brace parts were finalized, they were loosely sewn into the Neoprene sleeve, allowing for minor position adjustments.

5.1.4 Actuation Platform Design

One of the primary features of the device is the actuation of joints via non-local motors. The device is designed such that motors can actuate joints they are not directly adjacent to. Remote actuation is accomplished by housing the motors on a platform separate from the user with spools attached for winding cable in a Bowden system of cables. The cables connect to the joints being actuated causing them to move. The design of the Bowden system causes minimal force to be experienced along the length of the tubing while still causing actuation at the ends.

A particular challenge faced by the design team is that both the hand and elbow projects done previously had used a backpack to house the motor units, though each project had a separate backpack. The difficulty was in fitting the motor units for both elbow and hand actuation, as well as the wrist. Due to the limitations of the project, the backpack design was ultimately deemed a lower priority than demonstrating functionality for such an early-stage prototype and converted into a tabletop platform for testing and demonstration of the actuation and sensors. The tabletop platform can be seen below in **Figure 5.1.21**.

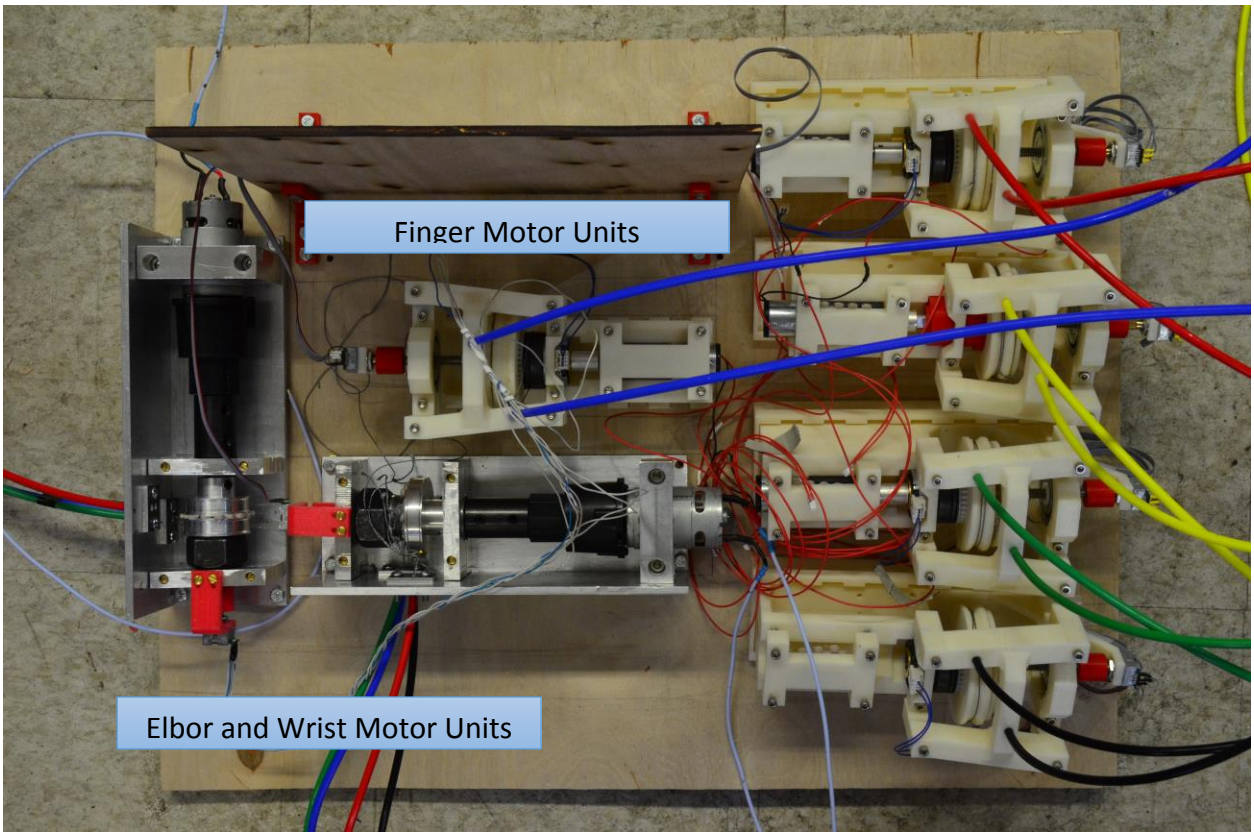


Figure 5.1.21: Actuation Platform Top View

This change would allow the team to develop a functioning prototype which could be easy to modify and maintain, without dedicating excessive resources to size compression. Regarding the mobility design goal, with further revising and optimizing of the design the motor units could be condensed to fit into a single backpack, but for the purposes of testing it was not seen as a priority.

Given that they were adapted from separate projects, the fingers and elbow portions of the sleeve were adapted differently to the established design requirements. This separation was further supported by the fact that both sections have rather different requirements both in performance and durability. That is to say that the components powering actuation of the elbow and wrist must endure much greater forces and strains that those for the fingers. As a

result, the motor units used to actuate each were developed separately and ultimately resemble each other very little.

5.1.4.1 Finger Motor Units

Being that the fingers do not require a great force for actuation, relatively small motors were sufficient. The motors ultimately used were the Maxon Motors A-Max model, which operated at 36V and 5W for a nominal speed of 110 rpm's. These motors were small, lightweight, and required power well within what the designed control system would be able to supply. Additionally, the high quality of the motors was such that they could be relied on for prototyping without breaking down or running into maintenance issues. An image of these motors can be seen below in **Figure 5.1.22**.

Figure 5.1.22: Maxon Finger Motor

The team decided that the materials located on the hand should be kept to a minimum to preserve the flexibility of the fingers. To this end it was decided that all methods of control for finger actuation should be located in the actuation platform with only the cables extending to the fingers. Electromagnetic clutches were added to the motor trains as a method of torque limiting. The clutches, which utilize electromagnets to variably press two discs together, allow

precision control over the torque being transmitted across it by controlling the voltage being applied across the electromagnet. By using a PWM controller one can produce a known maximum transmittable torque regardless of the force being output by the motors. This provided a simple method to exercise torque control on the actuation of the fingers. The clutches selected are by Ogura Industrial Corp., operate at 24 V, and are compact enough to comfortably fit into the motor units. A manufacturer's photo of the selected clutch can be seen below in **Figure 5.1.23**.

Figure 5.1.23: Clutch MIC-8NE

Furthermore, in creating a comprehensive sensor suite for the software to be able to interpret, encoders were added to measure the position of the spools. However, due to the clutch causing the spools to not be permanently fixed to the motors, the encoder had to be attached to the spools, rather than the more traditional method of attaching them to the motor. The encoders selected were CUI Inc. Panel Mount Encoders which operated at 5V and produced 32 ticks per revolution of the attached shaft with x4 quatrature. Using an adapter, these encoders were attached to the end of the threaded rod, and so were linked directly to

the rotation of the spools. The encoders themselves were held onto the motor unit with an extension of the base motor block piece. An image of the encoders can be seen below in **Figure 5.1.24**. The method for attaching the encoder to the motor unit can be seen below in **Figure 5.1.25**. The encoders allowed for the sensor board to know the exact position of the spools, as well as what direction they are traveling.

Figure 5.1.24: Encoder

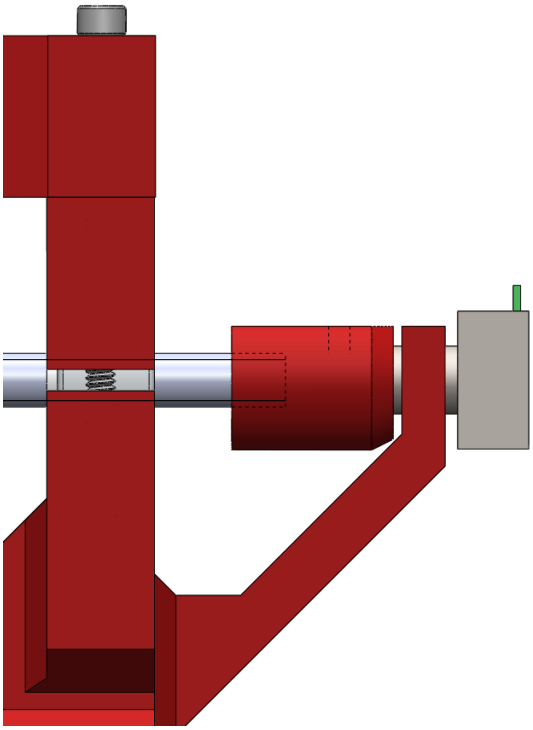


Figure 5.1.25: Encoder Attachment

Finally it was decided that the motor units would be best designed as modular units independent of each other. Previous projects had motors and spools attached and interwoven with the superstructure of the backpacks with varying degrees of complexity. Though this arrangement increased stability, it made the motor trains more difficult to revise and maintain. By designing each motor unit as a self-contained module, they would not be reliant on the superstructure of the actuator mounting platform to function, and therefore would require fewer changes to overall to revise later into the project, a critical feature for such an early-stage prototype. Taking advantage of the modular design, since the units were independent of the actuator mounting platform and each other, they could be easily removed if necessary with little interference to the other components. To this end, each motor unit was designed with a quick-release base featuring sliding tabs that interacted with opposing tabs on the motor mounting block to allow for the whole unit to be quickly slid in and out of place. Additionally, with the tensioning system mentioned earlier, modular units as described featuring the quick-remove base would make it easier to access the lock-nut used for loosening the spools.

Since overall the finger motor units would be subjected to relatively little force, and the nature of each unit was internally rather complicated, it was decided to create the base units using a 3D printer and ABS plastic. ABS, the same material that LEGO's are made of, is easy to form into complex shapes using a 3D Printer, and is quite strong when hardened. The various components were held in place on this framework by using clamps formed from ABS. The clamps were held in place using simple hex-head screws fitting into threaded inserts on the opposing piece. The threaded inserts were held in with Loctite Thread Locker glue. An

annotated image of the finger motor units can be seen below **in Figure 5.1.26**: Annotated Finger Motor Train. An image of a completed finger motor train can be seen below in **Figure 5.1.27:** Finished Finger Motor Train**.**

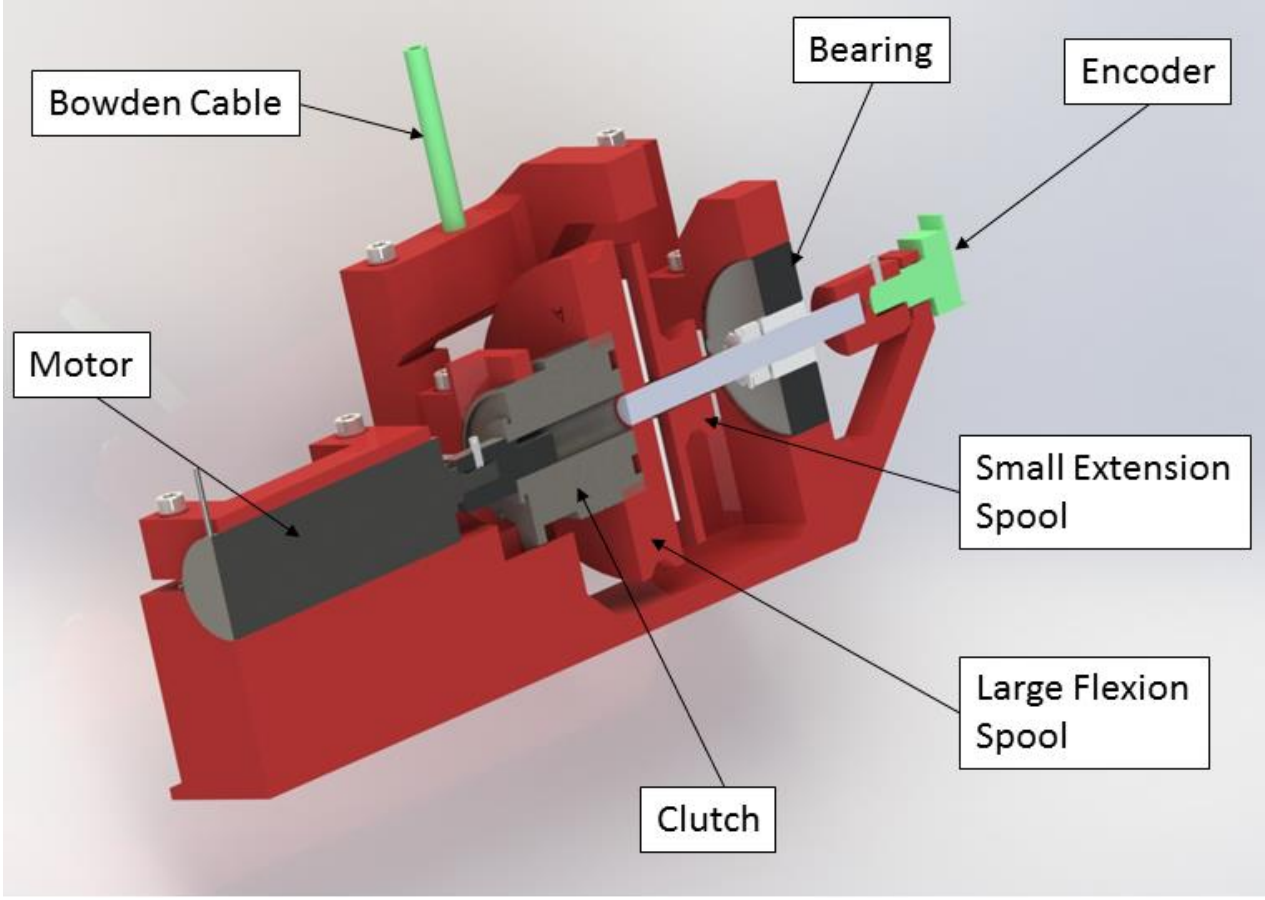


Figure 5.1.26: Annotated Finger Motor Train

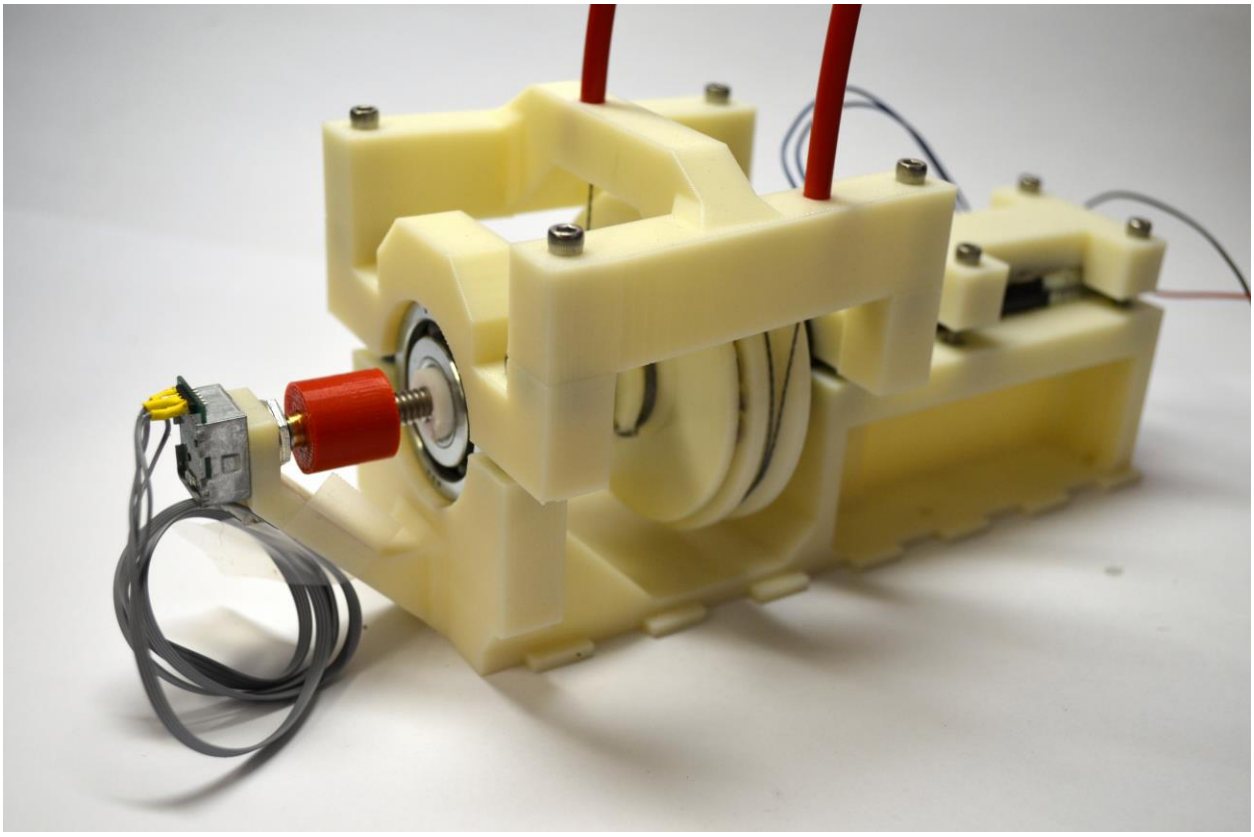


Figure 5.1.27: Finished Finger Motor Train

5.1.4.2 Elbow and Wrist Motor Units

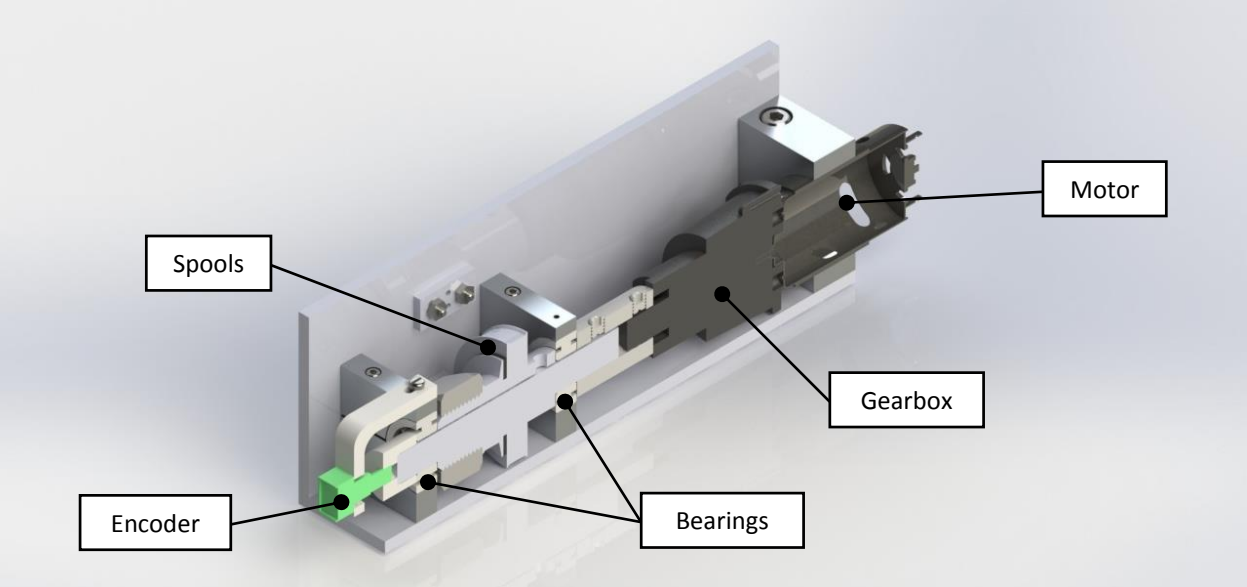


Figure 5.1.28: Annotated Final Elbow Module

Figure 5.1.28 shows the final CAD model of the elbow motor unit along with the names of all the critical components. The parts for the elbow and wrist motor units were fabricated with the use of both manual methods and CNC assistance. All of the machined parts were made of aluminum. Most of the manufacturing that would have proved difficult to manually fabricate was performed on CNC mills and lathes. The difficult processes included turning the two threaded larger spools, the two smaller spools, and cutting the larger holes in the motor and bearing retainer blocks. The L-channel and Bowden retaining blocks were completed with the use of manual machine tools. The motor and bearing retaining blocks and spools all required additional manual machining to be completed.

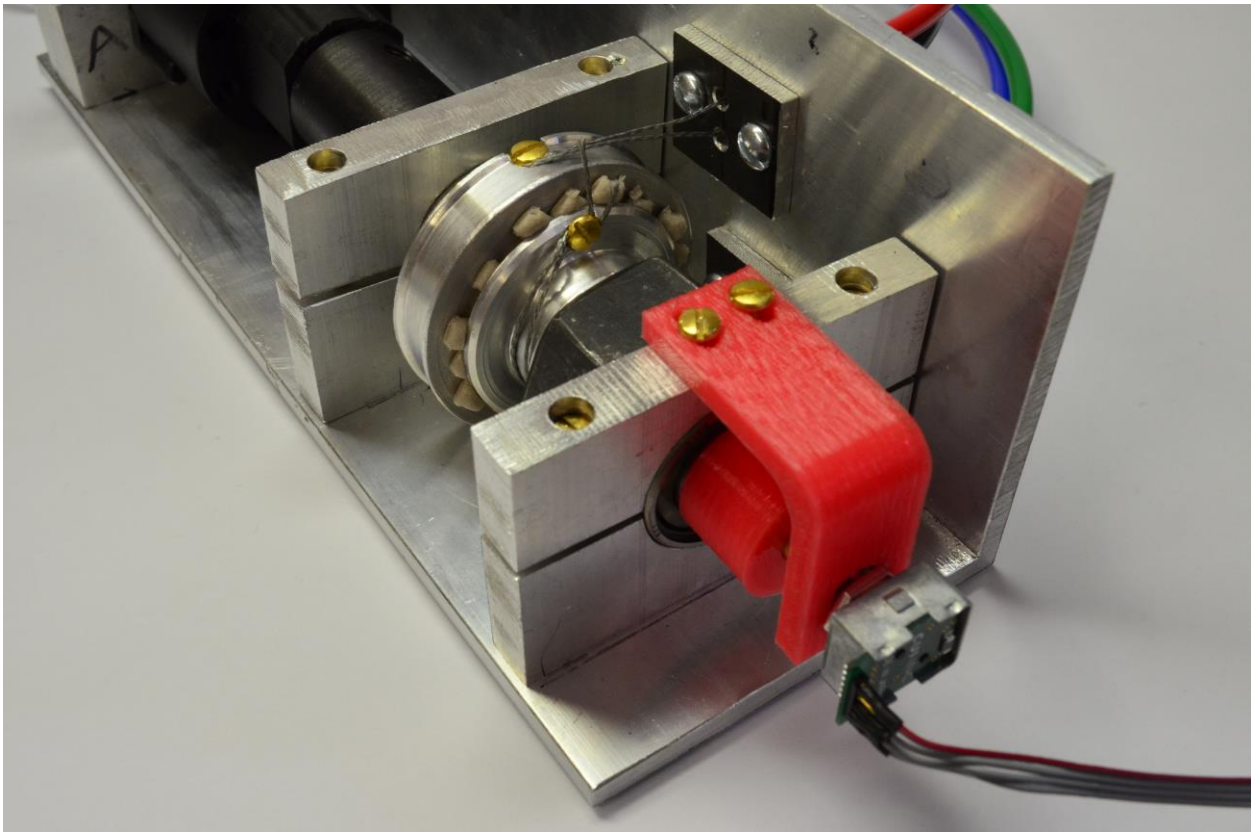


Figure 5.1.29: Assembled Elbow Motor Unit

Figure 5.1.29 is a close-up view of the elbow motor unit. A 3/8” end mill on a manual mill was used to machine flat sections on each of the spools. The flat sections allowed for the machining of #4-40 tapped holes. These holes were then used to place machine screws for fixing the ends of the Dyneema cables.

5.2 Electrical Subsystem

As described in the previous chapter, the electrical system is divided into four different subsystems, each with its own custom printed circuit board (PCB). Each PCB has its own microcontroller, which is programmed for its specific tasks and includes specific components for each task. The Main board is directly connected with the other Sensor and Motor boards as well as a computer running Ubuntu. All PCBs were designed using the Altium Designer 10

software package by Altium Limited (Australia). The three developed PCBs can be seen below in **Figure 5.2.1**. A high-level functional diagram of these circuit boards and their communication with the computer was shown before in **Figure 5.2**.

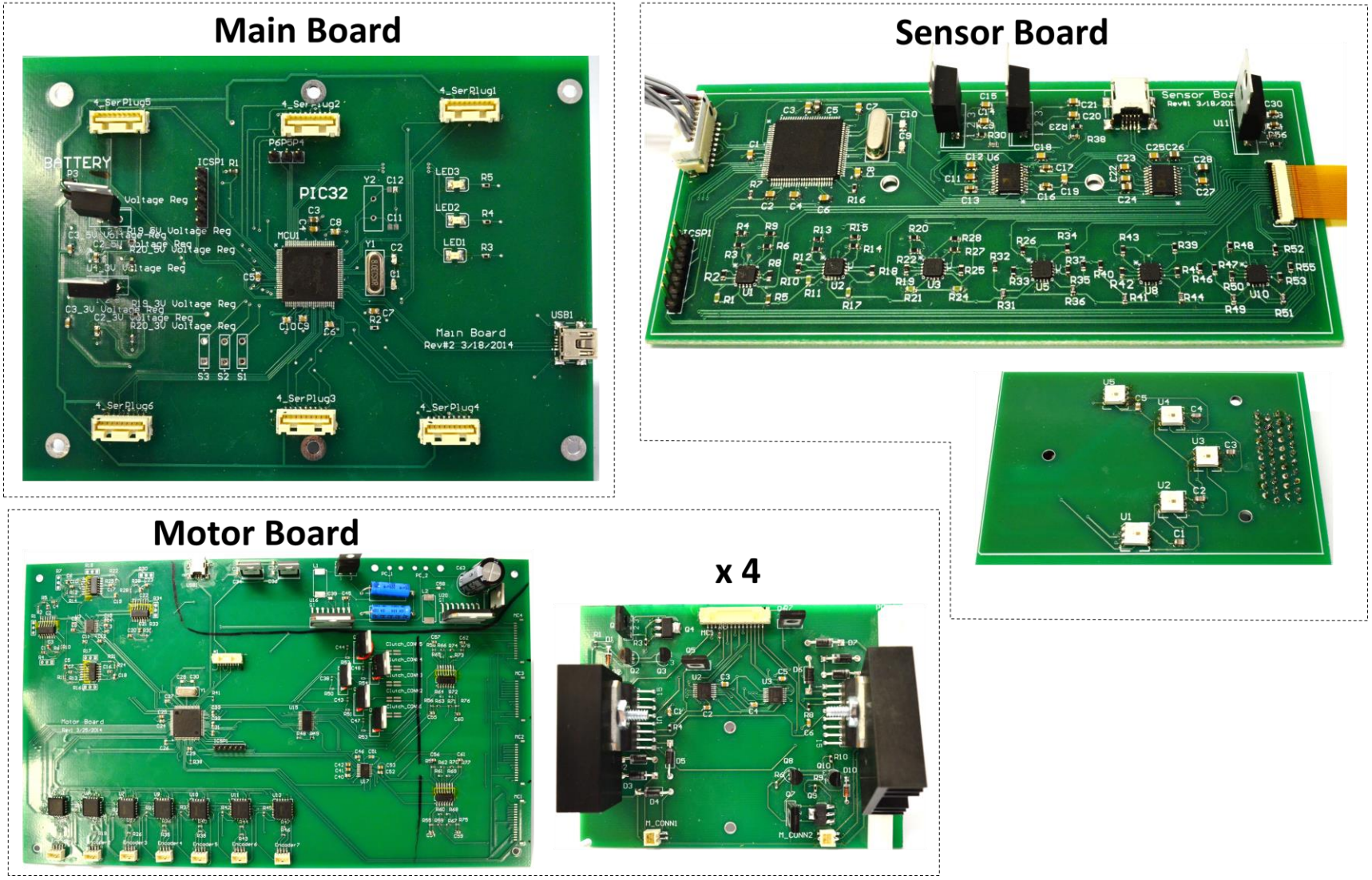


Figure 5.2.1: Fully Assembled Printed Circuit Boards included in the Electrical Subsystem of the Exomuscular Sleeve.

5.2.1 Main Board

The Main board, which is placed in the actuation platform of the device, is the master of the system, the one that requests data from other sub-systems, transfers the data from one sub-system to another, and communicates with the computer. The Main board consists of a PIC32MX795F512L chip, six 9-pin connectors for UART, battery lines, connectors for ISCP and USB, voltage regulators, and other general I/Os. PIC32's UART interface is used to transfer data between each PCB while ICSP is used to program the PIC32 chip mounted on the Main board. Lastly, USB is used to transfer data from PIC32 to the computer. A high-level functional diagram of the Main board is shown below in **Figure 5.2.2**. The high-level Altium schematics and PCB of this board can be seen in **Appendices A** and **B**, respectively.

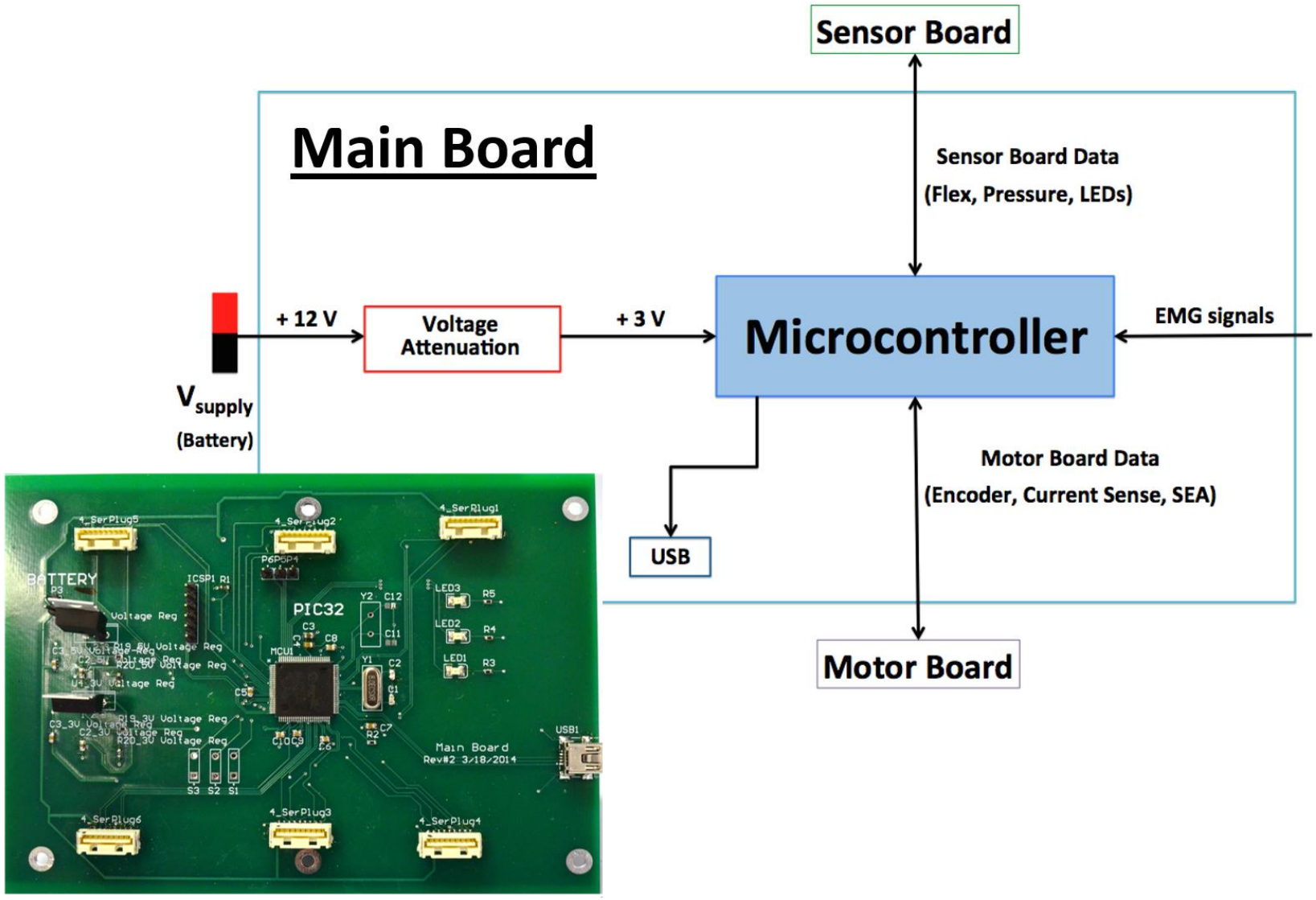


Figure 5.2.2: Main Board High-Level Functional Diagram and Fully Assembled PCB.

5.2.2 Sensor Board

The Sensor board is in charge of controlling all the sensors included at the glove. As mentioned before, there are 5 flex sensors and 5 force sensitive resistors located on the fingers, as well as 1 flex sensor on the wrist. The flex sensors are used to measure the angle of flexion/extension of the fingers and the wrist while the force sensitive resistors are used to measure the force applied by the fingertips. Since all of these sensors are located on the hand, the Sensor board is mounted on the forearm to ensure short communication lines dedicated for analog signals. The Sensor board includes a PIC32 chip, ICSP, USB and an UART. As in the Main board, ICSP is used to program the PIC32 microcontroller, USB is used to communicate to the computer and UART is used to communicate to the Main board. The Sensor board also includes the Wheatstone bridge sensor circuits with instrumentation amplifiers, Butterworth 2^{nd} order low-pass filters, analog-to-digital converters (AD7927), and voltage regulators which were all discussed in detail in previous chapters. All the signals from the sensors come in from the connectors and are then fed to the sensor circuits. The signals are then conditioned using the low pass filters before feeding them to the ADC. Furthermore, the ADC has a built-in SPI interface, which is used to send the data to the PIC32 for signal processing. This board also controls the feedback LEDs. Since the LEDs chosen have built-in integrated circuits, SPI data are required to turn them on or off and to change colors. This board provides power, ground, clock and SPI data to the LEDs' integrated circuits. Each of these steps can be seen as a single block in the high-level functional diagram of the Sensor board shown below in **Figure 5.2.3**. The high-level Altium schematics and PCB of this board, along with enlarged pictures of the different components included in it can be seen in **Appendices C** and **D**.

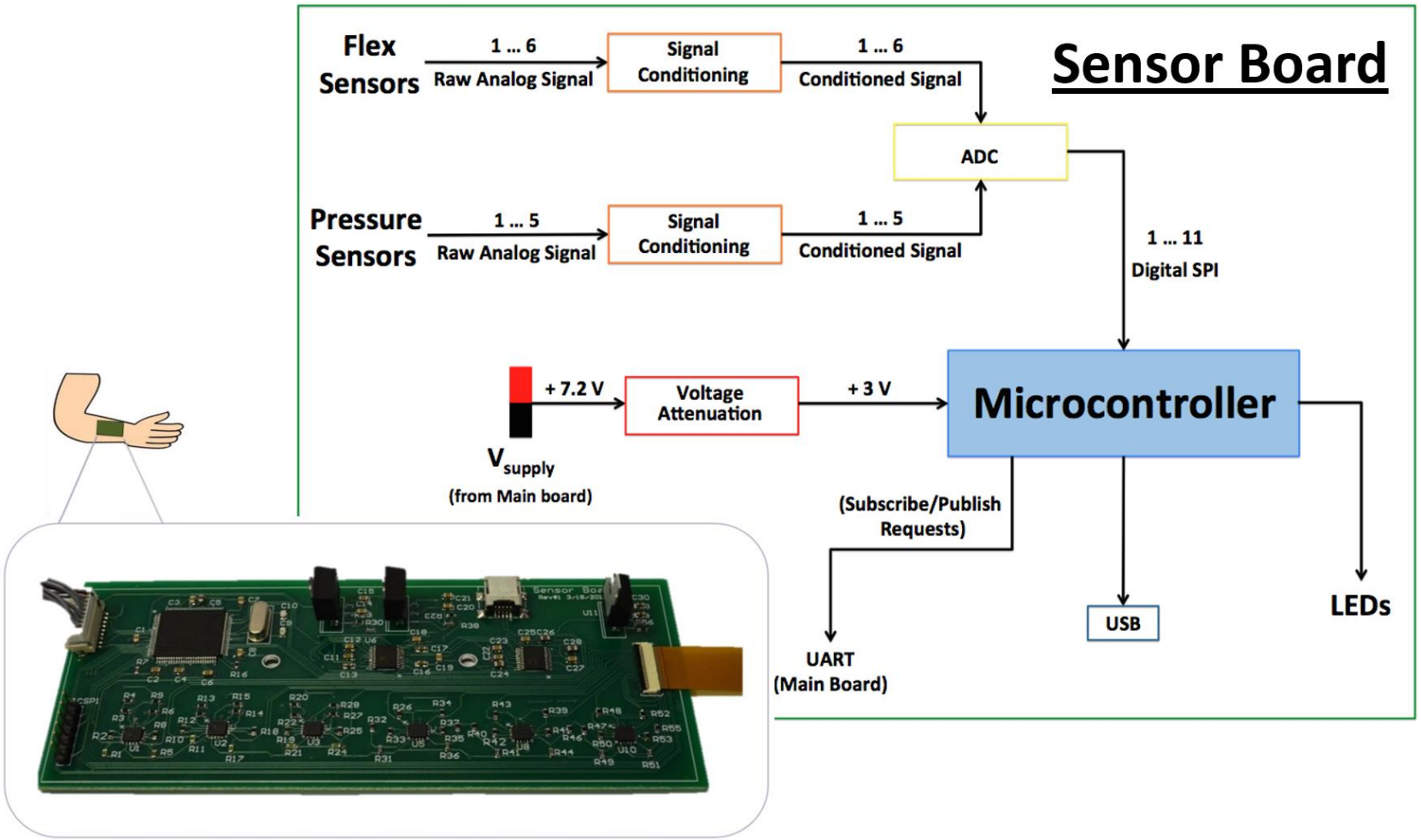


Figure 5.2.3: Sensor Board High-Level Functional Diagram and Fully Assembled PCB.

5.2.2.1 Glove Sensors

The final glove design includes a total of 10 sensors: 5 flex sensors to measure angle of flexion/extension of the fingers and 5 pressure sensors to measure force on fingertips. The five flex sensors are located on the knuckles of each one of the fingers. Additionally, the five pressure sensors are distributed among the fingers.

Flex and pressure sensors are capable of changing their resistance as they are bent or pressed respectively. The more flex sensors are bent, the greater their resistance value. On the contrary, the more pressure sensors are pressed, the smaller their resistance value. The flex and pressure sensors included in the glove are set up in a Wheatstone Bridge circuit in order to obtain differential signals that are further amplified via instrumentation amplifiers, as shown below in **Figure 5.2.4** and **Figure 5.2.5**.

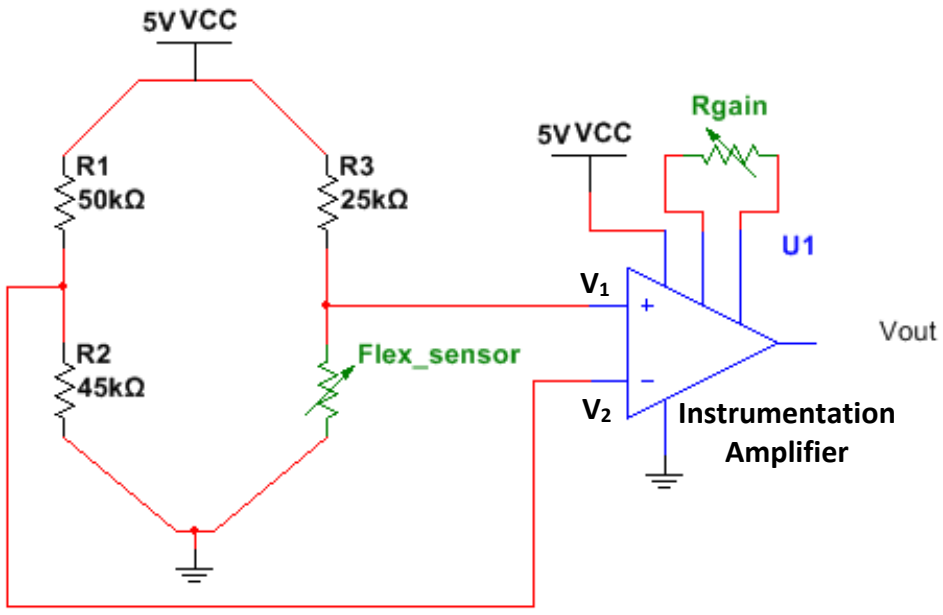


Figure 5.2.4: Wheatstone Bridge Setup for Flex Sensor

In this circuit, the input (V_1 and V_2) and output voltages (Vout) to/from the instrumentation amplifier are given by the following expressions:

$$V_2 = V_{cc}\left(\frac{R_2}{R_2 + R_1}\right)$$

$$V_1 = V_{cc}\left(\frac{R_{Sensor}}{R_{Sensor} + R_3}\right)$$

$$V_{out} = G\,(V_1 - V_2)$$

Substituting the expressions of the input voltages V_1 and V_2 into the output voltage expression V_{out}, the following general equation is obtained:

$$V_{out} = G \times V_{cc}\left(\frac{R_{Sensor}}{R_{Sensor} + R_3} - \frac{R_2}{R_2 + R_1}\right)$$

For the case of flex sensors, the positive input for the instrumentation amplifier (V_1) is the result of a voltage divider with a 25kΩ resistor and the flex sensor in series. The initial resistance of the flex sensor is 25kΩ with 25% tolerance. Similarly, the negative input for the instrumentation amplifier (V_2) is the result of a voltage divider with 50kΩ and 45kΩ resistors in series. This voltage divider along with the 5V supply is about 2.37V (5V * 45Ω/95Ω). Ideally, V_2 would be 2.5V and the difference between V_1 and V_2 would represent the output signal. But the tolerance is very high for these sensors. Therefore, in order to account for such high tolerance, V_2 is set to 2.37V instead of 2.5V, which was necessary to avoid negative Vout since the ADC cannot convert negative voltages.

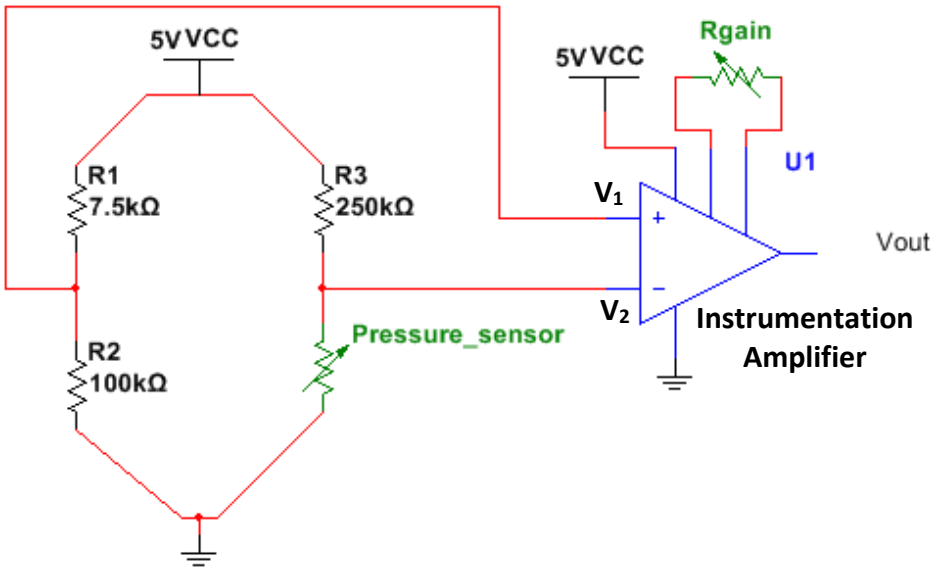


Figure 5.2.5: Wheatstone Bridge Setup for Pressure Sensor

In this circuit, the input (V_1 and V_2) and output voltages (Vout) to/from the instrumentation amplifier are given by the following expressions:

$$V_1 = V_{cc}\left(\frac{R_2}{R_2 + R_1}\right)$$

$$V_2 = V_{cc}\left(\frac{R_{Sensor}}{R_{Sensor} + R_3}\right)$$

$$V_{out} = G\,(V_1 - V_2)$$

Substituting the expressions of the input voltages V_1 and V_2 into the output voltage expression V_{out}, the following general equation is obtained:

$$V_{out} = G \times V_{cc}\left(\frac{R_2}{R_2 + R_1} - \frac{R_{Sensor}}{R_{Sensor} + R_3}\right)$$

For the case of pressure sensors, the negative input for the instrumentation amplifier (V_2) is the result of a voltage divider with a 250kΩ resistor and the pressure sensor in series. The initial resistance of the pressure sensor is infinite and decreases to 250kΩ when force is applied. Similarly, the negative input for the instrumentation amplifier (V_1) is the result of a voltage divider with 7.5kΩ and 100kΩ resistors in series. The result of this voltage divider along with the 5V supply is about 4.65V (5V * 100Ω/107.5Ω). Ideally, V_1 would be 5V and the difference between V_1 and V_2 would represent the output signal. But the resistance of pressure sensors changes from infinity to a certain value even with a slight touch. This change in resistance is monotonic and significant only after V_2 drops down to 4.7V. Therefore V1 is set to 4.65V.

The amplifier chosen was the AD8426 from Analog Devices, which has dual channel instrumentation amplifiers in one single chip. After testing the instrumentation amplifier, the voltage output results received were inconsistent with theoretical values. The sensors have high tolerance and resistors used were 5% tolerance and these are the reasons for the inconsistency. Data from the testing circuit are discussed in Results chapter and Appendix L. Therefore it was decided to use 1% tolerance resistors for circuits to improve design and get more accurate results. Furthermore digital pots were added to have variable gain and add flexibility to future implementations of the project. The digital potentiometer chosen for this application was the AD5262 from Analog Devices, which has 256 levels of resistance going up to 50K**Ω**. The gain for the instrumentation amplifier is given by following equation:

$$Gain = \frac{49.4\ k\ \Omega}{R_g} + 1$$

The circuits used for each sensor can be seen below in **Figure 5.2.6** and **Figure 5.2.7**.

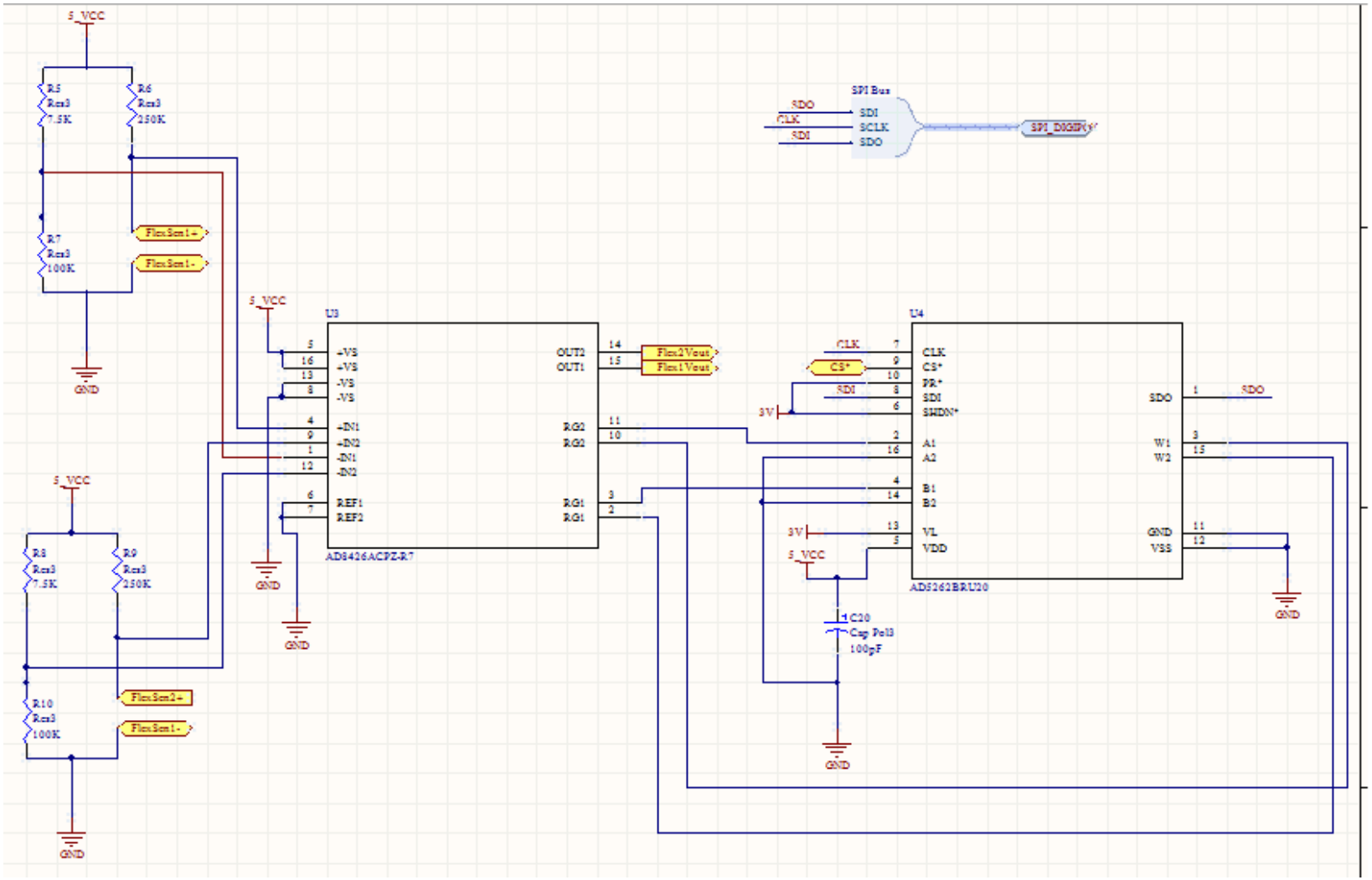


Figure 5.2.6: Flex Sensor Schematic

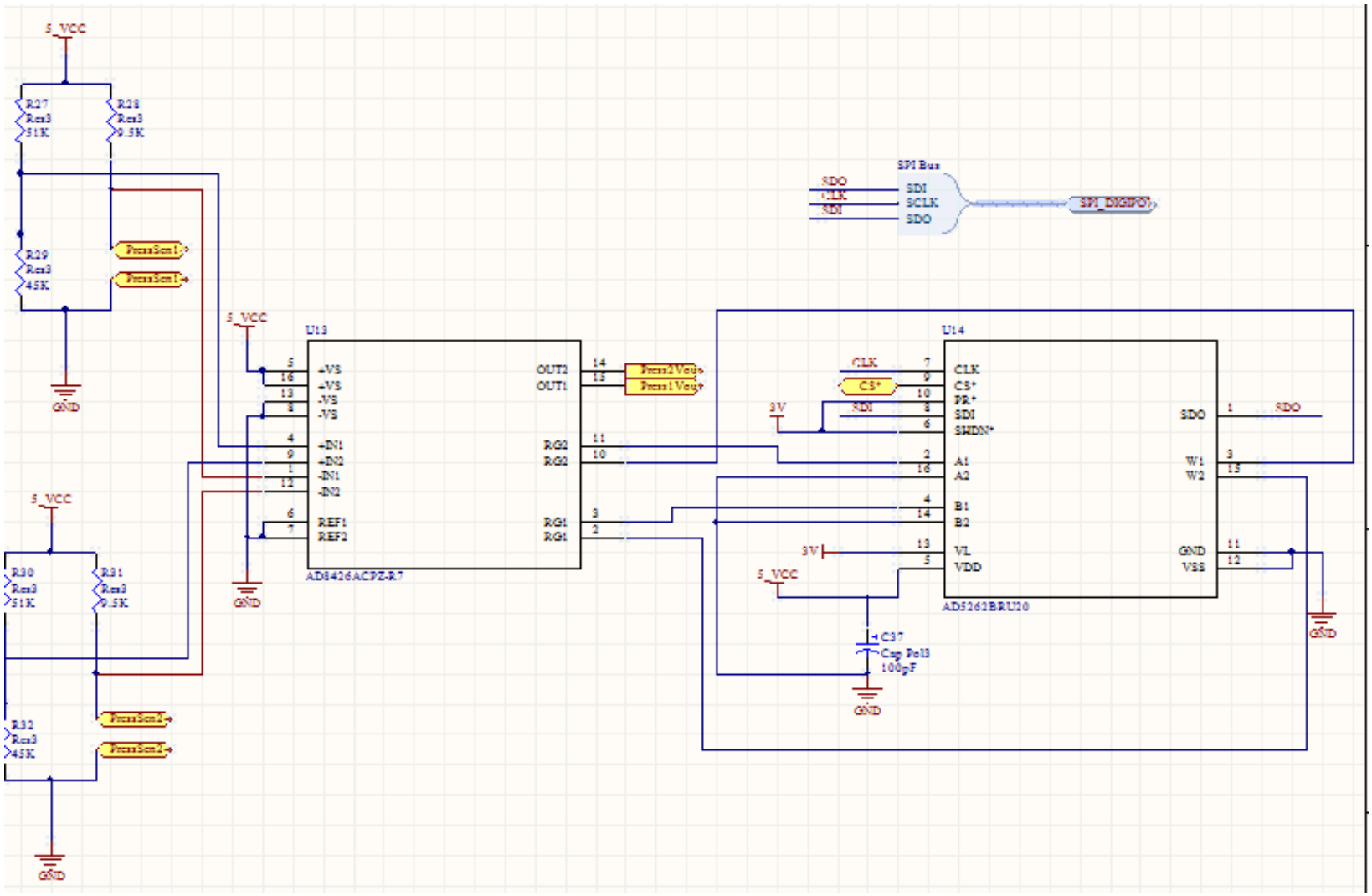


Figure 5.2.7: Pressure Sensor Schematic

The Sensor board also includes a series of LEDs to provide visual feedback to the user regarding the completion of the assigned rehabilitative tasks. The LEDs utilized in this system are capable of emitting red, green and blue light with the use of a built-in Integrated Circuit (IC). The three colors are used to indicate the following three stages of progress: red representing the farthest from the desired goal, yellow representing closer to goal, and green representing the completion of the assignment by the reaching the desired goal (which will vary for each exercise). The chosen LED was WS2812 from Worldsemi. This LED only requires a power supply, ground and one data line (Din). LEDs are controlled by the SPI data which can be daisy-chained through Dout to control multiple LEDs. The circuits used for the LEDs can be seen below in **Figure 5.2.8**.

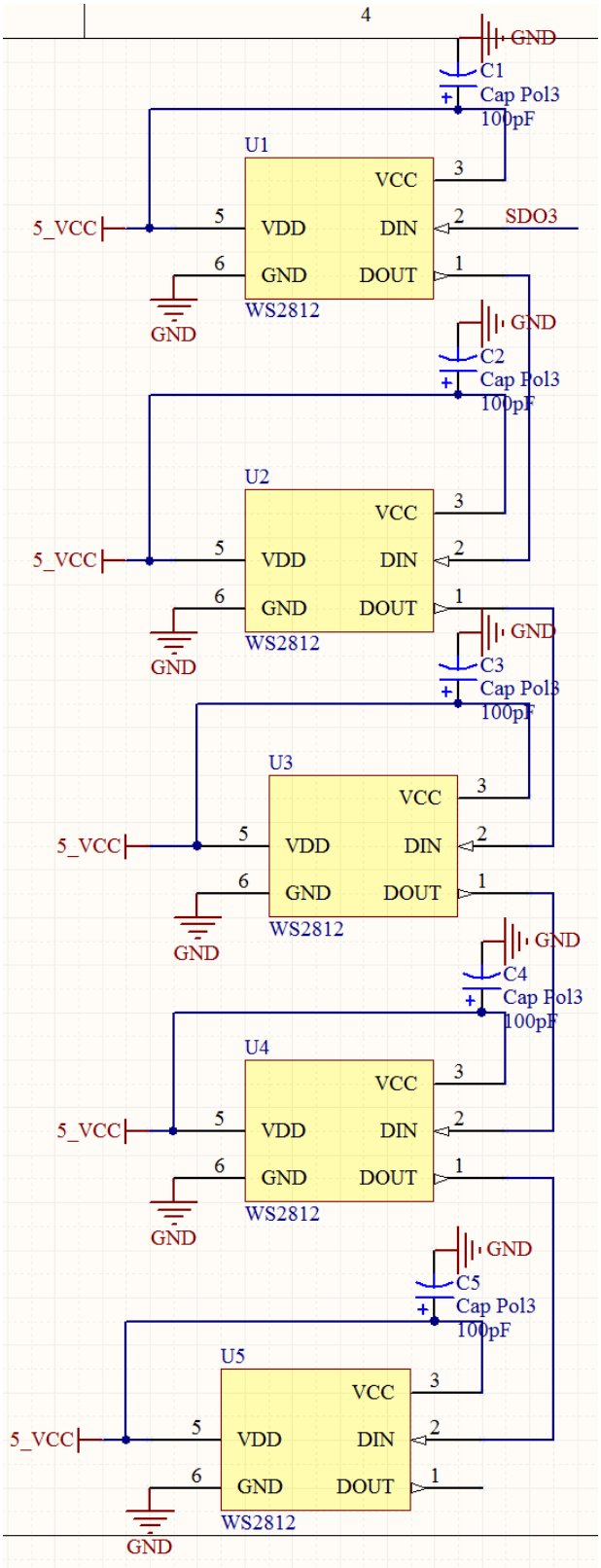


Figure 5.2.8: LED Schematic

5.2.2.2 Wrist and Elbow Sensors

An additional flex sensor was placed on the wrist structure to measure the angle of flexion and extension of the wrist, just as it was done with the fingers. A set of four SEAs and one encoder-decoder coupling for position control was included in both the wrist and the elbow structures. The data collected from the SEAs included in this system are filtered using anti-aliasing low-pass filters during the analog sampling phase to reduce noise. The low-pass filters utilized for this application are active Butterworth filters of second order with cutoff frequency of 26 Hz. The purpose of using low-pass filters is to remove aliasing during analog sampling and to possibly remove the 60Hz power noise. Aliasing is a phenomenon that occurs when the sampling rate is not high enough or signal filtering is not appropriate, meaning that samples leave a "shadow" that overlaps/interferes with each other in their spectrum (see **Figure 5.2.9**).

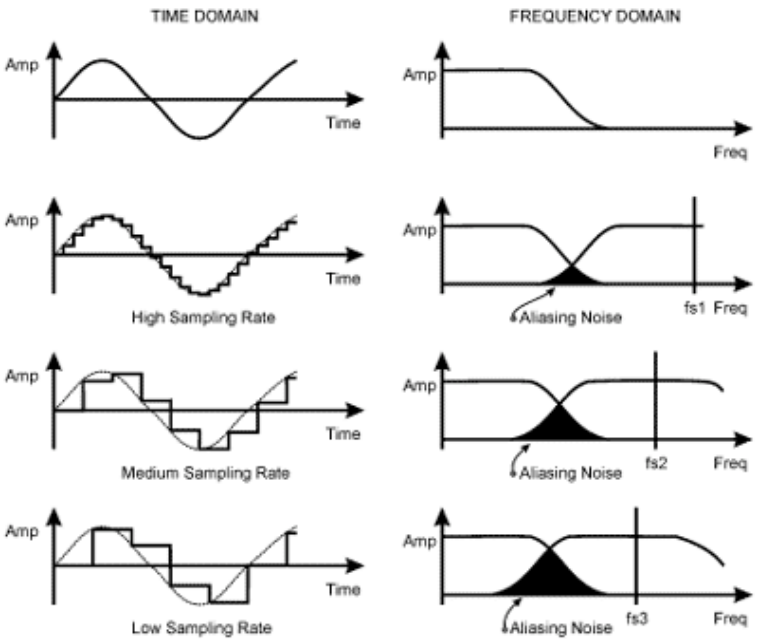


Figure 5.2.9: Signal Aliasing

Image retrieved from:
http://www.eetimes.com/document.asp?doc_id=1255124

The analog data are amplified in order to obtain a greater range of data from different resistor values. An operational amplifier was selected to ensure a high-impedance voltage follower since forward-flowing current was strictly undesirable. The final setup selected for all the SEAs and the low-pass filters can be seen below in **Figure 5.2.10**.

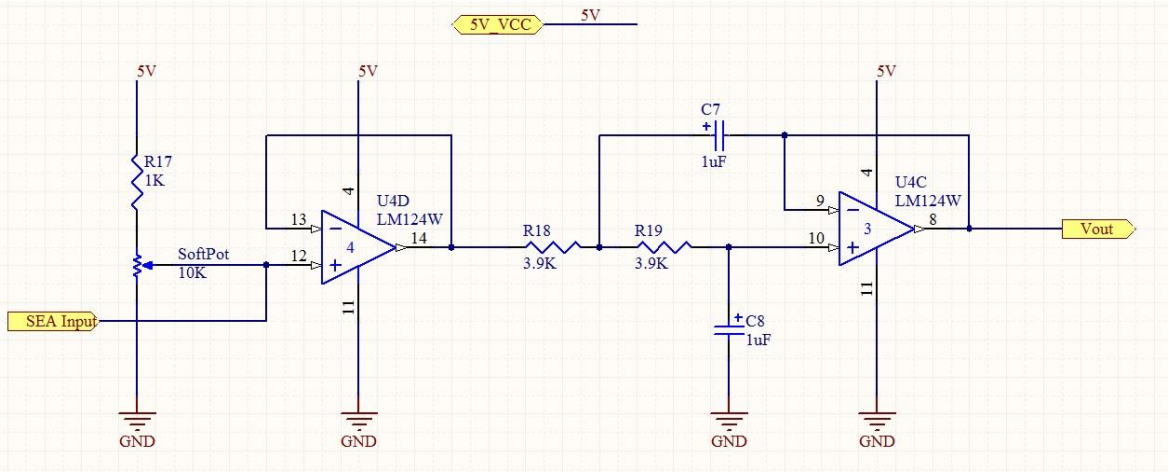


Figure 5.2.10: Magnetic Series Elastic Actuators with 2nd order Low-Pass Filter

5.2.2.3 Signal Conditioning

Sensor data are sent for analog signal conditioning by the unity-gain 2nd order Sallen-Key Butterworth low pass filter, which was introduced in the previous chapter. A final low pass design was implemented based on the calculated cutoff frequency (25Hz) with standard resistor values to achieve the frequency as close as possible. An active filter was preferred because it requires smaller inventory of consisting components with better signal to noise ratio (SNR) than passive filters (see **Figure 5.2.11** and **Figure 5.2.12**), which also increases common mode rejection ratio (CMRR) of the filter. The following calculations, which are based on transfer function of 2nd order unity gain Sallen-Key low pass filter [54], in order to determine the 25Hz cutoff frequency. A MATLAB code for evaluating and testing the transfer function can be found in Appendix L.

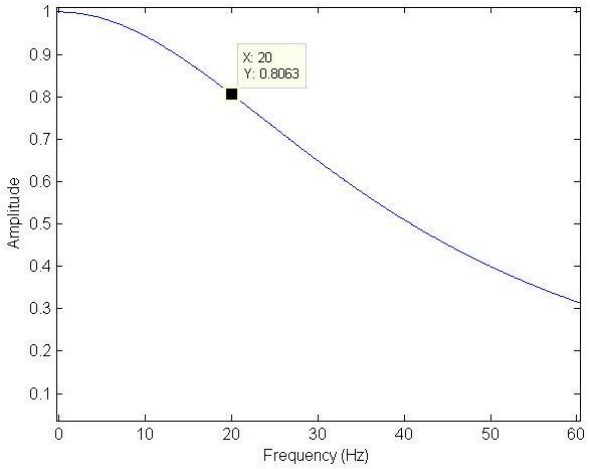


Figure 5.2.11: Sallen-Key Active Filter Frequency Response

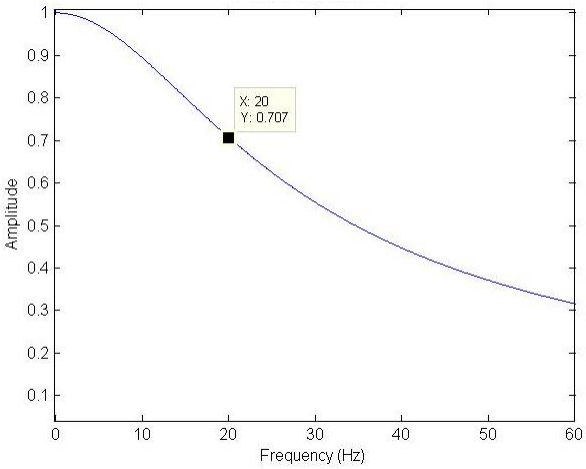


Figure 5.2.12: Passive Filter Frequency Response

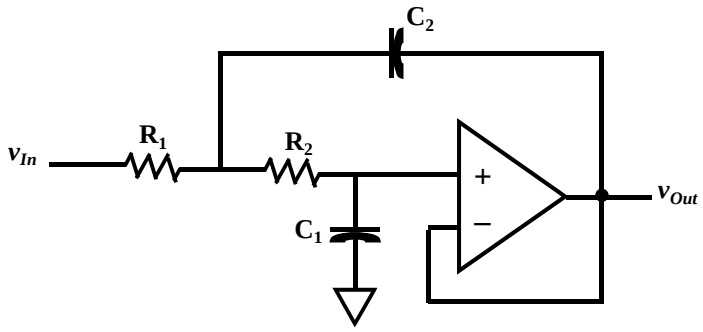


Figure 5.2.13: Unity-gain, Second Order Sallen-Key Butterworth Low Pass Filter [58]

$$H_{Low,Unity}(j\omega)=\frac{1}{1+C_1(R_1+R_2)j\omega+R_1R_2C_1C_2(j\omega)^2}$$

Pre-determining C1 = C2 and R1 = R2 in Figure 10 above, the equation became more simple. Evaluating it with resistor and capacitor values within the range recommended from the advisors yielded:

$$C_1 = C_2 = 1uF$$

$$R_1 = R_2 = 3.9k\Omega$$

5.2.3 Motor Board

The Motor control board is located inside the actuation platform, along with the Main board. This board includes direct connection to 7 motors, 5 clutches, 7 encoder-decoder couplings, 4 motor drivers, 1 PWM driver, 8 series elastic actuators, 2 ADCs, 3 voltage regulators, 1 inter-board serial communication connector, and a PIC32 chip.

The 7 motors are connected directly to quadrature encoders and each encoder has wires running from its two outputs to a 2-pin connector located on the board. The data obtained from the two encoders are then sent as input to the 2-channel decoder. The 8 decoders send the data to a PIC32 chip in parallel. Moreover, all 5 clutches, which are each attached to a transistor, obtain a PWM signal input (refer back to **Figure 4.2.7**) that is individually generated from the 16-channel PWM driver. The PWM driver is controlled through SPI from PIC32, with 5 of its outputs going to the clutches and the remaining 7 being sent to the motor drivers. The 4 motor drivers are used to drive the 7 motors since each driver is capable of driving two motors bidirectionally.

.

The PWM signals used to drive the motors are determined by a software control algorithm and the feedback obtained from the SEAs, the current limiting circuits and the encoders. The magnitude of the PWM signals is indicated by the desired current values sent through SPI from the PIC32 chip. The desired direction for motors is transmitted from the chip as well. The PIC32 also sends enable signals to the motor drivers to select which motor to

drive. The motors receive supply voltages from the voltage regulator and the 36V battery, coming from the Main board through power lines using a UART connector. A switching regulator is used to regulate the 20V needed for the clutches, the elbow and the wrist motors. Another switching regulator is used for regulating 36V down to 5V for sensors and the logic circuits. Two linear voltage regulators are used to obtain 3V for the PIC32 and 2.5V for ADCs (see **Figure 5.2.14** and **Figure 5.2.15**, respectively).

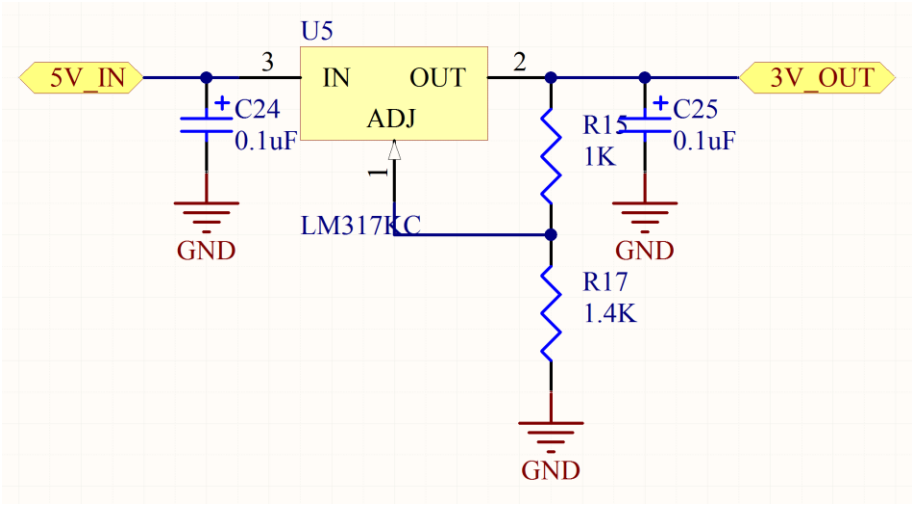


Figure 5.2.14: Linear Voltage Regulator Circuit (3V)

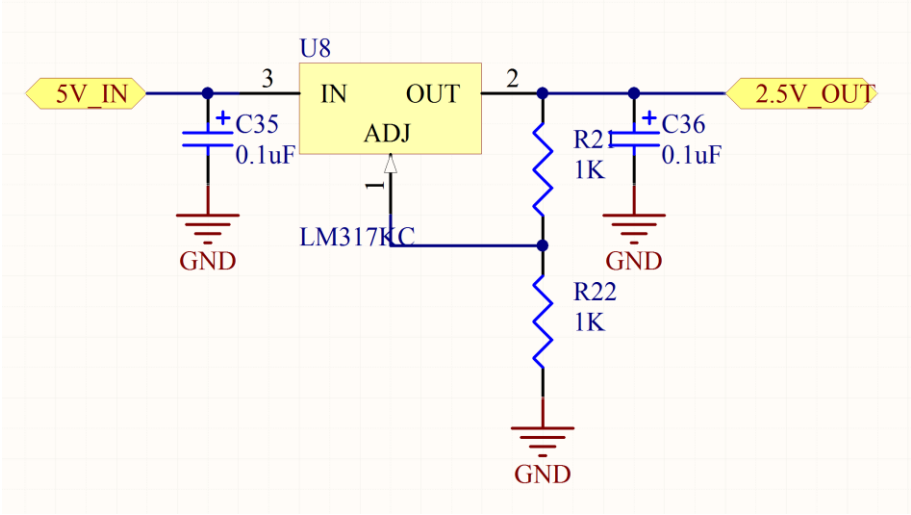


Figure 5.2.15: Linear Voltage Regulator Circuit (2.5V)

Each SEA circuit is made up of a soft-membrane potentiometer, a high-impedance voltage follower and a low pass filter. The analog output data from each sensor is sent to the ADC, which sends the data forward to the PIC32 through SPI. Another ADC is used to collect the current sense data from the motor drivers and send it to the PIC32. Each ADC is selected via its chip select, i.e. a digital input pin. The high-level functional diagram of the Motor board is shown below in **Figure 5.2.16**. The high-level Altium schematics and PCB of this board, along with enlarged pictures of the different components included in it can be seen in **Appendices E** and **F**.

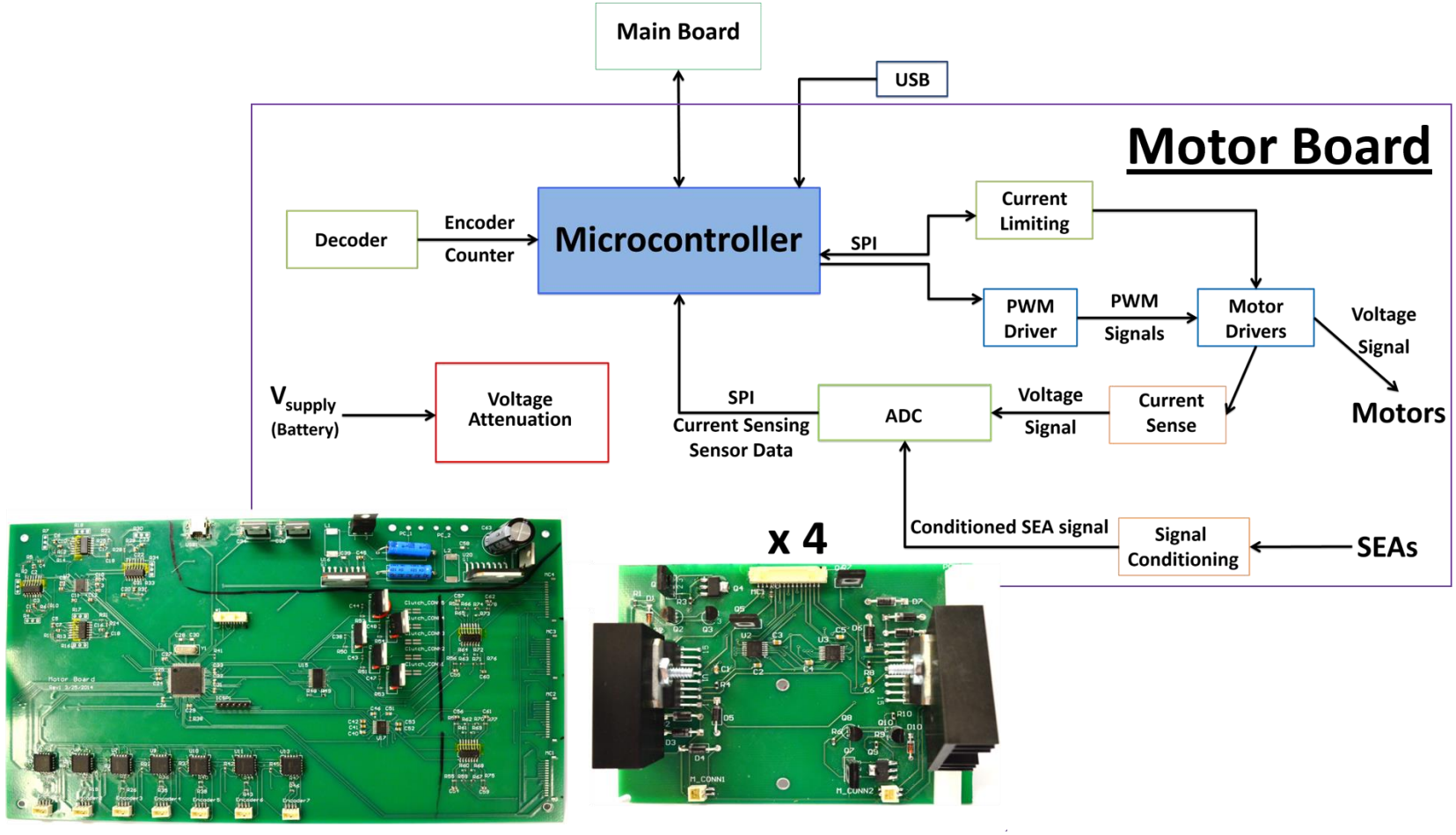


Figure 5.2.16: Motor Board High-Level Functional Diagram and Fully Assembled PCB.

5.2.4 Microcontroller

The system consists of three different microcontrollers. Each of the three different PCBs include a microcontroller on it, which performs various tasks and communicates with the Main board while the Main board communicates with each of the other boards and the computer. The few key specifications required for the microcontroller were to be able to communicate with two different interfaces (UART and SPI) and have enough digital I/O pins. The PIC32MX795F512L was selected for this project since it is a very powerful microcontroller, which includes both UART and SPI communication interfaces. Moreover, this specific microcontroller was highly suggested by the advisors of this project. Another advantage of using this particular microcontroller was that the AIM Lab, where this project took place, already had sufficient libraries to utilize the PIC32 as well as a few PIC32 development boards.

5.3 Software Architecture

There are 4 major components that needed software written for them (see **Figure 5.2**). The Main, Sensor, and Motor boards as well as a computer running a Java interface to interact with the Main board. The Main board and computer are connected together via USB while the Motor, and Sensor boards are all directly connected to the Main board but are not directly connected to each other. These boards act as coprocessors for the Main board.

5.3.1 Main Board Software Functionality

The Main board retrieves sensor information from all the other boards as well as receiving commands and sending data from and to the computer. The Main board sends the

desired control commands to the Motor board, but it does not directly control the motors. It interprets the data received by the flex and pressure sensors, and then sends position, velocity and/or current control commands to the Motor board. It also designates when to initiate or stop a control instance running on the Motor board.

5.3.2 Sensor Board Software Functionality

The Sensor board runs a very simple "sample and send" loop. The system monitors for the synchronous signal from the Main board that dictates when to sample data. When a synchronization signal is high, the controller retrieves a predefined amount of samples from all the pressure and flex sensors through the external ADCs. After the data have been gathered, it is filtered by averaging the samples, and then the most recent result is stored waiting to be transferred to the Main board. The Sensor board has an interrupt on the UART communication. When data are received, the interrupt fires, the packet is decoded to determine whether to light up an LED to a certain color or to start/stop publishing sensor information to the Main board.

5.3.3 Motor Board Software Functionality

The Motor board receives requests from the Main board such as desired motor position or velocity control set points and implements that control loop. It maintains an array of *ControlInstances,* which represent the control system. The program cycles through the *ControlInstances* to update and actuate each one according to settings defined in the *ControlInstance*. The Motor board continuously updates the Main board, over UART at 500Hz, with encoder position, encoder velocity, electrical current, and series elastic position so that the Main board can interpret the data along with sensor readings from other boards, thus

enabling the correct control instance to be commanded back to the Motor board. When the Motor board receives a command, it determines the proper action to take among the following: creating a new control instance, updating an existing control instance, or removing an existing control instance.

5.3.4 Inter-Board Software Communication Protocols

The Main controller supplies a synchronization signal over UART to the rest of the coprocessor boards. This signal tells the coprocessors the rate at which to sample data from the sensors. UART communication is used to send data between the boards. The Main board is directly connected to all the other coprocessor boards, which in turn are not directly connected to one another.

Asynchronous communication is used to communicate between the Main, Motor, and Sensor boards. Each communication packet and header between each board is customized to the data being sent between those boards. The packets and headers are not general, but instead specific to the data being sent.

Data are encapsulated into packets to be sent between the boards. Each packet contains a header as well as the payload to be sent. The header contains metadata about the packet, which is mainly information that doesn't need standard data types to be represented, such as in **Table 5.3.1** below (e.g. the _opt variable only needs 3 bits to be represented fully). A list of options for the _opt variable for each packet can be found as #define macros in commonStructs.h under **Appendix J**. Any data that need standard data types to be stored are listed in the packet, not the header.

Communication from the Main board to the Motor board is outlined in **Table 5.3.1** and **Table 5.3.2** below while the implementation code is shown in **Appendices G** and **I**, respectively. Each packet sent from the Main board to the Motor board is a set of size 10 bytes—a two byte header and an eight byte body. The design of this communication protocol was largely based on the control system implementation explained below in **Section 5.3.4.1**. Since there is a control loop nested inside another, the Motor board only needs to receive control commands based on the information it has: encoder position, encoder velocity, electrical current ADC counts, and series elastic ADC counts. In **Table 5.3.4** below, the *MotorTxMainRxPacket* shows that these sensor data are communicated to the Main board, where a decision is computed, and then the new desired control commands are sent to the Motor board. Sensor data are sent from the Motor board and the desired sensor readings for control are sent by the Main board.

Table 5.3.1: *MainTxMotorRxHeader*

Bit	Field	Description
0-3	_id	Motor to control for this control instance
4-6	_opt	Specifies whether to start/update, stop, or remove control instance, subscribe/unsubscribe to motor data
7-10	_controlType	Type of control (ex: position, velocity, etc.)
11-15	_aux	Auxiliary unused bits reserved for future use

Table 5.3.2: *MainTxMotorRxPacket*

Bit	Field	Type	Description
0-15	header	MainTxMotorRxHeader	Header for this packet
16-31	encoderPos	int16_t	Desired encoder position

32-47	encoderVel	int16_t	Desired encoder velocity
48-63	currentAdc	uint16_t	Desired current limit
64-79	seriesElasticAdc	uint16_t	Desired series elastic

Communication from the Motor board to the Main board is outlined in **Table 5.3.3** and **Table 5.3.4** below.

Table 5.3.3: *MotorTxMainRxHeader*

Bit	Field	Description
0-3	_id	Specifies which motor the data are associated with
4-6	_opt	Specifies whether an error occurred or it reached steady state
7	_aux	Auxiliary unused bits reserved for future use

Table 5.3.4: *MotorTxMainRxPacket*

Bit	Field	Type	Description
0-7	header	MotorTxMainRxHeader	Header for this packet
8-23	encoderPos	int16_t	Measured encoder position
24-39	encoderVel	int16_t	Measured encoder velocity
40-55	currentAdc	uint16_t	Measured current in ADC counts
56-71	seriesElasticAdc	uint16_t	Measured series elastic in ADC counts

Communication between the Main board and Sensor board is outlined in **Table 5.3.5** through ***Table 5.3.7***. Since the Sensor board only samples data and sends it to the Main board, the communication between the two is fairly simple. **Table 5.3.5** describes the

MainTxSensorRxPacket data which are sent to the Sensor board. The Main board either commands the Sensor board to light the LEDs or to publish/unpublish data for a sensor set (e.g. flex and pressure sensor data for a specified finger).

Table 5.3.5: *MainTxSensorRxPacket*

Bit	Field	Description
0-3	_id	Identifier of motor-sensor set to either subscribe or command
4	_s	Set whether to subscribe/command or unsubscribe 0b = unsubscribe 1b = subscribe or command LED (For command LED, _id must be 1111b)
5-7	_opt	Set LED color

Table 5.3.6: *SensorTxMainRxHeader*

Bit	Field	Description
0-3	_id	Identifier of motor-sensor set
5-7	_aux	Auxiliary unused bits reserved for future use

Table 5.3.7: *SensorTxMainRxPacket*

Bit	Field	Type	Description
0-7	header	SensorTxMainRxHeader	Header for this packet
8-23	flexAdc	uint16_t	Measured flex sensor value in ADC counts
24-39	presAdc	uint16_t	Measured pressure sensor value in ADC counts

5.3.4.1 Main-Motor Board Control Loop

The main idea is having simple peripheral boards, such as the Sensor board, running real-time control loops, while having the main logic and control processing taking place on the Motor and Main boards. The Main board sends out a synchronization signal to the peripheral boards and these boards respond with the measured values requested by the Main board. There are two control loops running, one on the Motor board and another one on the Main board, which encompasses the control loop of the Motor board.

5.3.5 PC/Controller Communication Software

A java framework specificially designed for the PIC32 was used to communicate between the PIC32 and a desktop computer. This works by defining the communication protocol as a set of remote procedure call (rpc) functions on the Main board, which dictates how many bytes, words, and ints the Main board will receive and send for each rpc function. Through the java framework on the desktop computer, and of these rpc functions can be called in which data will be sent from the desktop computer to the Main board, the function will be called to process the data and run other commands, and then the result will be set to the packet and sent back to the desktop computer. The communication protocol definition can be seen in **Appendix G**.

6. Results and Discussion

The final prototype of the system (see **Figure 6.1**) allows for motorized flexion and extension of the fingers and the elbow, as well as the pronation and supination of the wrist. This third iteration of the system improved from the previous two by adding a modular motor units that can be easily removed and modified, a wrist support with trapezoid-shaped hard stops, and a detachable mechanism that allows the extension cables to be disengaged while dinning the device. Additional improvements were made in the design of more compact and comfortable elbow braces, more robust design of finger cable guides to prevent losses from glove elasticity, as well as more compact and workable design for cable tensioning.

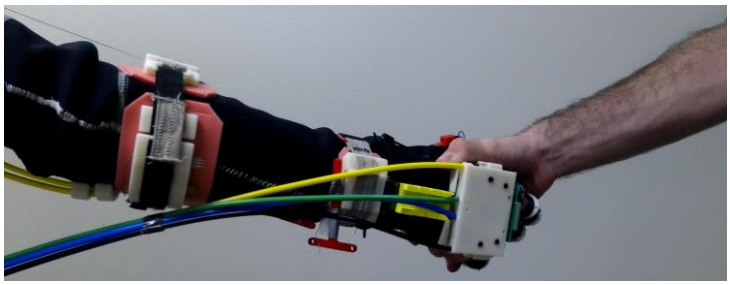


Figure 6.1: Exomuscular Sleeve Final Prototype

Improvements were also made with the development of several PCBs for dedicated tasks, such as reading and interpreting sensor data, actuating and controlling the system, and communicating with the master computer. The system is capable of accurately collecting and interpreting sensor data. However, the control of the finger and elbow spools through the Motor board was not accomplished. One of the major issues the team encountered was the integration of the system, which took longer than expected, specially due to the many custom circuit boards and mechanical components that were developed with several iterations for each

(details offered in the following sections). A demonstration video of the system can be found at the following link:

https://www.youtube.com/watch?v=Xs_7N57YGdU&feature=youtu.be

6.1 Mechanical Subsystem

6.1.1 Finger Actuation

Reflecting on the mechanical oriented goals of this project, the final device achieved finger actuation, wrist pronation/supination and actuation of the elbow joint through an angle of 150 degrees. After successfully actuating the fingers on the glove in conjunction with the detachable cable connector on a team member's hand, the glove was fitted onto a mannequin's hand to demonstrate unaided actuation. The fingertip was then hooked up to a fish scale in order to measure the end effector force generated when the Bowden cable is pulled by the motor module spool. The setup can be seen below in **Figure 6.1.1**.

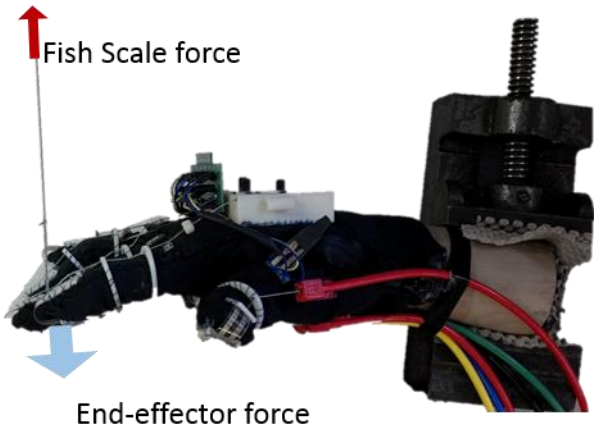


Figure 6.1.1: Force Setup

The force generated at the fingertip is function of the angle at which the finger is bent due to the geometry of the finger guides. To analyze the force generation, two angles of the pointer finger were selected to measure the end effector force.

Calculating the torque generated from the finger motors, the force on the fingertip were calculated based on the tension in the Bowden cables. Inspecting the torque curve from the maxon motors specified in the final design section, the output force was calculated to be 0.8. Running the motor and Gear box combination at .4 amps and 32 volts, the pointer finger generated a force of 0.5 lbs. The kinematics are visualized below in **Figure 6.1.2.**

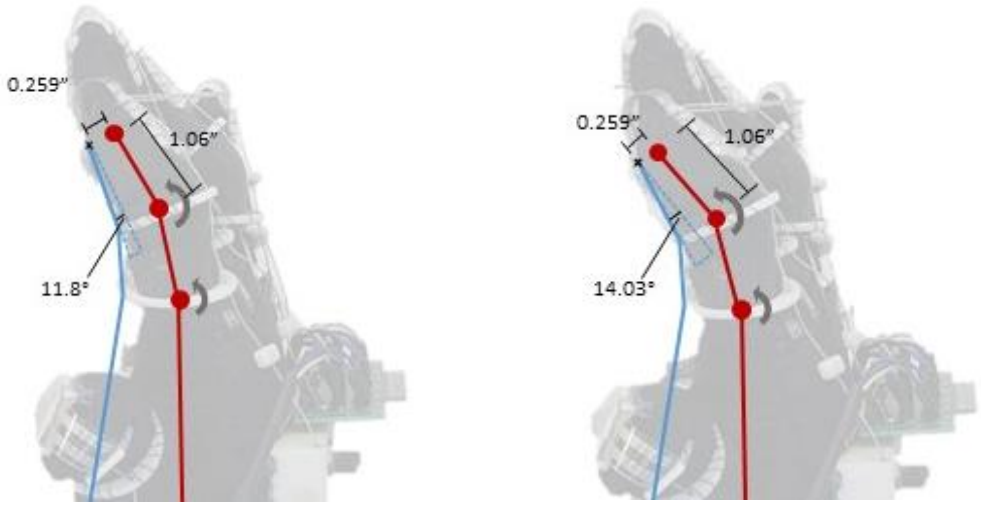


Figure 6.1.2: Finger Kinematics

An unexpected outcome of the finger motor concerned the exact nature of the clutches in their position in the finger motor unit. As seen in **Figure 5.1.23**, one end of the clutch is a tooghed radial gear. In the fingwer motor units, the spools were designed with a mirrored recess that these teeth slotted into with a desired press fit. A press fit would allow the spools to be removed from the clutches without damaging them, but still prevent them from simply falling off. An unintended side effect of the press fit was that due to the sensitivity of the clutches, this spool tended to compress the plastic gear on the spool, tightening its grip on the

central shaft which was coupled to the other side. The effect of this was that whenever the spools were attached to the clutches, they were at least partially engaged from the squeezing effect of the press fit. The team therefore recommends that any future projects involving the clutches use a method other that a press fitting recess for coupling the spools to the clutches.

6.1.2 Elbow Actuation

Elbow actuation was somewhat successful. Early tests of the elbow motor determined that at no load, the motor required approximately 4 amps at 20 volts to turn over. Once the motor started moving, the current draw dropped to about 2 amps at 20 volts. This is likely a result from disassembling the motor, removing the lubrication, and replacing the lubrication with Royal Purple 013012 NLGI No. 2 High Performance Multi-Purpose Synthetic Ultra Performance Grease, which seemed more viscous than the original lubrication. Because the motor modules were not functional during testing, the motors were instead connected back to the cordless drill speed controllers and batteries for testing elbow actuation.

During testing, the team determined that the motor with the original speed controller and battery was capable of actuation the elbow; however, during one of the tests, the Dyneema snapped. Even though the edges of the Bowden retainer on the motor unit were chamfered, the edges still caused the Dyneema to wear and eventually break after repeated actuation. When one of the flexion Dyneema cables snapped, the other cable was loaded excessively, and the PTFE tubing used for the Bowden on the intact cable buckled, rendering the motor unit ineffective for flexion.

Because of the high forces required to actuate the elbow, the Bowden system should have instead used Jagwire instead of PTFE tubing to prevent the possibility of buckling. Jagwire housing and steel cable would be met the high-force requirements for actuating the elbow.

6.1.3 Wrist Actuation

Full range of motion for wrist pronation and supination was achieved in the final design. However, torque outputs of the system were not measured. It is unlikely that the torque output of the system would have matched normal torque output of a human. The brace part just proximal of the wrist would begin sliding in the proximal direction of the forearm if attempts were made to match the output torque of a healthy individual.

6.2 Electrical Subsystem

6.2.1 Sensors

Flex sensor circuits and pressure sensor circuits were bread-boarded and tested in order to verify their functionality (see **Figure 6.2.1**). The circuits used were presented in the previous chapter in **Figure 5.2.4** and **Figure 5.2.5**. For flex sensor a module was created by the team to test the angle (see **Figure 6.2.1**). Two different sets of AD8426, resistors and flex sensors were tested separately. The data from these tests are available in Appendix L. The graph shown below summarizes the results of the tests.

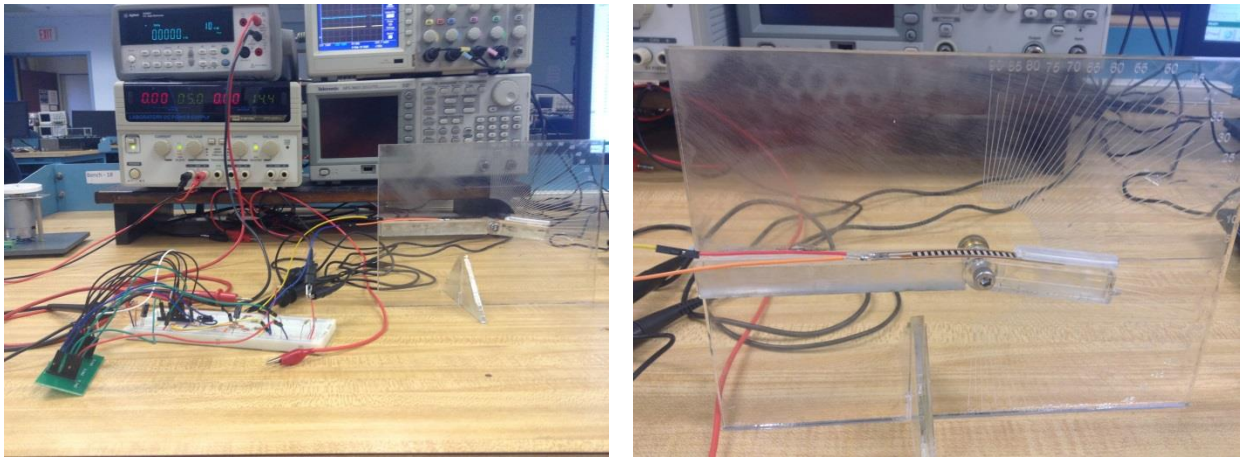


Figure 6.2.1: Bread-Boarded Circuit for Calibration of Flex Sensors

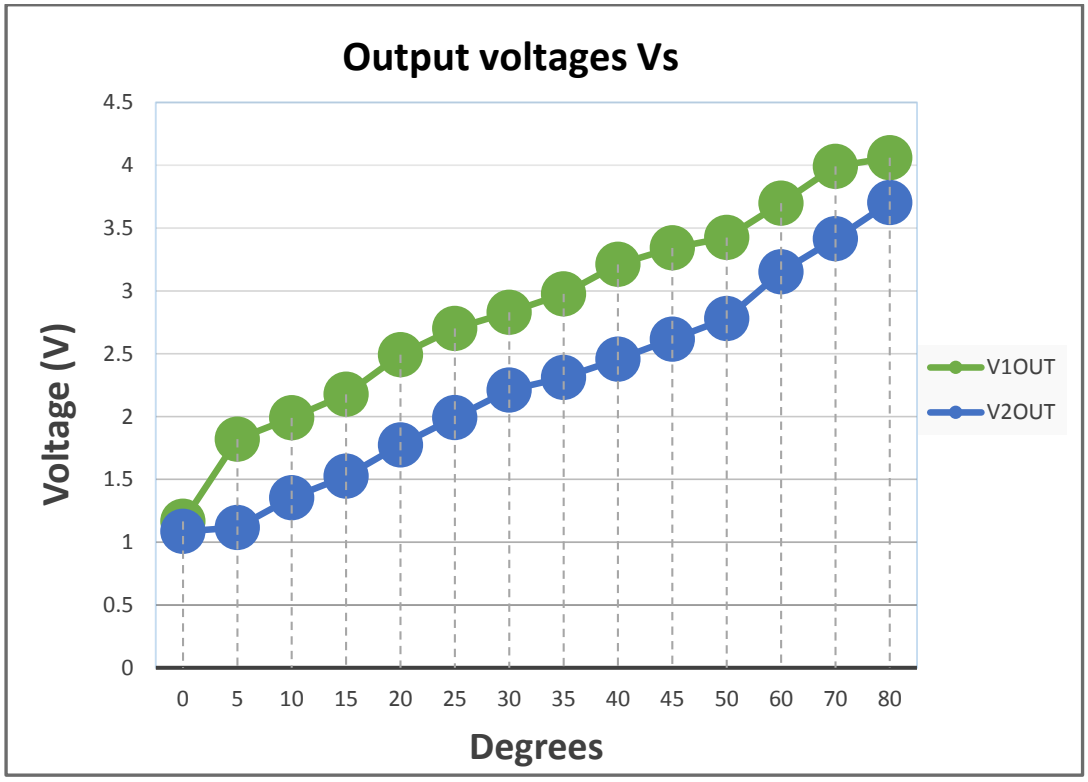


Figure 6.2.2: Flex Sensor Calibration (Output Voltage vs. Degree)

In the graph above, "V1OUT" is the output result of tests done with the 1st set of components (shown in Appendix L), which had difference of 0.767V when bent from 0° to 80°. "V2OUT" is

the output result of tests done with the 2nd set of components (shown in Appendix L) which had difference of 0.852V when bent from 0° to 80°.

6.2.2 Tests with ADCs and Digital Potentiometers

ADCs and Digital-pots needed Serial Peripheral Interface (SPI) to communicate with the PIC32 micro-controller. These peripherals were tested using the PIC32 USB Starter Kit Development Board as seen in **Figure 6.2.3**. The ADC has several built in modes of operation which can be set from SPI. For the initial testing purpose, the mode chosen was a simple counter mode which keeps count of the previous channel and keeps looping around all channels. After successful implementation this mode, it was further modified to loop from zero to specific channel to loop around which was used in custom PCBs. The following bytes were sent to program the ADC for this specific mode:

```
BYTE sendSetupADCByte1 = 0b01011111;
sendSetupADCByte1 |= (channelEnd << 3);
BYTE sendSetupADCByte2 = 0b10010000;
```

After programming the ADC, a potentiometer was used to determine if the ADC was able to read the correct voltage. The wiper of the potentiometer was connected to one ADC pin and it was observed that data sent out of the ADC was changing from 0 to 4095 as the potentiometer was turned. This result was expected because the wiper was changing values from ground to power rail without any other circuitry.

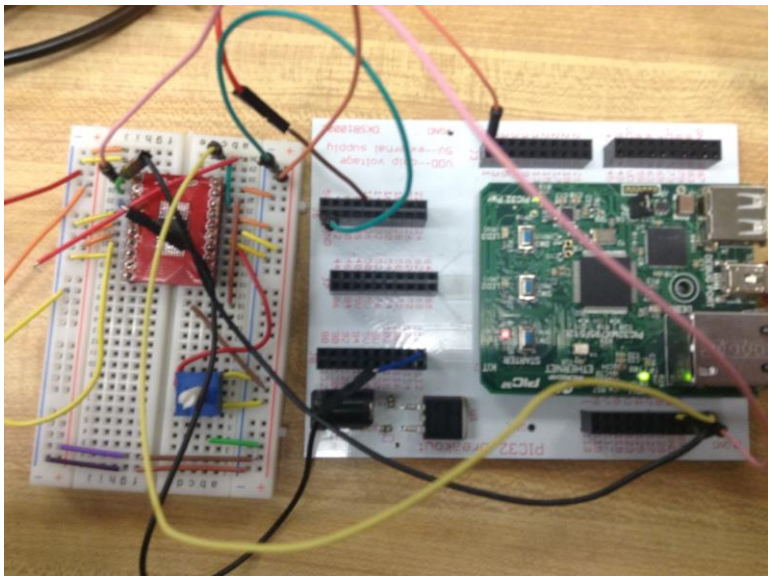


Figure 6.2.3: Bread-Boarded Circuit for testing the ADC using the PIC USB Starter Kit Development Board

Similarly, the digital potentiometer was also tested using the PIC32 USB Starter Kit Development Board as seen in **Figure 6.2.4**. The digital potentiometer has 255 levels where byte value 0 sets it to 0Ω and byte value 255 sets it to 50 kΩ. To test the functionality of the Digi-Pot and a 51kΩ resistor was connected in series acros a 10V power supply. PIC32 was programmed to loop and increase the byte value by 1 at the end of every loop unless it was maximum value which was then reset to 0. The result was successful implementation of the code and can be observed in **Figure 6.2.5**.

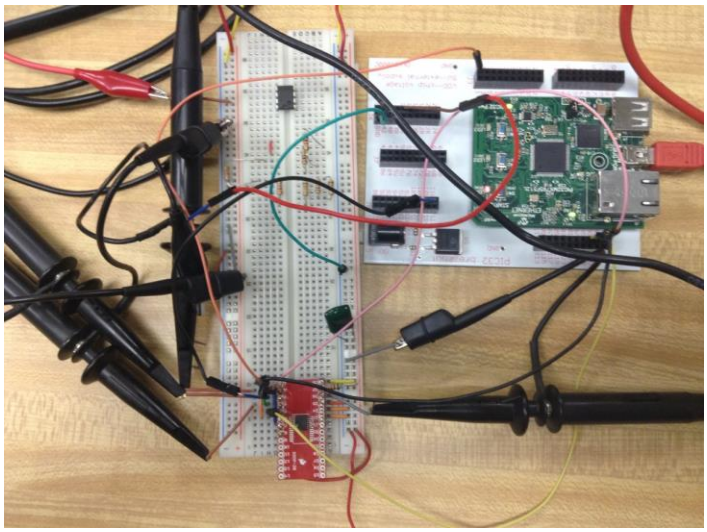


Figure 6.2.4: Bread-Boarded Circuit for testing the Digi-Pot using the PIC USB Starter Kit Development Board

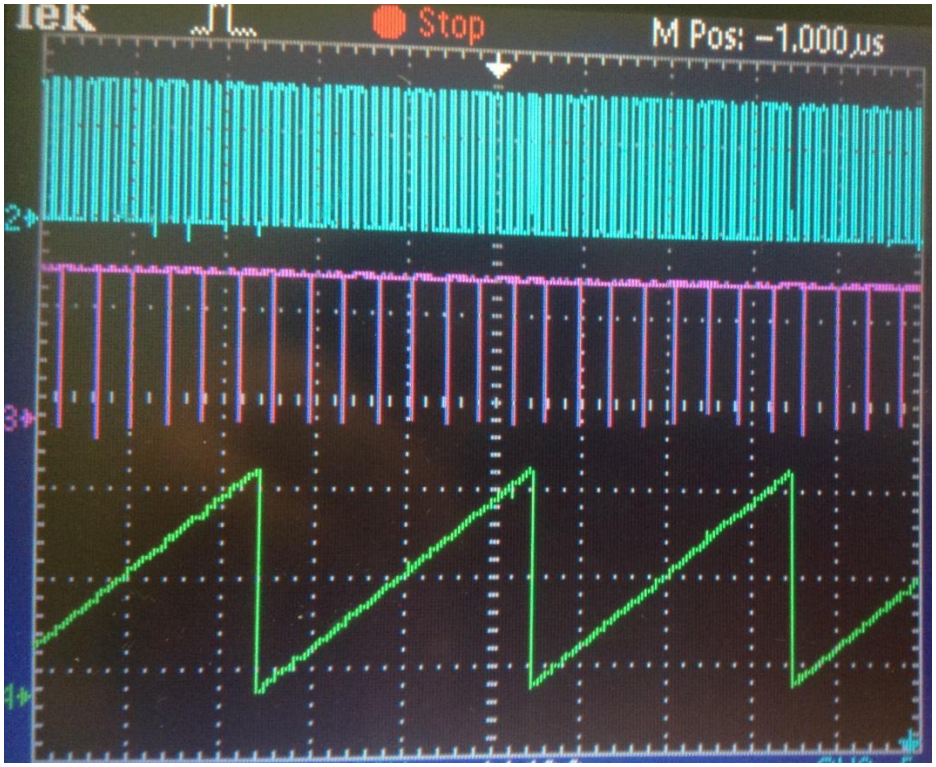


Figure 6.2.5: Digi-Pot Output Response to Loop-Counting Functionality Test

6.2.3 LEDs

WS2812 from Worldsemi were the chosen LEDs for visual feedback. Research confirmed that SPI could control these LEDs, but it was not simple SPI data that lights them up. These LEDs require PMW signals with specific high and low timings. About 0.35 µs high and 0.8 µs low PMW signal gets translated as code 0. About 0.7 µs high and 0.6 µs low PMW signal gets translated into code 1 (see **Figure 6.2.6**). Twenty-four of these codes are required to set a specific color for 1 LED. Having SPI clock period of 1.25 µs and sending 24 bit data will not work on this. For this LED to work with SPI, the clock speed has to be exactly 4 Mhz which gives clock period of 0.25µs. With this clock speed 5 bit code (10000) becomes logic 1 for LED and another 5 bit code (11110) becomes logic 0 for LED. Therefore it needs 120 bits (5*24) of data at 4 Mhz to light up one LED.

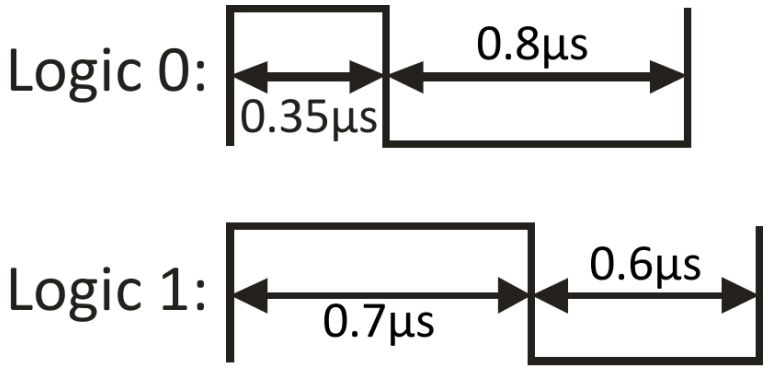


Figure 6.2.6: Required Duty Cycles for Logic 0 and 1 for the selected LEDs

Code was successfully written for the PIC32 to operate these LEDs with their corresponding colors for feedback using SPI. A video demonstrating the LEDs changing color as pressure was exerted at the fingertips can be seen at the following link:

https://www.youtube.com/watch?v=k4Ubl5CXWMM&feature=youtu.be

6.2.4 External 16 Channel PWM Generator

The selected TLC5940 16-channel external PWM generator by Texas Instruments (TI) was not the correct peripheral to interface with the clutches and motor drivers. This chip works by switching the ground on and off instead of switching the power on and off, which does not make it suitable for this application. The PWM outputs could not be directly connected to the motor driver and clutch, so custom circuitry was created to integrate the PWM generator to the motor drivers and clutches. There was also difficulty programing the device. TI created a programming flow chart to aid in writing a program to interact with the chip, which the team followed exactly and yet the chip could not be configured correctly. Research on this matter revealed that there is a library for Arduino to communicate with the TLC5940 which works, yet it contradicts programming guidelines stated in the TI flowchart.

A solution to this problem is to use PWM generator Integrated Circuits (ICs) that source current. These ICs don't have nearly as many pins as the ground switching PWM generators, but this is not necessarily a problem. Since the Motor board concept consisted of a motor board and motor modules, the new PWM chips could be placed on the modules themselves. This new PWM generator also removes the need to have interfacing circuitry, which reduces the number of components. A suitable PWM generator would be the Toshiba TB62779FNG.

6.2.5 Graphical User Interface

A graphical user interface was created to display information from the Sensor board (see **Figure 6.2.7**). When one of the checkboxes are selected, a request is sent to the Main board, which sends a packet to the Sensor to publish the selected finger's information. When "Begin Sampling" is selected, the computer polls the Main board which in turns command the

Sensor board to sample and send sensor data. This action runs in a new thread, which allows any finger to be selected or unselected to have the sensor board send data.

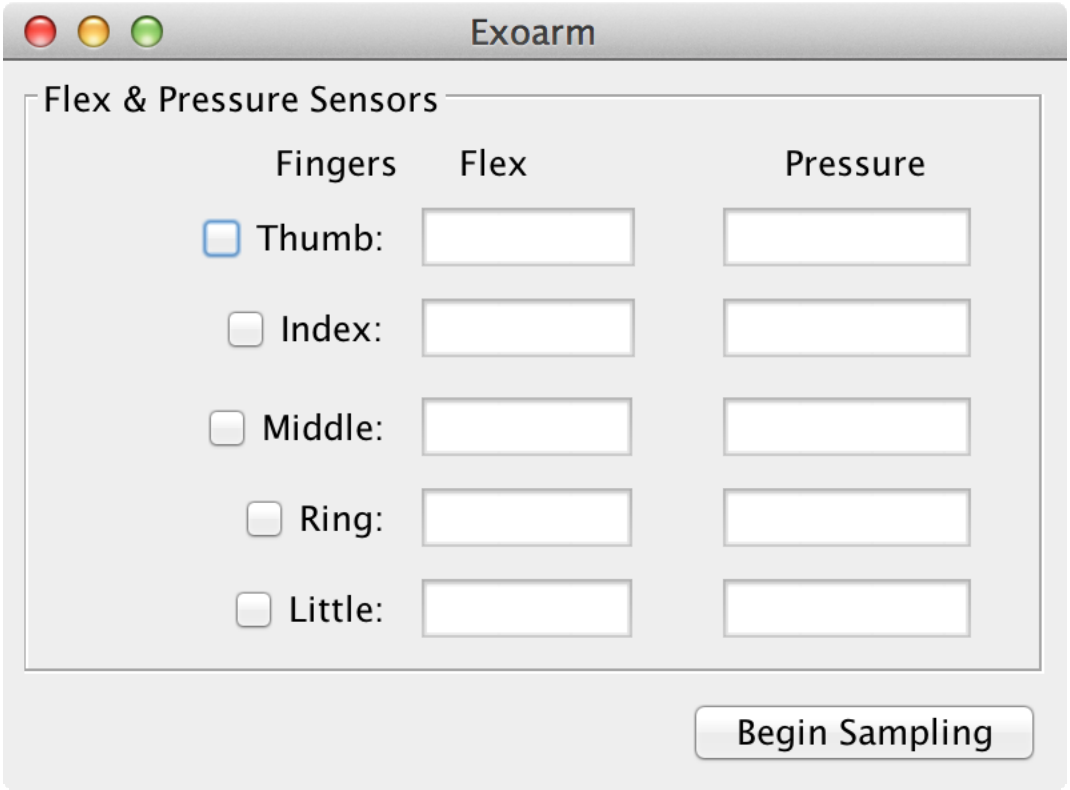


Figure 6.2.7: Graphical User Interface

7. Future Work and Recommendations

The exomuscular device designed for stroke rehabilitation was designed as a research platform to explore upper limb rehabilitation. Incorporating previous designs for finger and elbow actuation, seven degrees of freedom are controlled by a modular motor system with the ability for position or torque control of the patient's extremities.

Significant design work was devoted to developing a system to detach the Bowden cables at the wrist. The detachable concept certainly aided in donning the device. If the device was developed for a clinical system, connecting the detachable mechanism would have to be significantly easier than the current design. The current iteration of the detachable mechanism requires the finger extension cables to be drawn out when the upper part of the mechanism is attached. In **Figure 7.1** below, the steps are listed during the attaching procedure.

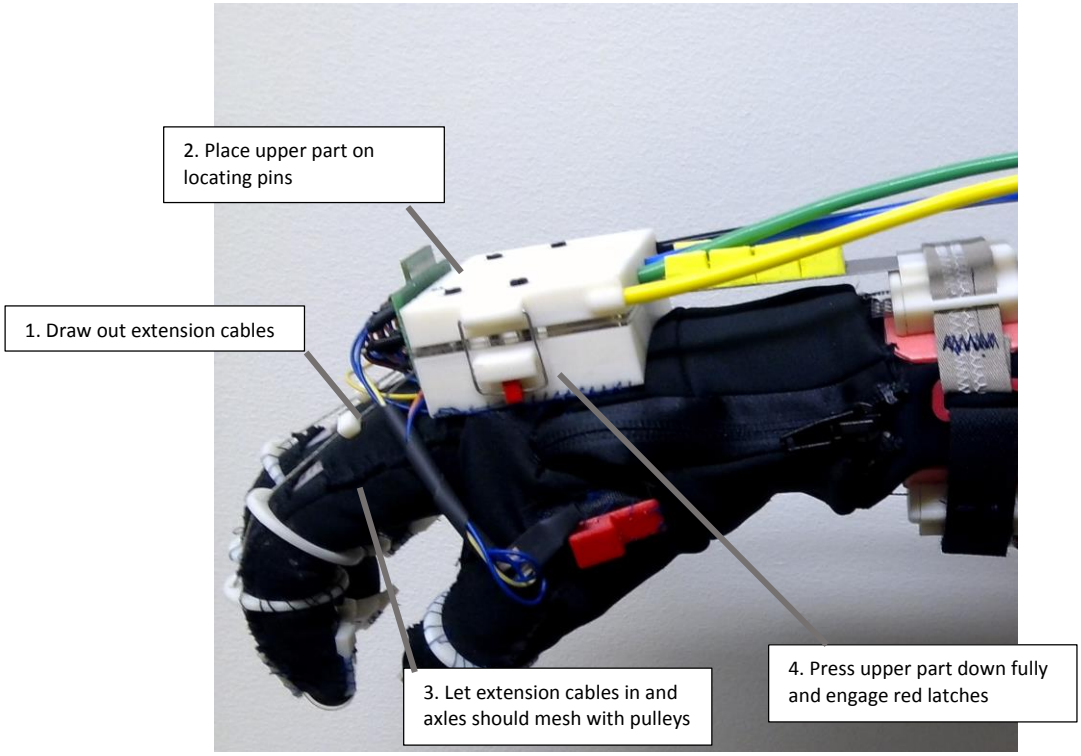


Figure 7.1: Steps to Operate Detachable Mechanism

Ideally the upper half of the mechanism should snap on in one simple motion instead of placing it part way on and rotating the extension pulleys until the two halves line up. The constant force springs keep the cables wound around the pulleys in both halves of the mechanism. Both halves require the constant tension to keep the cables lined up while the two halves are disconnected, but their orientation presents an interesting problem. As shown in **Figure 7.2** below if both the top and bottom pulleys are wound in to keep the constant tension they cannot be connected axially because there would not be any play for the cable to move in either direction.

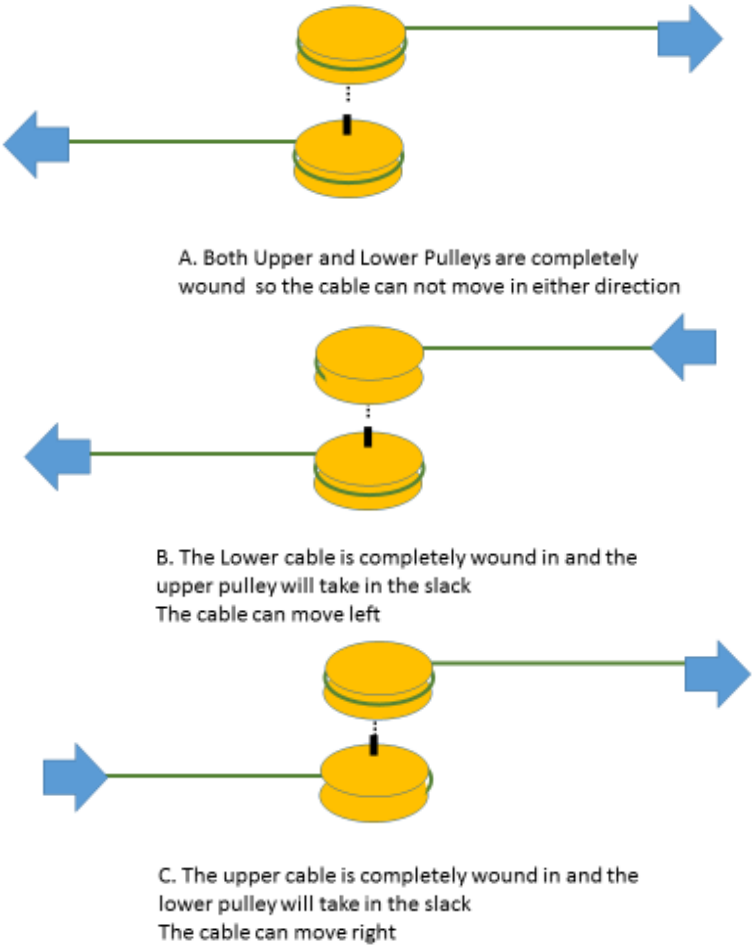


Figure 7.2: Pulley Mechanism

Our recommendation is to still include constant force springs to keep the extension cables wound on the bottom pulleys. On the top set of pulleys, the cables should be completely unwound and tensioned at the motor module end of the Bowden cable as in case B above. This way the axles can be lined up and the detachable mechanism can be attached in one motion.

Although the spools in the actuation platform are more accessible than previous designs, the spools still proved a little difficult to tension. Much more prototyping and researching is required for figuring out methods for cable actuation and cable management. The decreased sizes of the spools also meant much narrower acceptable tolerances. The narrower tolerances were especially noticeable on the elbow spool. When actuating quickly, the Dyneema would sometime pop off of the spools. This problem could be solved be increasing the height of the walls of the spool or by limiting the rate of actuation.

An additional problem with using aluminum for the elbow motor housing was the edges of the aluminum in contact with the Dyneema. Although the edges were chamfered, they still caused unnecessary wear on the Dyneema, causing the Dyneema to snap in one instance of testing. This Dyneema to aluminum contact problem can be resolved by including plastic fittings in all the locations that the Dynnema might come into direct contact with an aluminum edge. An alternative solution would be to use steel cable and Jagwire for the Bowden part of the system. The steel cable could be coupled to Dyneema just before the cable would exit the Bowden and become exposed.

Reducing the size of the actuation platform would also be ideal. This, however, will require more prototyping and testing of different ways of actuating the device through the use

of cables. For example, instead of using spools driven directly by motors, the size of the actuator might be reduced by using smaller motor with a worm gear and cam design.

The Motor board can be modified to remove design weaknesses such as trace widths and electrical connectors, which are too small for the required amount of current flowing to operate the entire board. It is also recommended to pay close attention to ordering components with packages that exactly correspond to the footprints on the PCBs to avoid major on-board fixes. For connections, it came to our attention to consider appropriate wires for handling associated current. Having a correct and exact crimping tool is recommended in case of designing custom connector parts. Additional work is suggested in the software architecture to ensure it is real-time. It may be beneficial to move onto a device that is running a real-time operating system. This is important to ensure the tasks are scheduled properly especially when working with multiple devices.

References

[1] M. A. Delph, S. A. Fischer, P. W. Gauthier, C. H. M. Luna, E. A. Clancy and G. S. Fischer, "A Soft Robotic Exomusculature Glove with Integrated sEMG Sensing for Hand Rehabilitation," in *International Conference on Rehabilitation Robotics (ICORR)*, 2013.

[2] M. J. Brauckmann, E. S. Calamari, B. J. Leone, S. M. Lipkind, C. J. Molica, A. S. Piscopiello and W. F. Terry, *A Soft Robotic Exo-muscular Arm Brace (Major Qualifying Project),* Worcester Polytechnic Institute, 2013, Available online at: http://www.wpi.edu/Pubs/E-project/Available/E-project-042513-105735/unrestricted/GSF_M131_-_Soft_Robotic_Exo-Muscular_Arm_Brace.pdf.

[3] V. L. Roger, A. S. Go, D. M. Lloyd-Jones, E. J. Benjamin, J. D. Berry, W. B. Borden, D. M. Bravata, S. Dai, E. S. Ford, C. S. Fox, H. J. Fullerton, C. Gillespie, S. M. Hailpern, J. A. Heit, V. J. Howard, B. M. Kissela, S. J. Kittner and D. T. Lackland, "Heart Disease and Stroke Statistics—2012 Update A Report From the American Heart Association," *Circulation,* vol. 125, no. 1, p. e2–e220, 2012.

[4] G. Kwakkel, B. J. Kollen and H. I. Krebs, "Effects of robot-assisted therapy on upper limb recovery after stroke: a systematic review," *Neurorehabilitation and Neural Repair,* vol. 22, no. 2, p. 111–121, 2008.

[5] A. Sunderland, D. J. Tinson, E. L. Bradley, D. Fletcher, R. L. Hewer and D. T. Wade, "Enhanced physical therapy improves recovery of arm function after stroke. A randomised

controlled trial," *Journal of Neurology, Neurosurgery & Psychiatry,* vol. 55, no. 7, p. 530–535, 1992.

[6] C. Bütefisch, H. Hummelsheim, P. Denzler and K. H. Mauritz, "Repetitive training of isolated movements improves the outcome of motor rehabilitation of the centrally paretic hand," *Journal of the Neurological Sciences,* vol. 130, no. 1, p. 59–68, 1995.

[7] B. L. Kincaid and K. N. An, "Elbow Joint Biomechanics For Preclinical Evaluation Of Total Elbow Prostheses," *Journal of Biomechanics,* 2013.

[8] M. Foglia and M. Valori, "A high performance wire device for an elbow prosthesis," in *4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, 2012.

[9] C. L. Schwarz and R. J. Taylor, "The Anatomy and Mechanics of the Human Hand," *Atifficial Limbs,* p. 22, 1955.

[10] A. Schweizer, O. Frank, P. E. Ochsner and H. A. C. Jacob, "Friction between human finger flexor tendons and pulleys at high loads," *Journal of Biomechanics,* vol. 36, no. 1, p. 63–71, 2003.

[11] M. Malvezzi, G. Gioioso, G. Salvietti, D. Prattichizzo and A. Bicchi, "SynGrasp: a MATLAB Toolbox for Grasp Analysis of Human and Robotic Hands," in *IEEE International Conference on Robotics and Automation*, 2013.

[12] B. H. Dobkin, "Rehabilitation after stroke," *New England Journal of Medicine,* vol. 352, no.

16, p. 1677–1684, 2005.

[13] "Muscle Weakness After Stroke: Hemiparesis," *National Stroke Association,* 2008.

[14] P. H. McCrea, J. J. Eng and A. J. Hodgson, "Time and magnitude of torque generation is impaired in both arms following stroke," *Muscle & Nerve,* vol. 28, no. 1, p. 46–53, 2003.

[15] J. Cauraugh, K. Light, S. Kim, M. Thigpen and A. Behrman, "Chronic Motor Dysfunction After Stroke Recovering Wrist and Finger Extension by Electromyography-Triggered Neuromuscular Stimulation," *Stroke,* vol. 6, no. 1360–1364, p. 31, 2000.

[16] J. W. Krakauer, "Arm function after stroke: from physiology to recovery," *Seminars in Neurology,* vol. 25, 2005.

[17] L. Ada, C. G. Canning and S. L. Low, "Stroke patients have selective muscle weakness in shortened range," *Brain,* vol. 126, no. 3, p. 724–731, 2003.

[18] T. H. Wagner, A. C. Lo, P. Peduzzi, D. M. Bravata, G. D. Huang, H. I. Krebs, R. J. Ringer, D. G. Federman, L. G. Richards, J. K. Haselkorn, G. F. Wittenberg, B. T. Bever, P. W. Duncan, A. Siroka and P. D. Guarino, "An economic analysis of robot-assisted therapy for long-term upper-limb impairment after stroke," *Stroke,* vol. 42, no. 9, p. 2630–2632, 2011.

[19] B. R. Brewer, S. K. McDowell and L. C. Worthen-Chaudhari, "Poststroke upper extremity rehabilitation: a review of robotic systems and clinical results," *Topics in Stroke Rehabilitation,* vol. 14, no. 6, p. 22–44, 2007.

[20] D. J. Reinkensmeyer, B. D. Schmit and W. Z. Rymer, "Assessment of active and passive restraint during guided reaching after chronic brain injury," *Annals of Biomedical Engineering,* vol. 27, no. 6, p. 805–814, 1999.

[21] J. Dewald and R. F. Beer, "Abnormal joint torque patterns in the paretic upper limb of subjects with hemiparesis," *Muscle & Nerve,* vol. 24, no. 2, p. 273–283, 2001.

[22] C. G. Burgard, P. S. Lum, P. C. Shor and H. F. M. V. d. Loos, "Robot-assisted movement training compared with conventional therapy techniques for the rehabilitation of upper-limb motor function after stroke," *Archives of Physical Medicine and Rehabilitation,* vol. 83, no. 7, p. 952–959, 2002.

[23] H. I. Krebs, N. Hogan, M. L. Aisen and B. T. Volpe, "Robot-aided neurorehabilitation," *IEEE Transactions on Rehabilitation Engineering,* vol. 6, no. 1, p. 75–87, 1998.

[24] D. J. Reinkensmeyer, L. E. Kahn, M. Averbuch, A. McKenna-Cole, B. D. Schmit and W. Z. Rymer, "Understanding and treating arm movement impairment after chronic brain injury: progress with the ARM guide," *Journal of Rehabilitation rResearch and Development,* vol. 37, no. 6, p. 653–662, 2000.

[25] J. J. Summers, F. A. Kagerer, M. I. Garry, C. Y. Hiraga, A. Loftus and J. H. Cauraugh, "Bilateral and unilateral movement training on upper limb function in chronic stroke patients: a {TMS} study," *Journal of the Neurological Sciences,* vol. 252, no. 1, p. 76–82, 2007.

[26] J. V. Basmajian, C. A. Gowland, M. A. Finlayson, A. L. Hall, L. R. Swanson, P. W. Stratford, J.

E. Trotter and M. E. Brandstater, "Stroke treatment: comparison of integrated behavioral-physical therapy vs traditional physical therapy programs," *Archives of Physical Medicine and Rehabilitation,* vol. 68, no. 5, p. 267, 1987.

[27] R. Boian, A. Sharma, C. Han, A. Merians, G. Burdea, S. Adamovich, M. Recce, M. Tremaine and H. Poizner, "Virtual reality-based post-stroke hand rehabilitation," *Studies in Health Technology and Informatics,* pp. 64-70, 2002.

[28] J. L. Patton, M. Kovic and F. A. Mussa-Ivaldi, "Custom-designed haptic training for restoring reaching ability to individuals with poststroke hemiparesis," *Journal of Rehabilitation Research and Development,* vol. 43, no. 5, p. 643, 2006.

[29] J. L. Patton, M. E. Stoykov, M. Kovic and F. A. Mussa-Ivaldi, "Evaluation of robotic training forces that either enhance or reduce error in chronic hemiparetic stroke survivors," *Experimental Brain Research,* vol. 168, no. 3, p. 368–383, 2006.

[30] B. R. Brewer, R. Klatzky and Y. Matsuoka, "Initial therapeutic results of visual feedback manipulation in robotic rehabilitation," *2006 International Workshop on Virtual Rehabilitation,* p. 160–166, 2006.

[31] Y. Wei, J. Patton, P. Bajaj and R. Scheidt, "A real-time haptic/graphic demonstration of how error augmentation can enhance learning," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2005.

[32] M. L. Aisen, H. I. Krebs, N. Hogan, F. McDowell and B. T. Volpe, "The effect of robot-

assisted therapy and rehabilitative training on motor recovery following stroke," *Archives of Neurology,* vol. 54, no. 4, p. 443, 1997.

[33] B. T. Volpe, H. I. Krebs, N. Hogan, L. Edelsteinn, C. M. Diels and M. L. Aisen, "Robot training enhanced motor outcome in patients with stroke maintained over 3 years," *Neurology,* vol. 53, no. 8, p. 1874–1874, 1999.

[34] B. T. Volpe, H. I. Krebs, N. Hogan, L. Edelstein, C. Diels and M. Aisen, "A novel approach to stroke rehabilitation Robot-aided sensorimotor stimulation," *Neurology,* vol. 54, no. 10, p. 1938–1944, 2000.

[35] H. I. Krebs, B. T. Volpe, M. L. Aisen and N. Hogan, "Increasing productivity and quality of care: robot-aided neuro-rehabilitation," *Journal of Rehabilitation Research and Development,* vol. 37, no. 6, p. 639–652, 2000.

[36] H. I. Krebs, M. Ferraro, S. P. Buerger, M. J. Newbery, A. Makiyama, M. Sandmann, D. Lynch, B. T. Volpe and N. Hogan, "Rehabilitation robotics: pilot trial of a spatial extension for MIT-Manus," *Journal of NeuroEngineering and Rehabilitation,* vol. 1, no. 1, p. 5, 2004.

[37] S. E. Fasoli, H. I. Krebs and N. Hogan, "Robotic technology and stroke rehabilitation: translating research into practice," *Topics in Stroke Rehabilitation,* vol. 11, no. 4, p. 11–19, 2004.

[38] M. Ferraro, J. J. Palazzolo, J. Krol, H. I. Krebs, N. Hogan and B. T. Volpe, "Robot-aided sensorimotor arm training improves outcome in patients with chronic stroke," *Neurology,*

vol. 61, no. 11, p. 1604–1607, 2003.

[39] S. E. Fasoli, H. I. Krebs, J. Stein, W. R. Frontera, R. Hughes and N. Hogan, "Robotic therapy for chronic motor impairments after stroke: Follow-up results," *Archives of Physical Medicine and Rehabilitation,* vol. 85, no. 7, p. 1106–1111, 2004.

[40] L. E. Kahn, P. S. Lum, W. Z. Rymer and D. J. Reinkensmeyer, "Robot-assisted movement training for the stroke-impaired arm: Does it matter what the robot does?," *Journal of Rehabilitation Research and Development,* vol. 43, no. 5, p. 619, 2006.

[41] P. S. Lum, C. G. Burgar, P. C. Shor, M. Majmundar and M. V. d. Loos, "Robot-assisted movement training compared with conventional therapy techniques for the rehabilitation of upper-limb motor function after stroke," *Archives of Physical Medicine and Rehabilitation,* vol. 83, no. 7, p. 952–959, 2002.

[42] P. S. Lum, C. G. Burgar, M. V. d. Loos, P. C. Shor, M. Majmundar and R. Yap, "The MIME robotic system for upper-limb neuro-rehabilitation: Results from a clinical trial in subacute stroke," in *9th International Conference on Rehabilitation Robotics (ICORR)*, 2005.

[43] P. S. Lum, C. G. Burgar, M. V. d. Loos, P. C. Shor, M. Majmundar and R. Yap, "MIME robotic device for upper-limb neurorehabilitation in subacute stroke subjects: A follow-up study," *Journal of Rehabilitation Research and Development,* vol. 43, no. 5, p. 631, 2006.

[44] P. S. Lum, C. G. Burgar and P. C. Shor, "Evidence for improved muscle activation patterns after retraining of reaching movements with the MIME robotic system in subjects with

post-stroke hemiparesis," *IEEE Transactions on Neural Systems and Rehabilitation Engineering,* vol. 12, no. 2, pp. 186-194, 2004.

[45] P. S. Lum, C. G. Burgar, P. C. Shor, M. Majmundar and M. V. L. der, "“Robot-assisted movement training compared with conventional therapy techniques for the rehabilitation of upper-limb motor function after stroke," *Archives of Physical Medicine and Rehabilitation,* vol. 83, no. 7, pp. 952-959, 2002.

[46] S. E. Fasoli, H. I. Krebs, J. Stein, W. R. Frontera and N. Hogan, "Effects of robotic therapy on motor impairment and recovery in chronic stroke," *Archives of Physical Medicene and Rehabilitation,* vol. 84, no. 4, pp. 477-482, 2003.

[47] S. Hesse, C. Werner, M. Pohl, S. Rueckriem, J. Mehrholz and M. L. Lingnau, "Computerized Arm Training Improves the Motor Control of the Severely Affected Arm After Stroke A Single-Blinded Randomized Trial in Two Centers," *Stroke,* vol. 36, no. 9, p. 1960–1966, 2005.

[48] D. J. Gladstone, C. J. Danells and S. E. Black, "The Fugl-Meyer assessment of motor recovery after stroke: a critical review of its measurement properties," *Neurorehabilitation and Neural Repair,* vol. 16, no. 3, p. 232–240, 2002.

[49] T. Brassil and J. M. Brassil, "Hand rehabilitation glove". US Patent 6,454,681, 2002.

[50] J. F. Farrell, H. B. Hoffman, J. L. Snyder, C. A. Giuliani and R. W. Bohannon, "Orthotic aided training of the paretic upper limb in chronic stroke: results of a phase 1 trial,"

NeuroRehabilitation, vol. 22, no. 2, p. 99–103, 2007.

[51] E. Carmeli, S. Peleg, G. Bartur, E. Elbo and J.-J. Vatine, "HandTutorTM: enhanced hand rehabilitation after stroke—a pilot study," *Physiotherapy Research International,* vol. 16, no. 4, p. 191–200, 2011.

[52] P. De Leva, "Adjustments to Zatsiorsky-Seluyanov's segment inertia parameters," *Journal of Biomechanics,* vol. 29, no. 9, pp. 1223-1230, 1996.

[53] J. Pratt, B. Krupp and C. Morse, "Series elastic actuators for high fidelity force control," *Industrial Robot: An International Journal,* vol. 29, no. 3, pp. 234-241, 2002.

[54] M. Mozhanova, *Design of a High-Resolution Surface Electromyogram (EMG) Conditioning Circuit,* Worcester Polytechnic Institute, 2012, Available online at: http://www.wpi.edu/Pubs/E-project/Available/E-project-011212-103328/unrestricted/Mozhanova_MQP_EMG_1.pdf.

[55] Stratasys, Inc., *ABSplus-P430 Datasheet,* 2013.

Appendices

A. Main Board Schematics

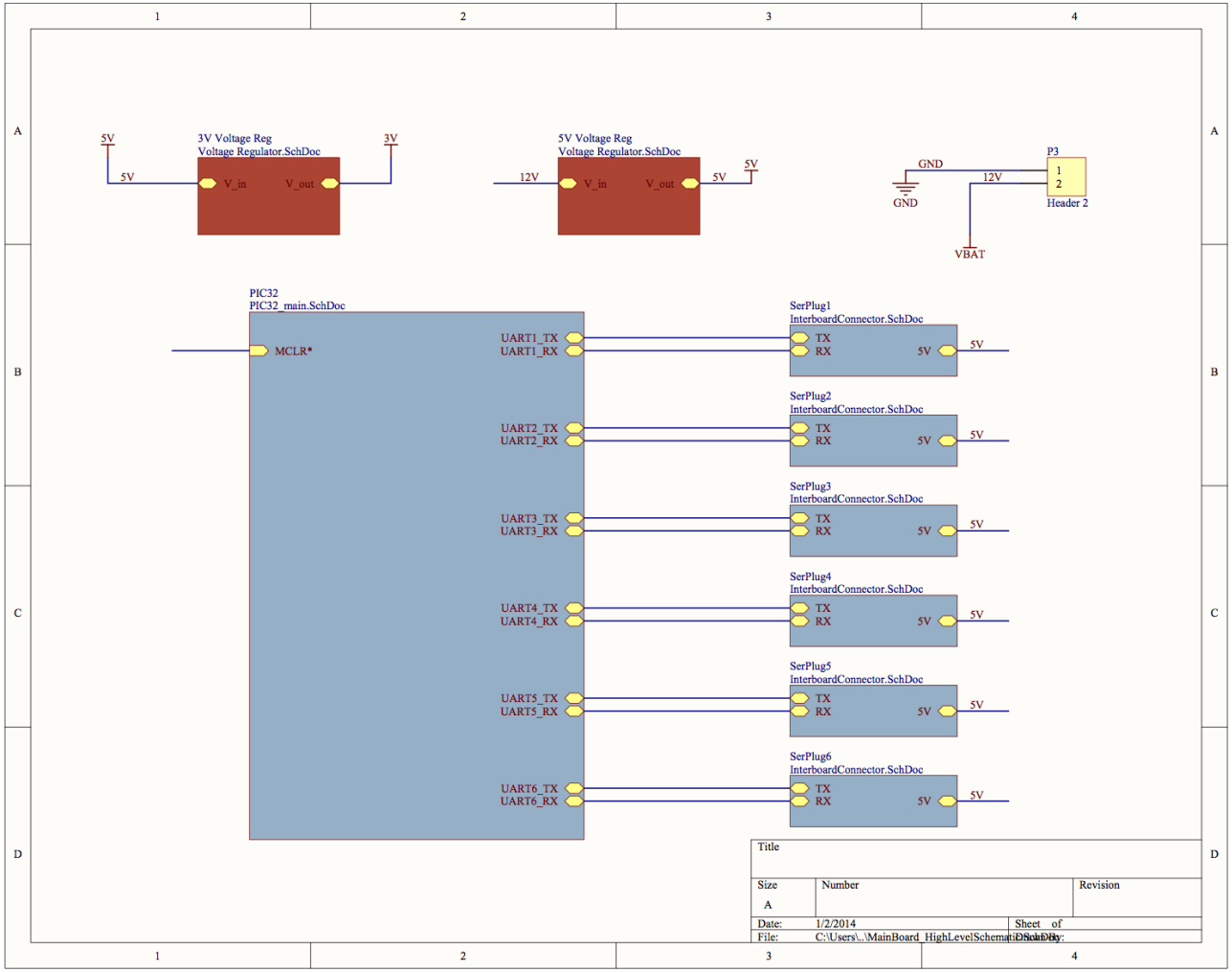


Figure A.1: Main Board High-Level Schematic

B. Main Board PCB

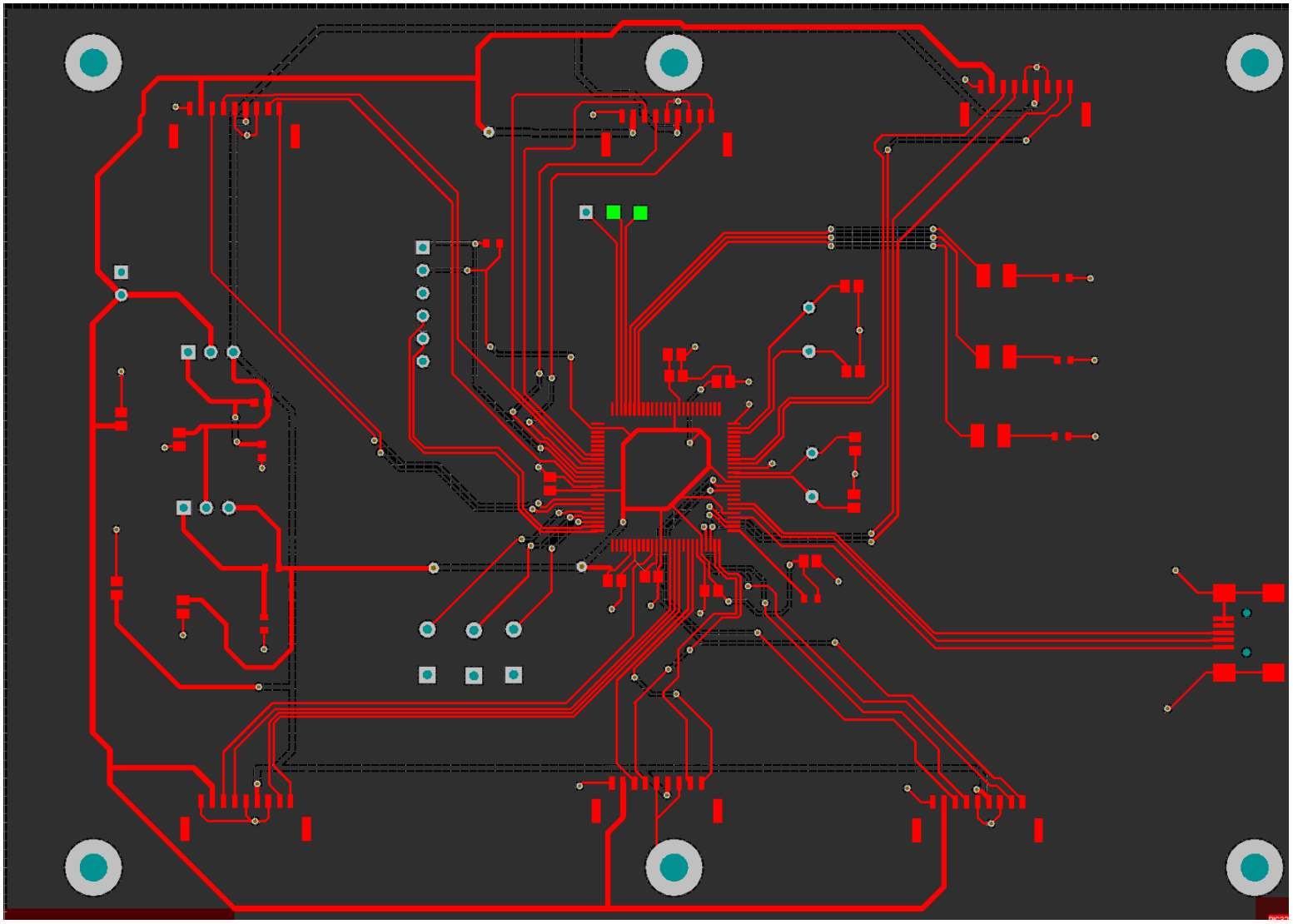


Figure B.1: Main Board PCB, Altium Front View

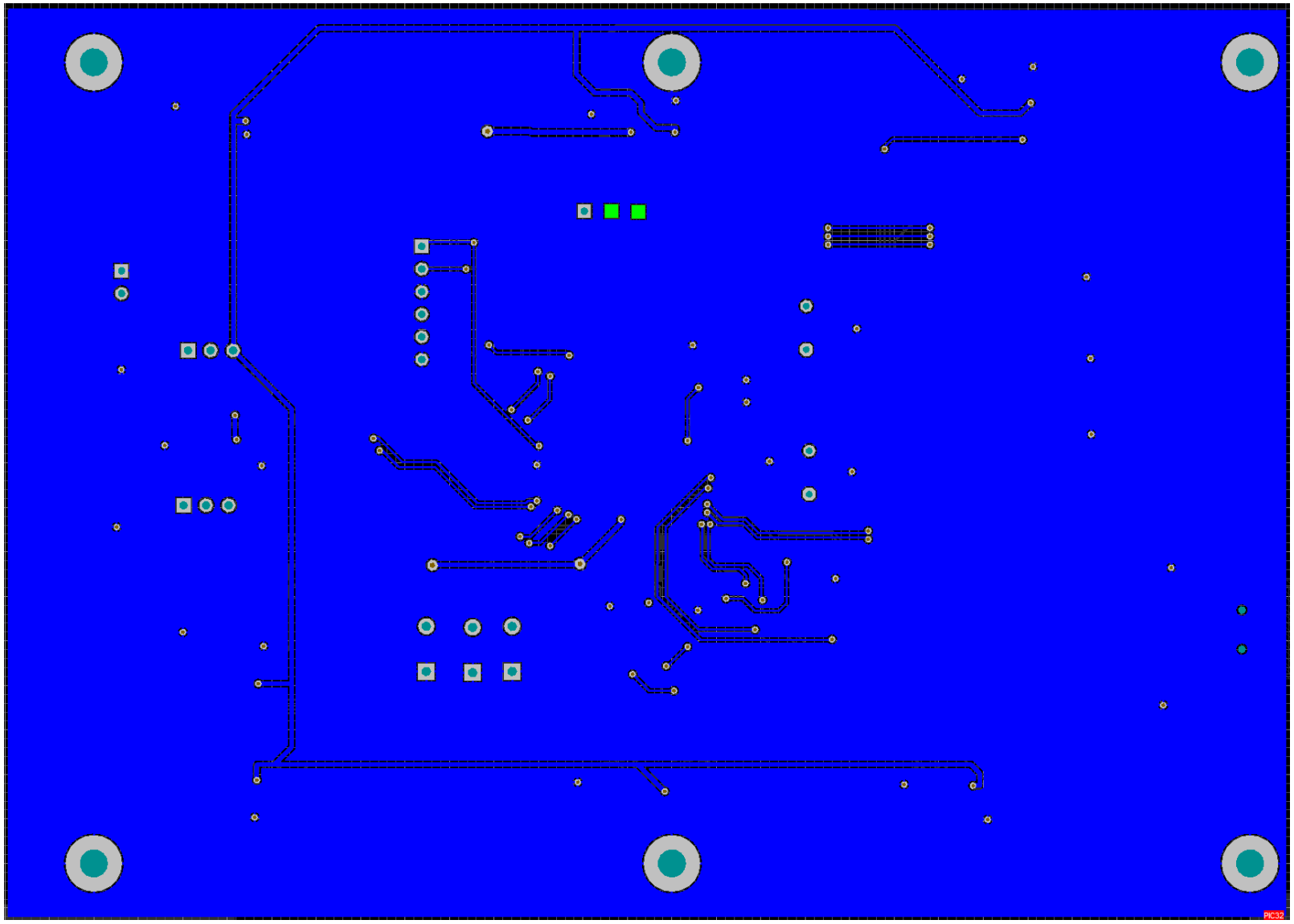


Figure B.2: Main Board PCB, Altium Rear View

C. Sensor Board Schematics

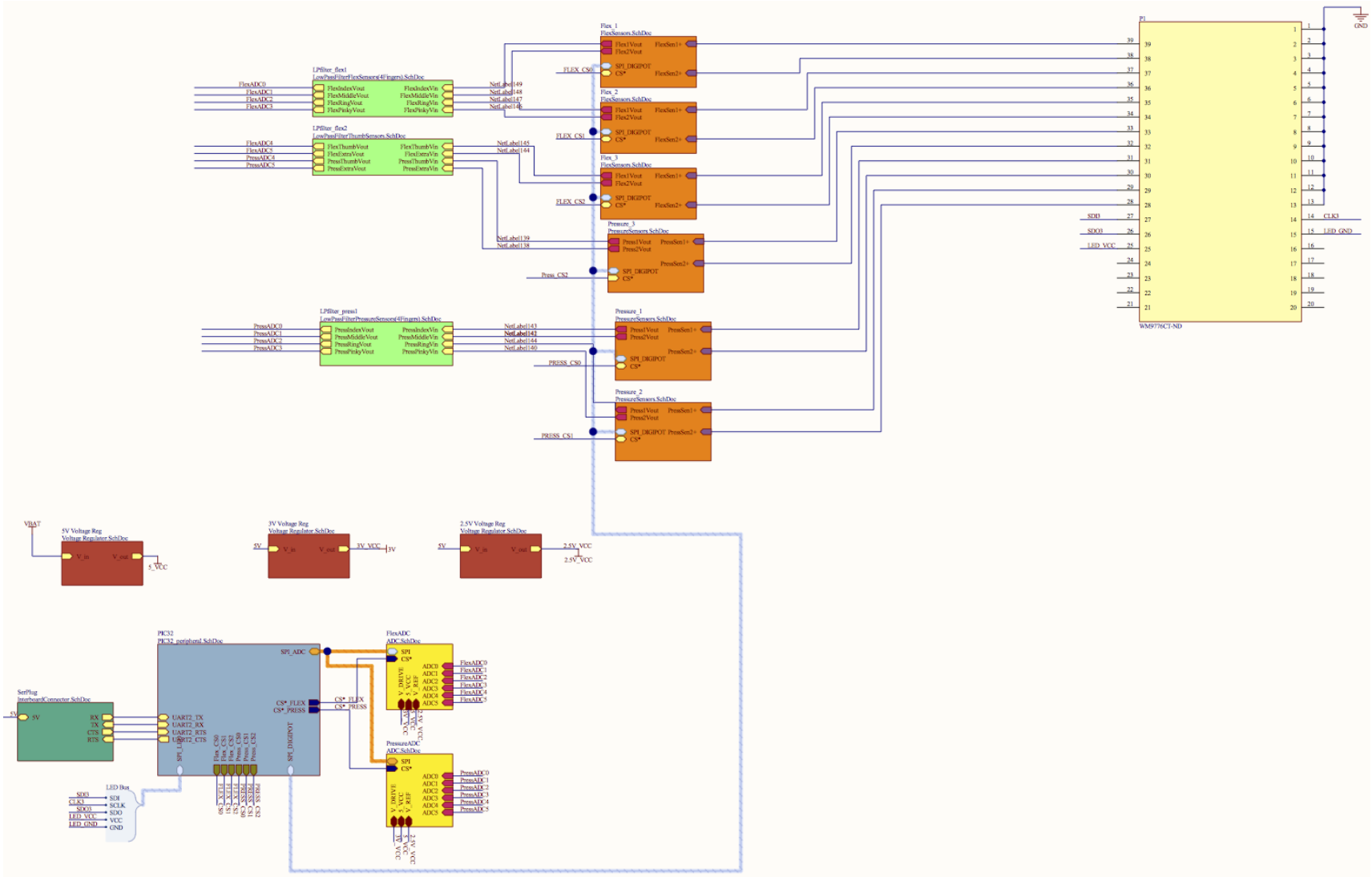


Figure C.1: Sensor Board High-Level Schematic

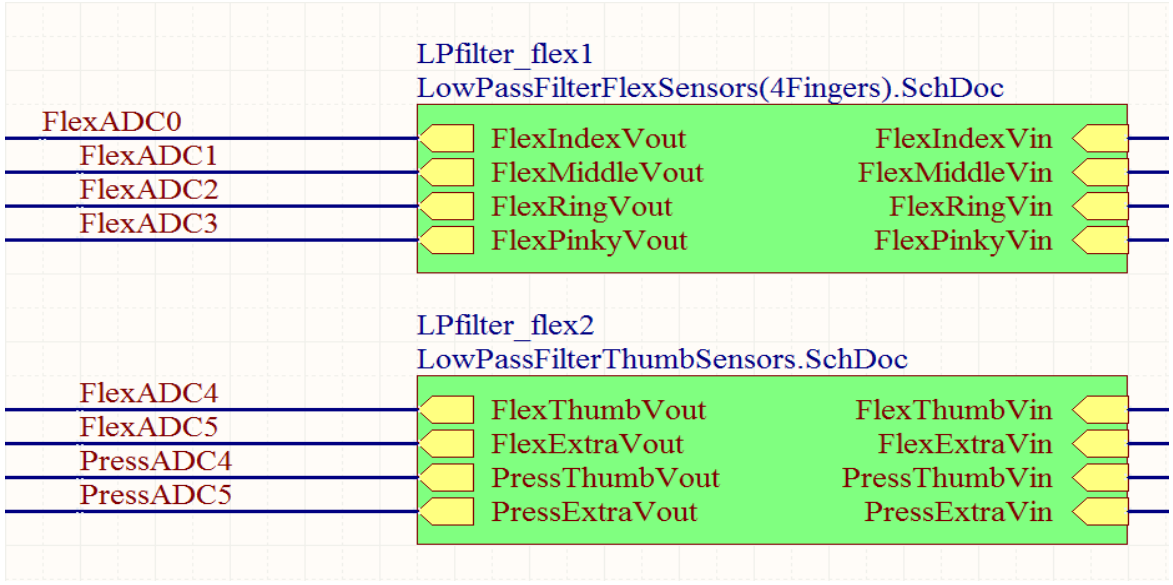


Figure C.2: Zoom-in of Flex Sensor Low-Pass Filters Included in the Sensors Board Schematic Displayed Above

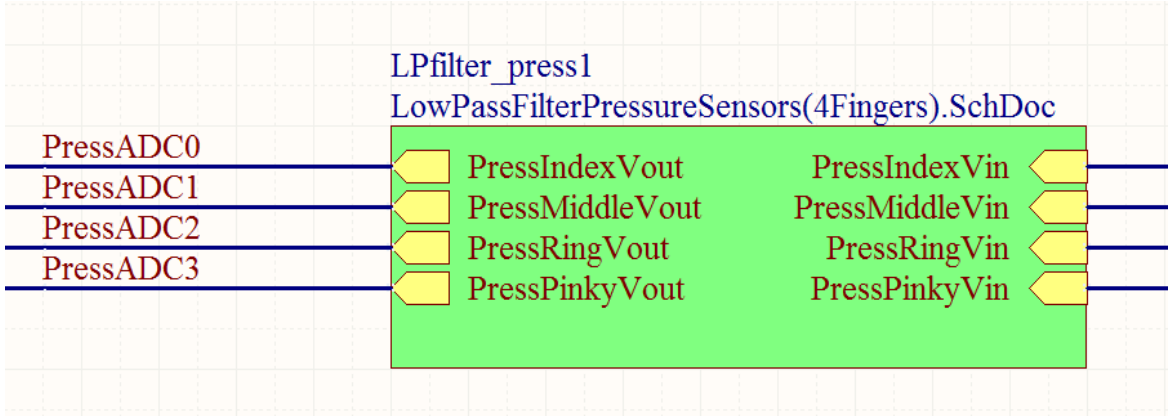


Figure C.3: Zoom-in of Pressure Sensor Low-Pass Filters Included in the Sensors Board Schematic Displayed Above

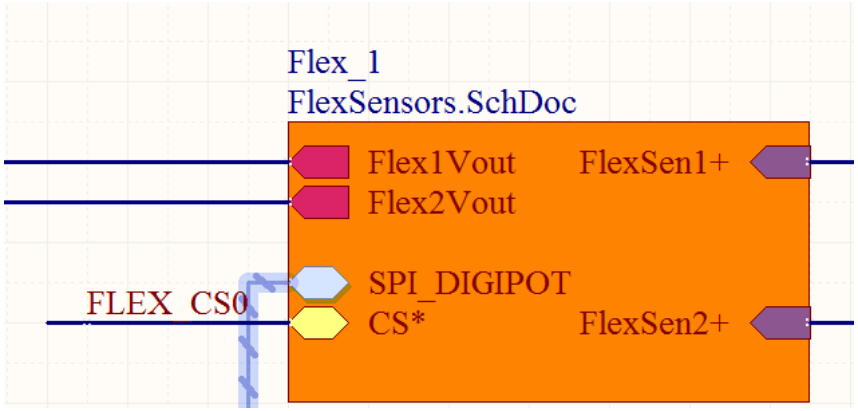


Figure C.4: Zoom-in of Flex Sensors Included in the Sensors Board Schematic

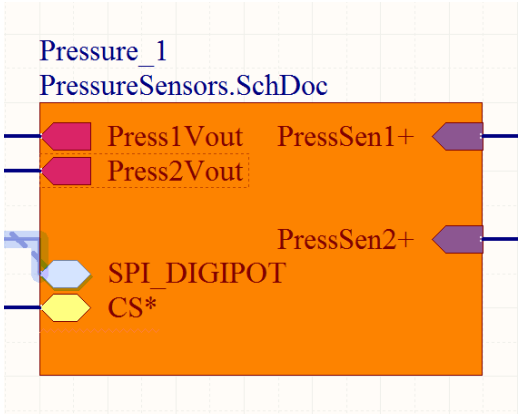
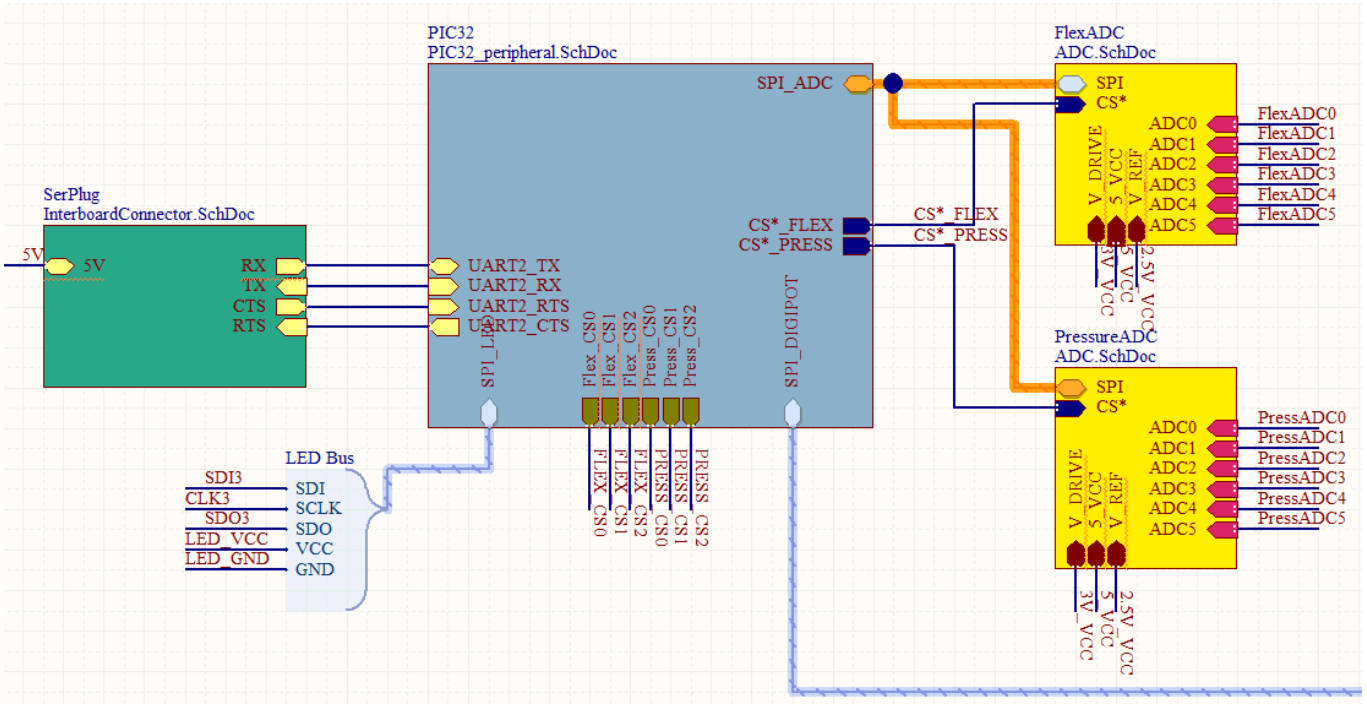


Figure C.5: Zoom-in of Flex Sensors Included in the Sensors Board Schematic



D. Sensor Board PCB

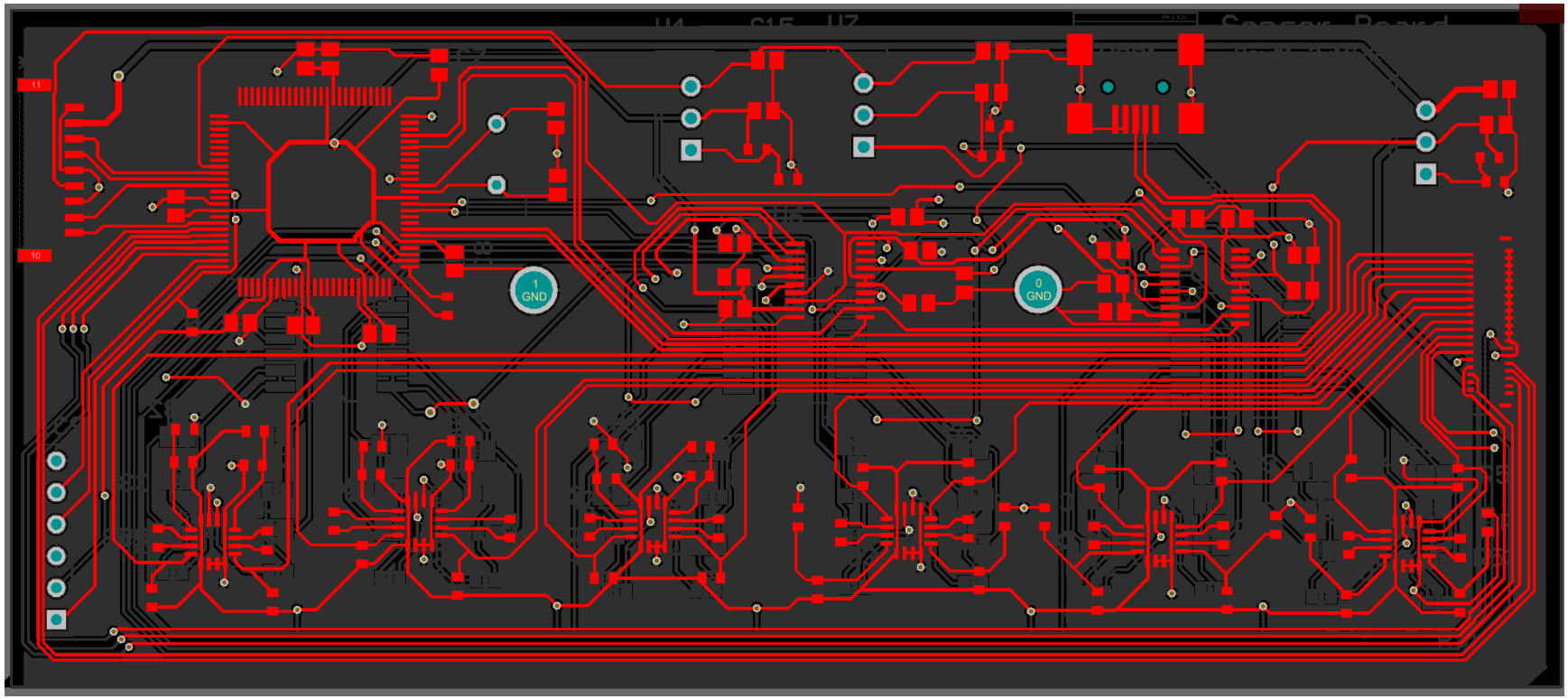


Figure D.1: Sensor Board PCB, Altium Front View

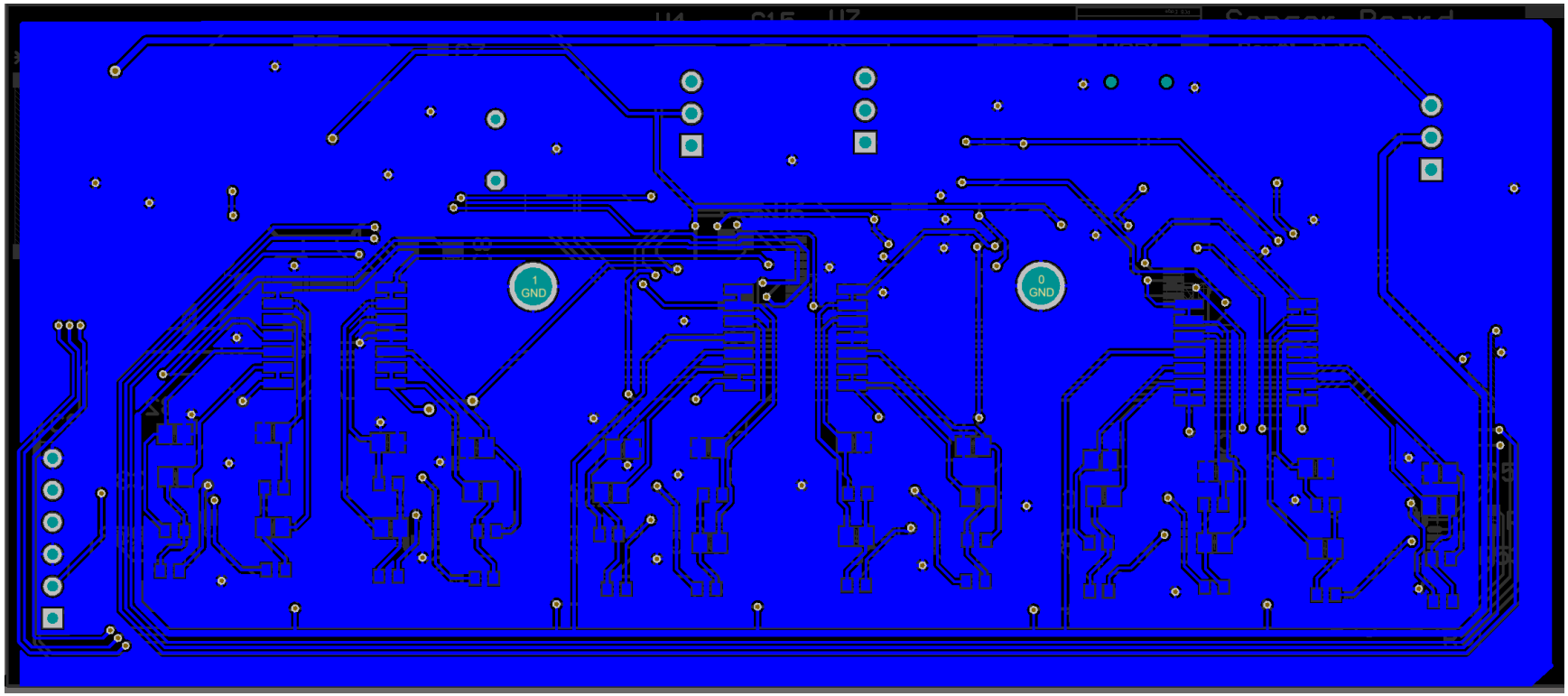


Figure D.2: Sensor Board PCB, Altium Rear View

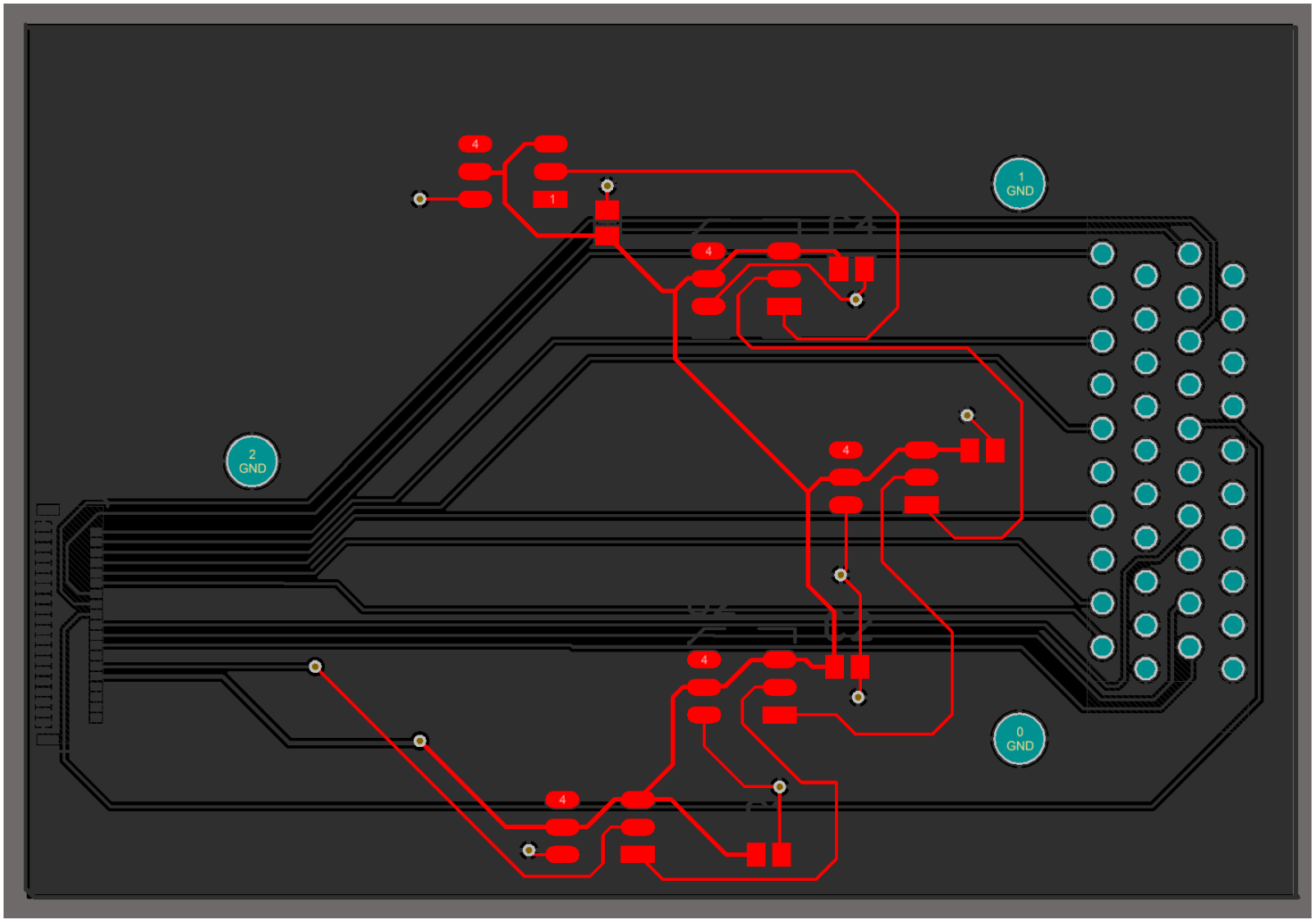


Figure D.3: Sensor Board PCB, Altium Front View (LEDS-section)

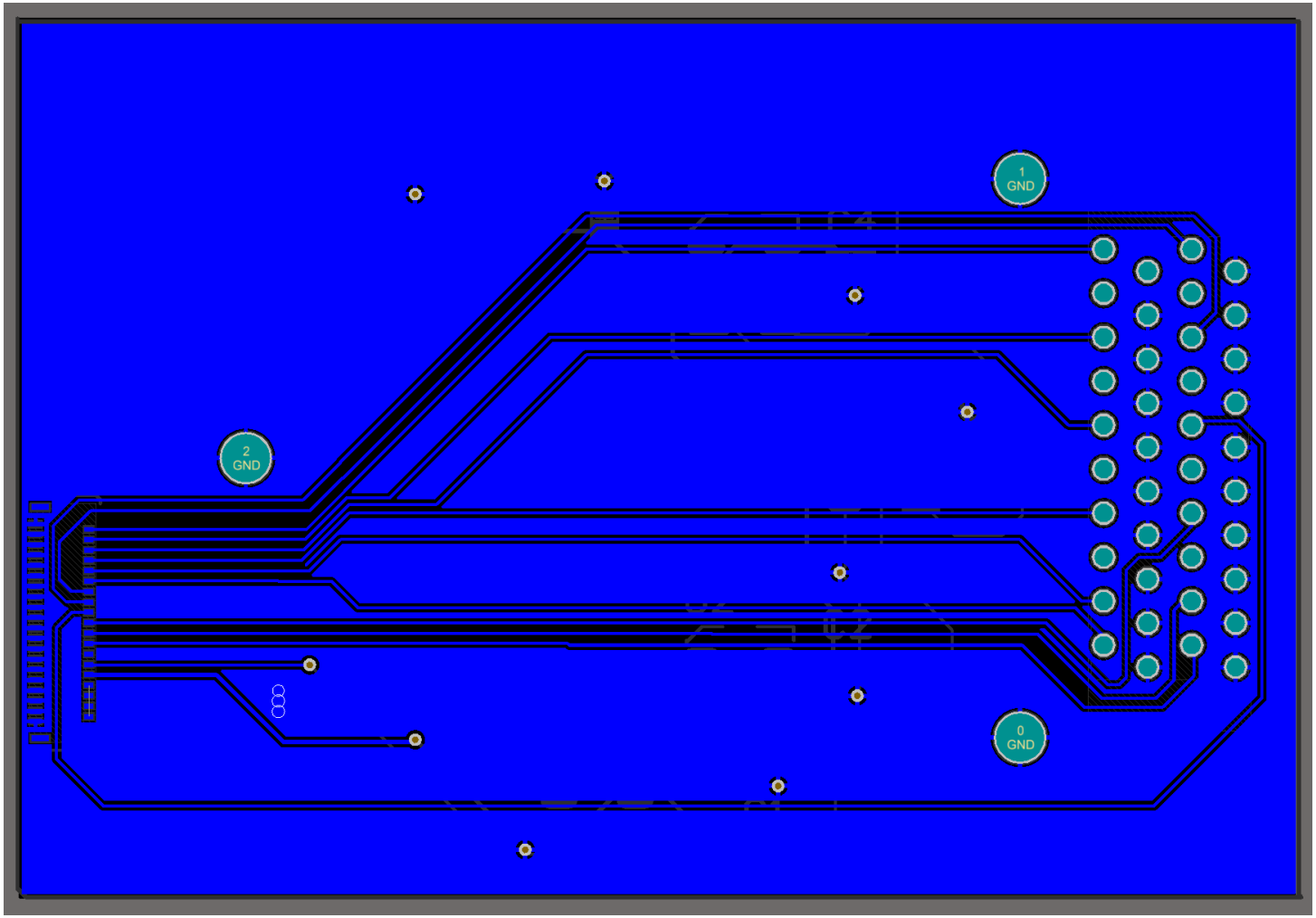


Figure D.4: Sensor Board PCB, Altium Rear View (LEDS-section)

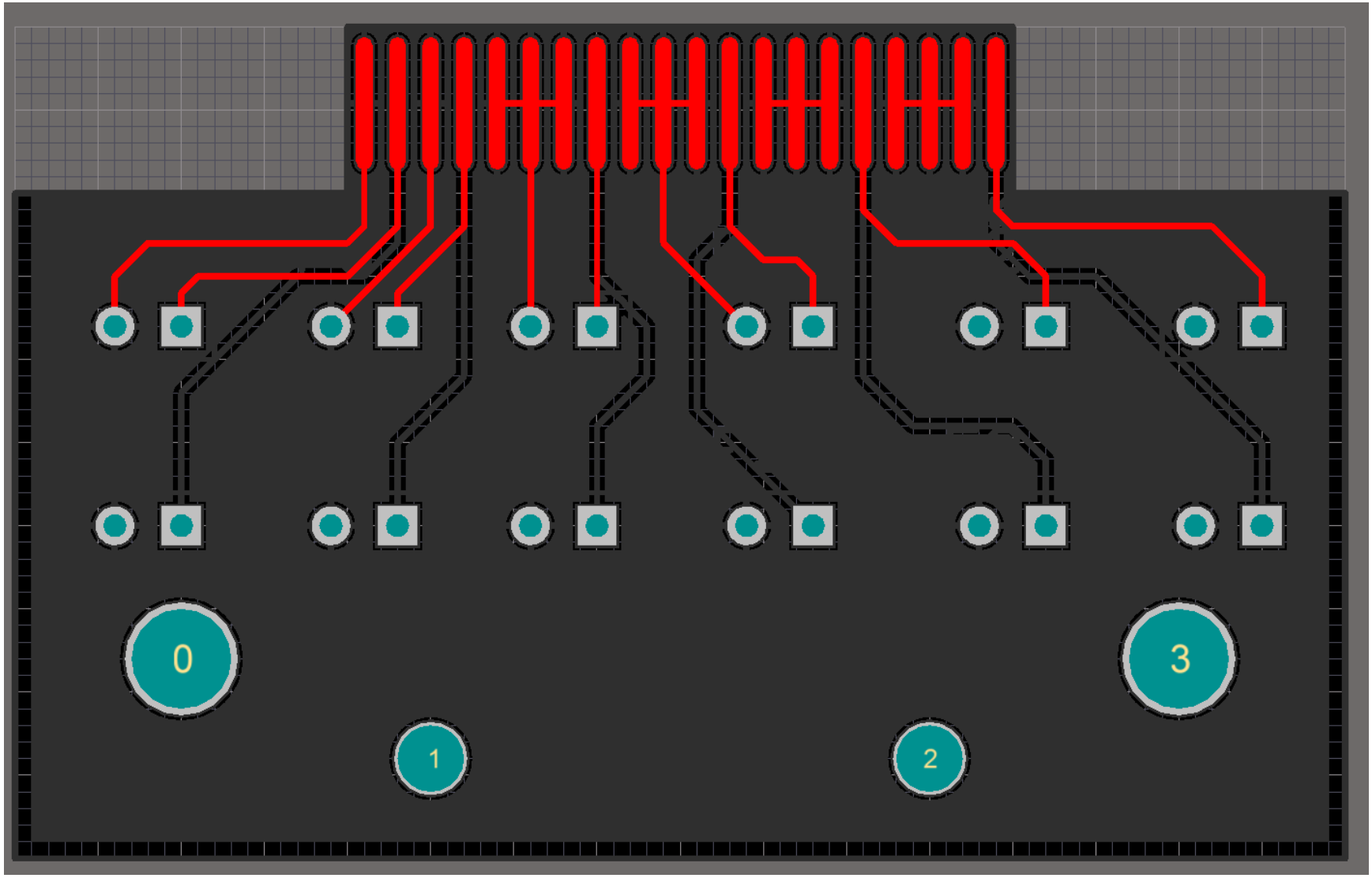


Figure D.5: Sensor Board PCB, Altium Front View (Flex and Pressure sensor connection section)

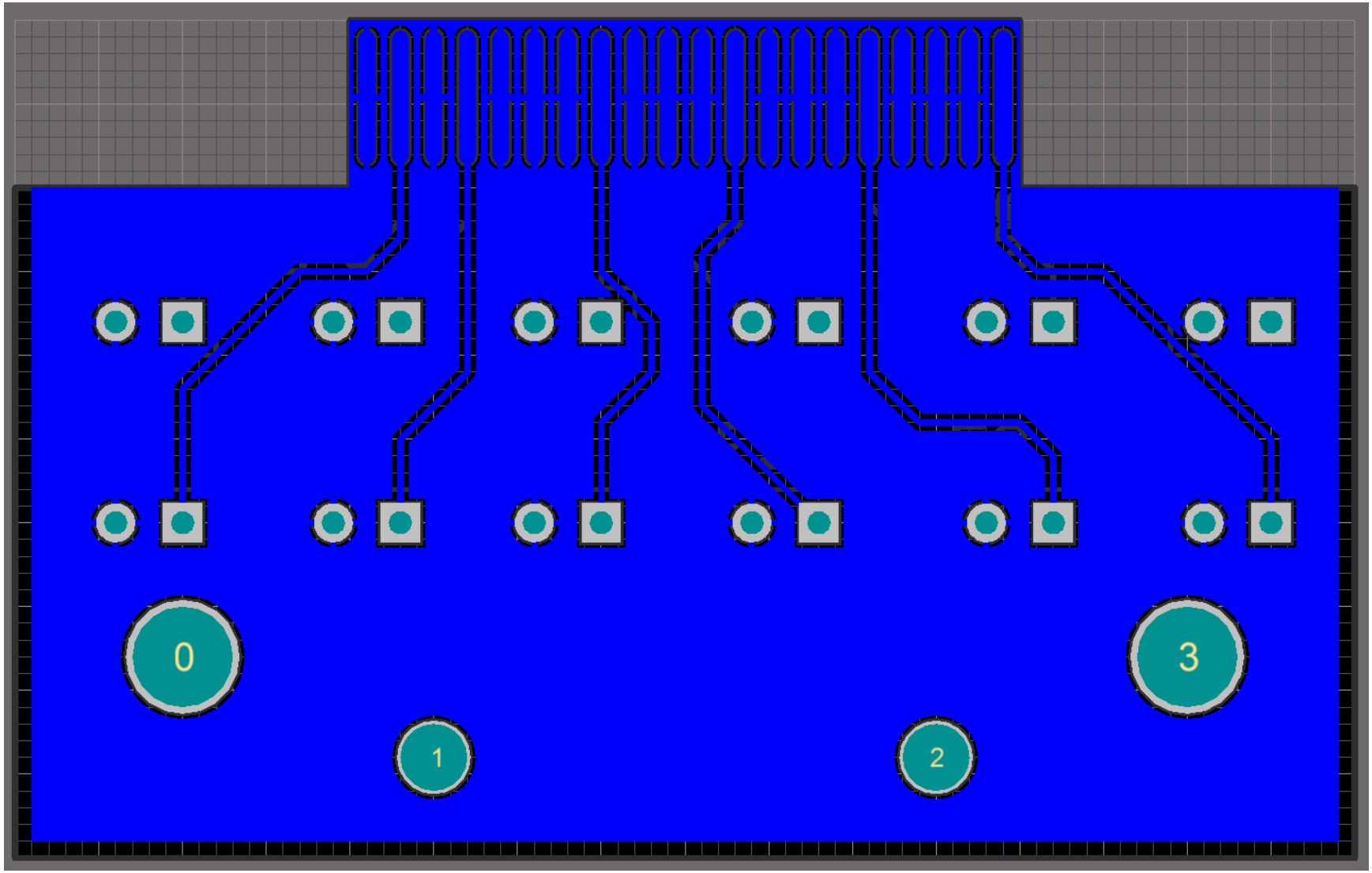


Figure D.6: Sensor PCB, Altium Rear View (Board Flex and Pressure sensor connection section)

E. Motor Board Schematics

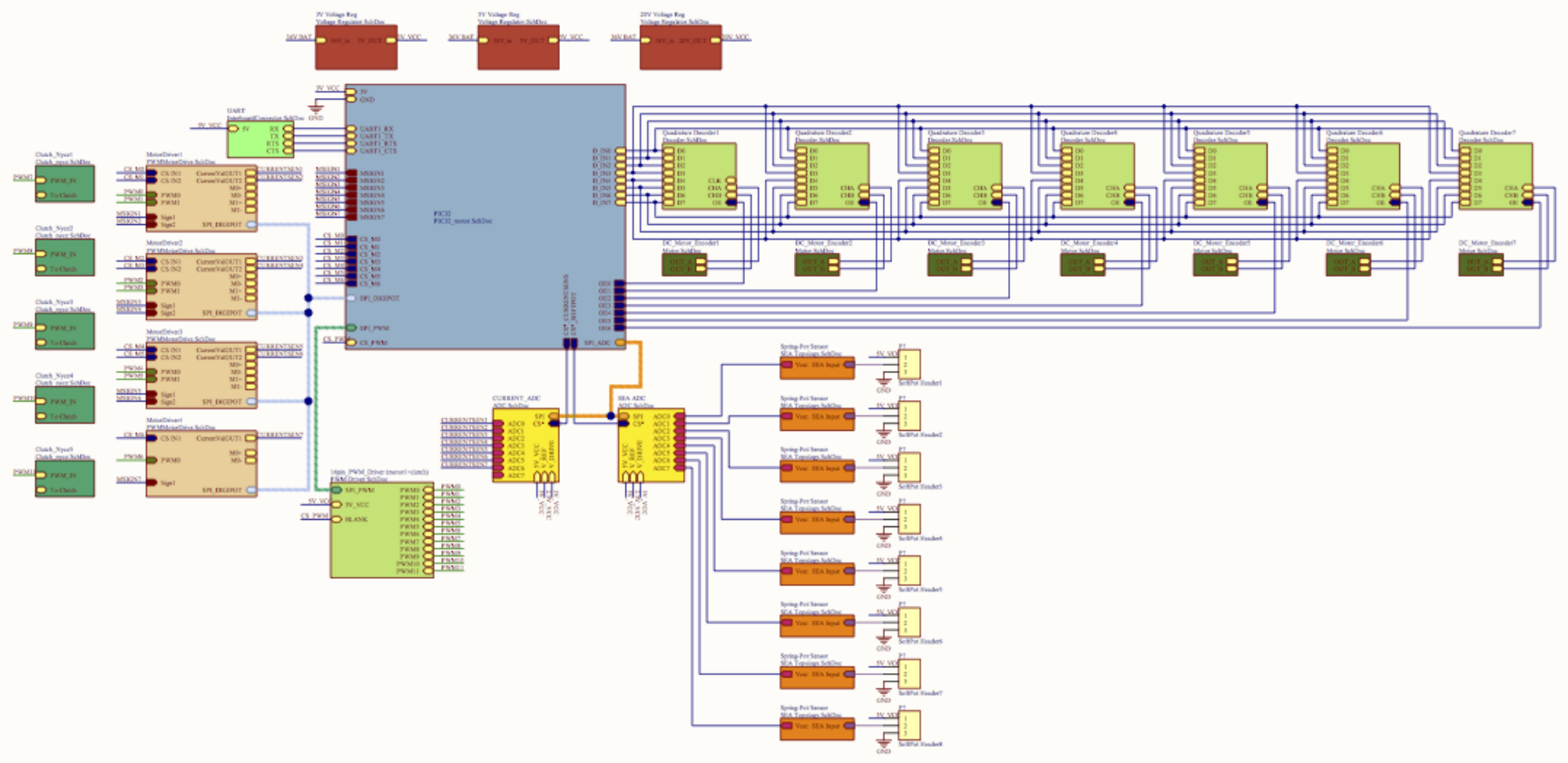


Figure E.1: Motor Board High-Level Schematic

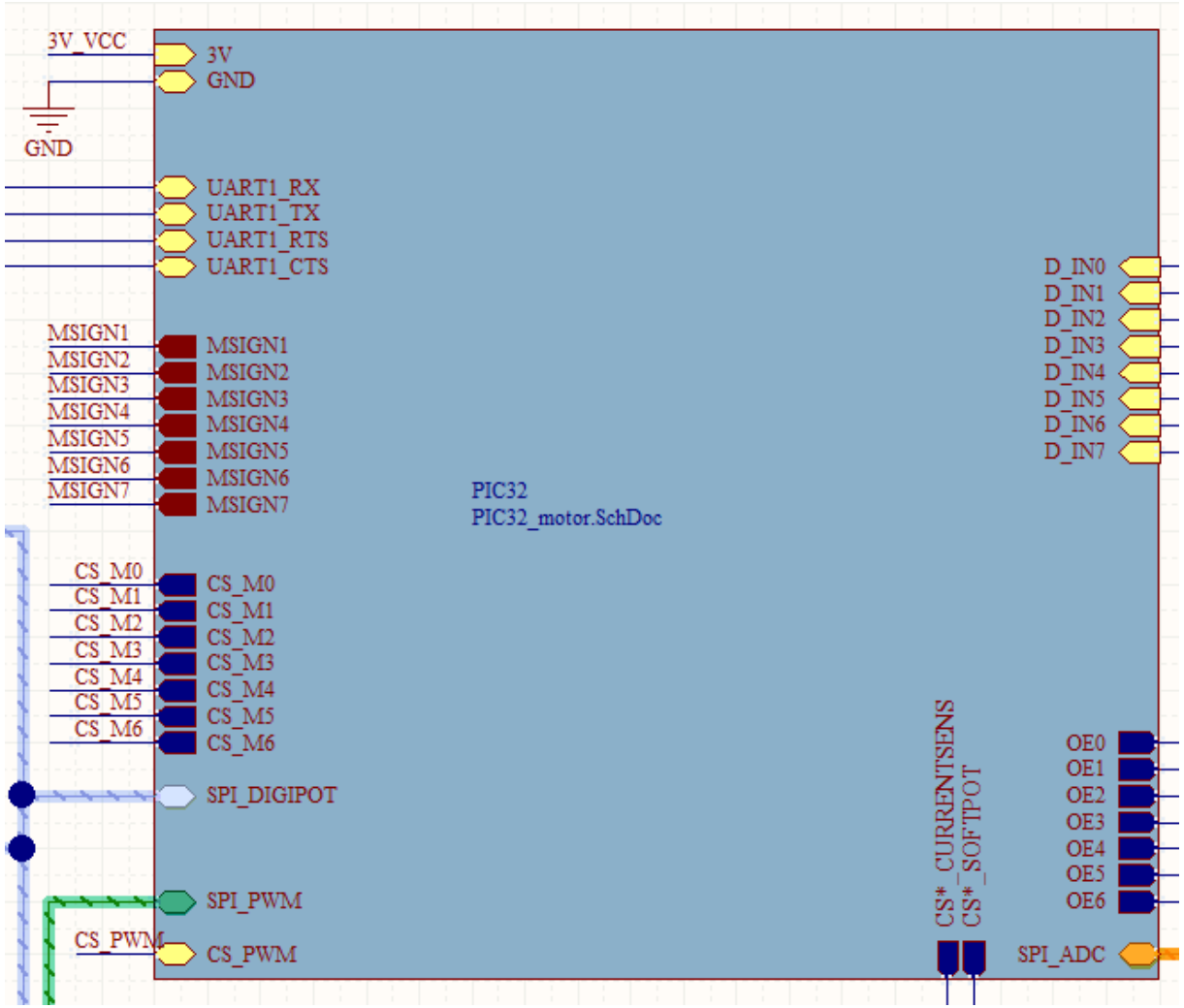


Figure E.2: Zoom-in of PIC32 Included in the Motor Board Schematic Displayed Above

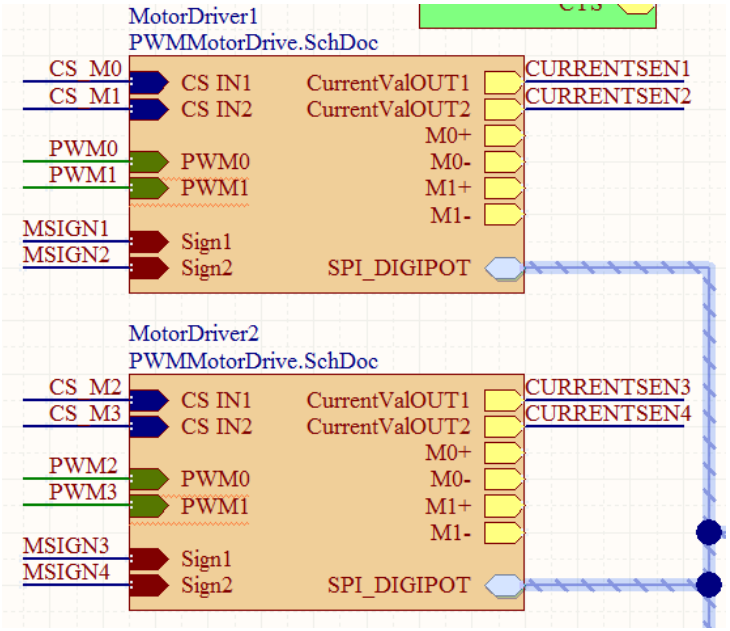


Figure E.3: Zoom-in of Motor Drivers Included in the Motor Board Schematic

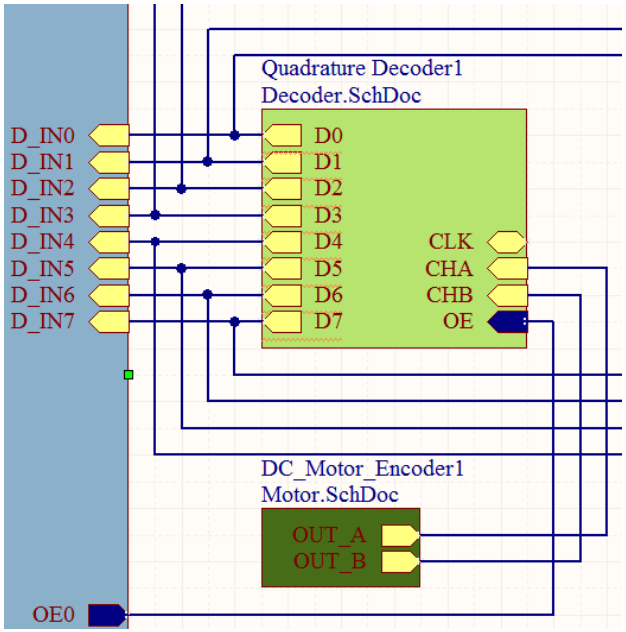


Figure E.4: Zoom-in of Quadrature Encoders and DC Motor Encoders Included in the Motor Board Schematic

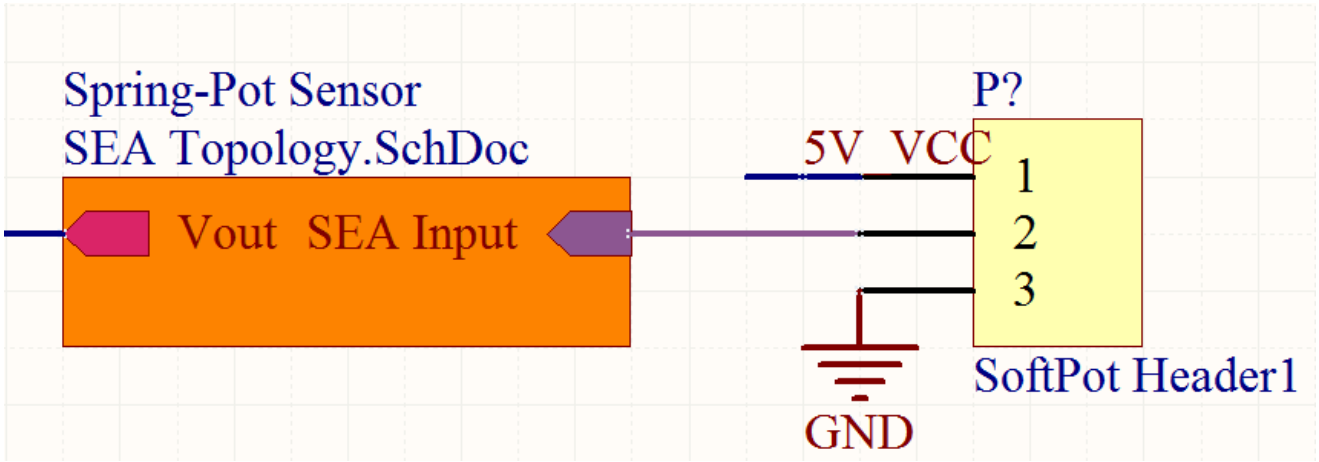


Figure E.5: Zoom-in of Series Elastic Actuators and Soft Potentiometers Included in the Motor Board Schematic

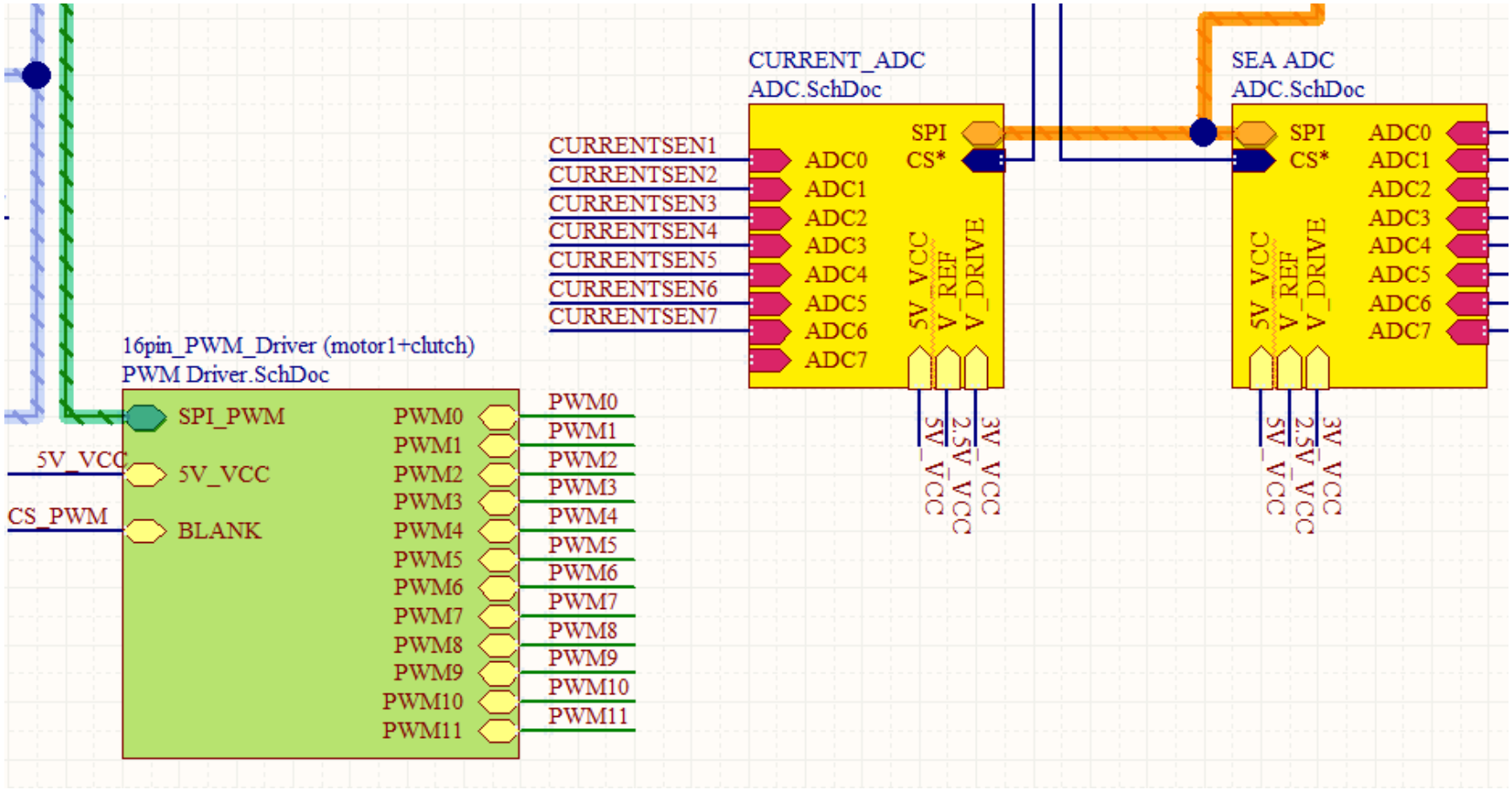


Figure E.6: Zoom-in of PWM Drivers and ADCs Included in the Motor Board Schematic

F. Motor Board PCB

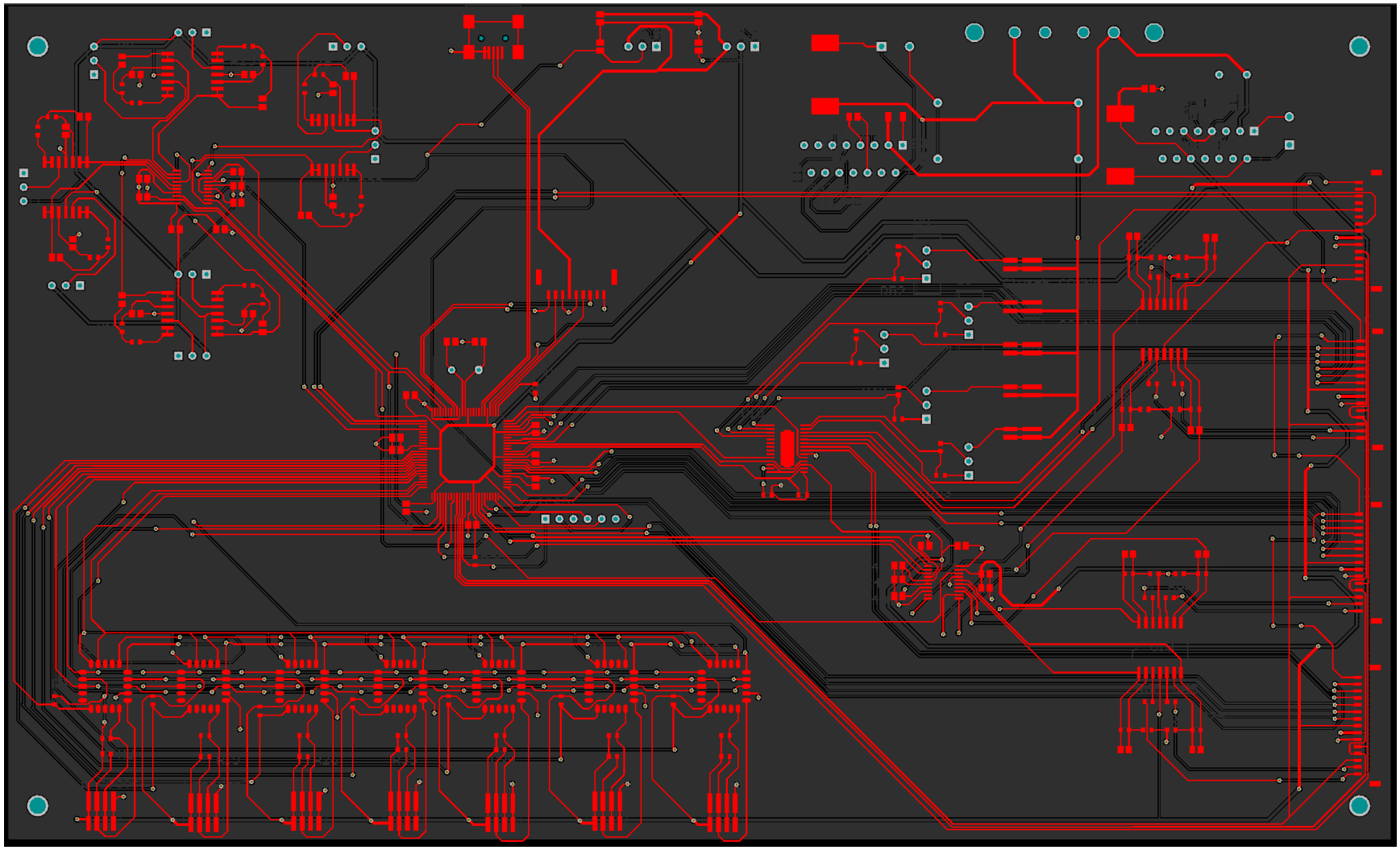


Figure F.1: Motor Board PCB, Altium Front View

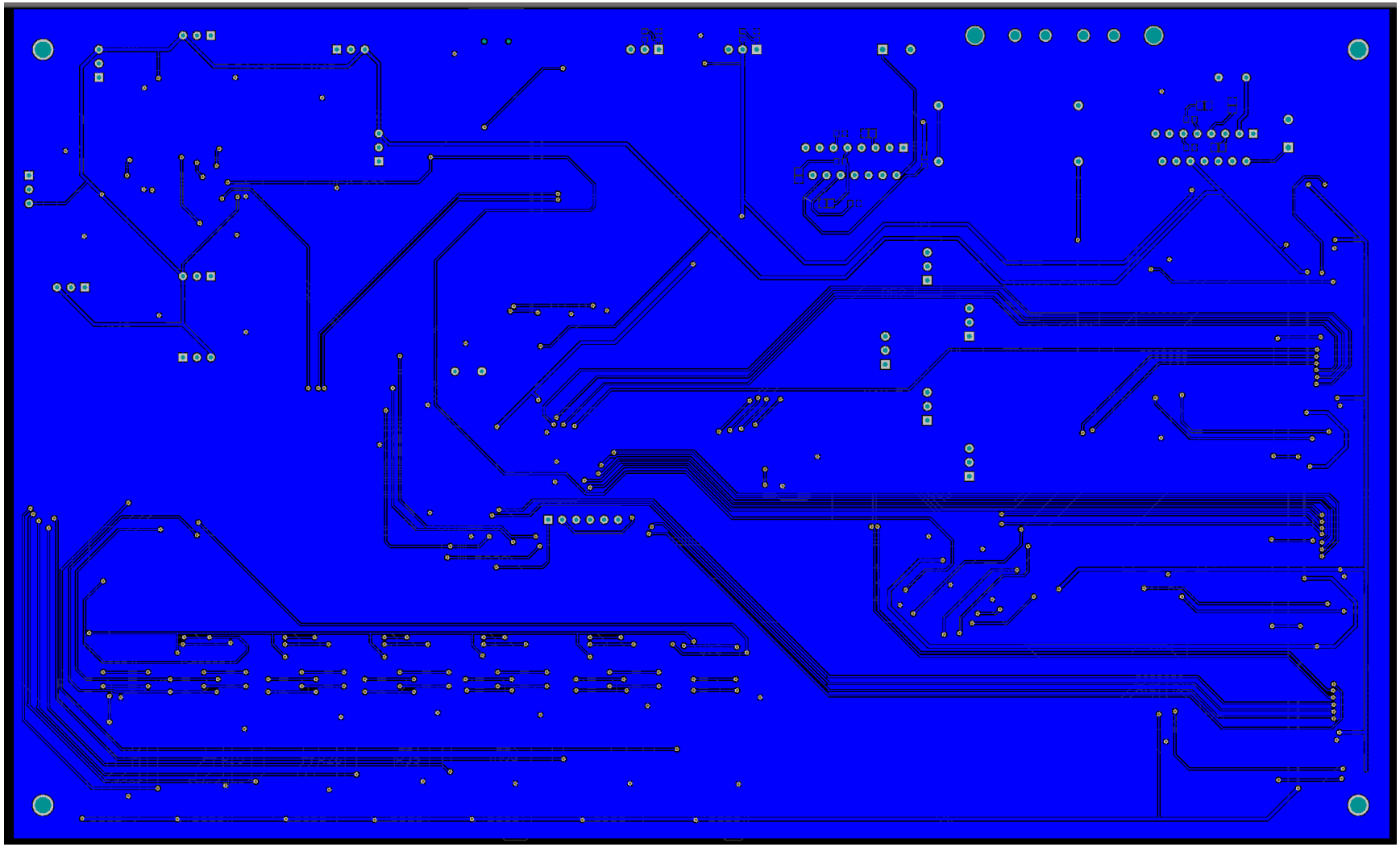


Figure F.2: Motor Board PCB, Altium Rear View

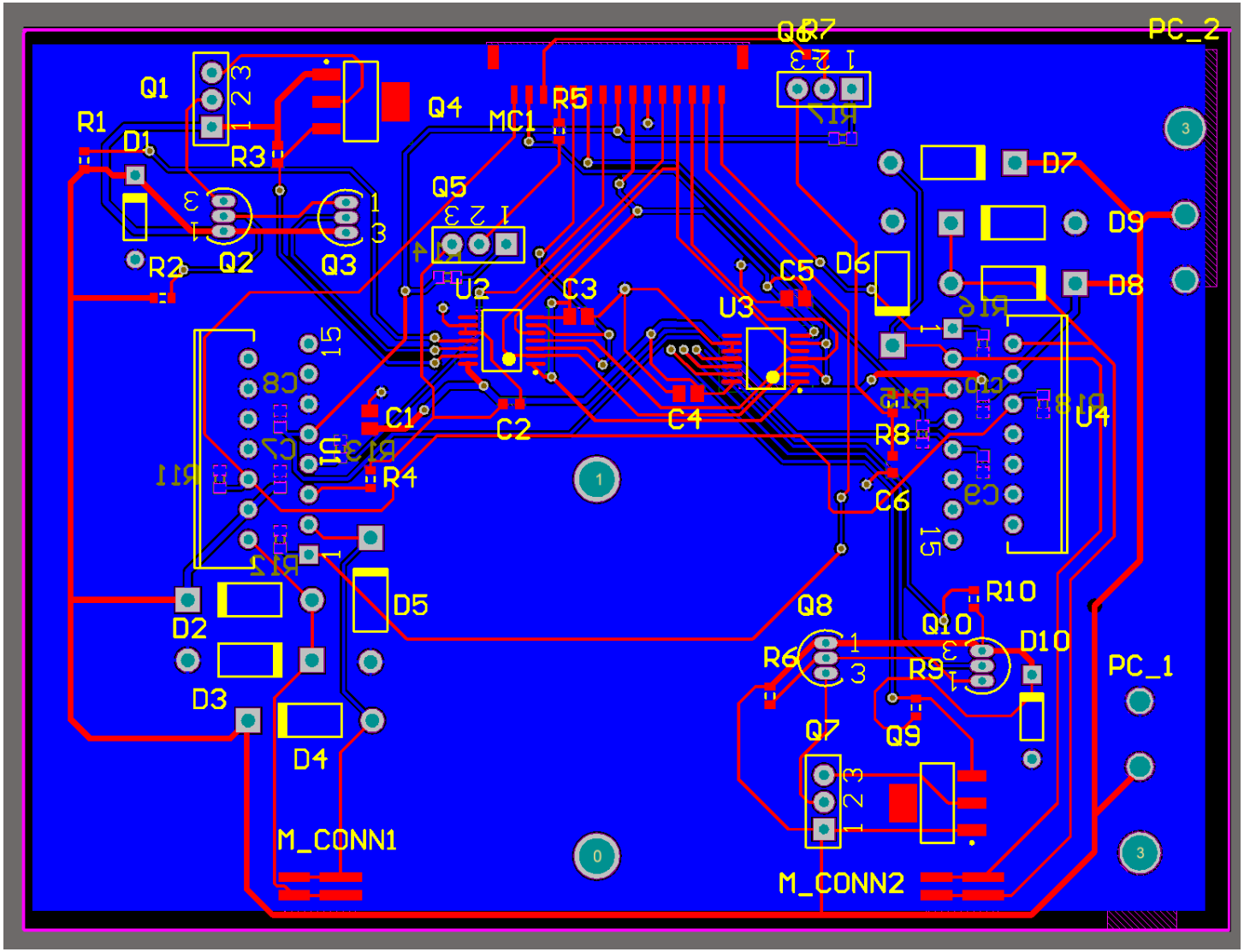


Figure F.3: Motor Board Modules PCB, Altium Front View

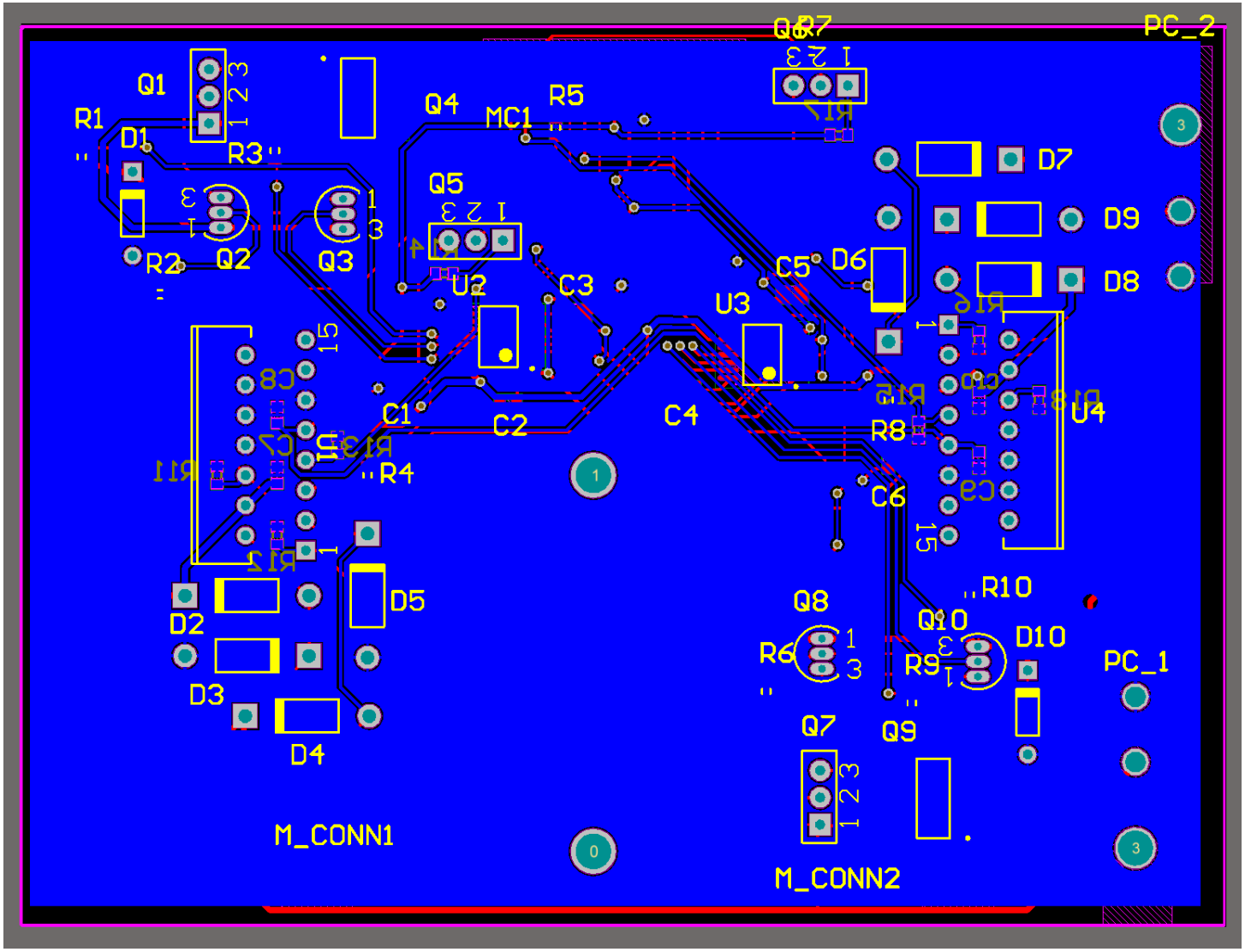


Figure F.4: Motor Board Modules PCB, Altium Rear View

G. Main Board Software

a. Main Header File

```
#ifndef MAINBOARD_H
#define MAINBOARD_H

#ifdef  __cplusplus
extern "C" {
#endif

/************************/
/*- Pragma Directives -*/
/************************/
#ifdef USB_A0_SILICON_WORK_AROUND
        #pragma config UPLLEN   = OFF      // USB PLL Enabled (A0 bit inverted)
#else
        #pragma config UPLLEN   = ON       // USB PLL Enabled
#endif

#pragma config FPLLMUL  = MUL_20       // PLL Multiplier
#pragma config UPLLIDIV = DIV_2        // USB PLL Input Divider
#pragma config FPLLIDIV = DIV_2        // PLL Input Divider
#pragma config FPLLODIV = DIV_1        // PLL Output Divider
#pragma config FPBDIV   = DIV_1        // Peripheral Clock divisor
#pragma config FWDTEN   = OFF          // Watchdog Timer
#pragma config WDTPS    = PS1          // Watchdog Timer Postscale
#pragma config FCKSM    = CSDCMD       // Clock Switching & Fail Safe Clock Monitor
#pragma config OSCIOFNC = OFF          // CLKO Enable
#pragma config POSCMOD  = HS           // Primary Oscillator
#pragma config IESO     = ON           // Internal/External Switch-over
#pragma config FSOSCEN  = ON           // Secondary Oscillator Enable (KLO was off)
#pragma config FNOSC    = PRIPLL       // Oscillator Selection
#pragma config CP       = OFF          // Code Protect
```

```
#pragma config BWP      = OFF          // Boot Flash Write Protect
#pragma config PWP      = OFF          // Program Flash Write Protect
#pragma config ICESEL   = ICS_PGx2     // ICE/ICD Comm Channel Select
#pragma config DEBUG    = OFF          // Background Debugger Enable



/************************/
/*- Include Directives -*/
/************************/
#include "Bowler/Bowler.h"
#include "commonLib.h"
#include "commonUART.h"
#include "commonTimer.h"



/************************/
/*- Macro Definitions -*/
/************************/
//#define COMPUTER_UART                 UART5
//#define COMPUTER_UART_VECTOR    _UART_5_VECTOR

#define MAIN_SENSOR_UART            UART2
#define MAIN_SENSOR_UART_VECTOR           _UART_2_VECTOR

#define MOTOR_UART                    UART4
#define MOTOR_UART_VECTOR           _UART_4_VECTOR

#define TEN_K_TIMER                   3
#define TEN_K_TIMER_VECTOR          _TIMER_3_VECTOR
#define TEN_K_TIMER_CLEAR_FLAG      INT_T3

#define SYS_CLK_FRQ                   (80000000ul)
```

```
#define BAUD_RATE                       (9600)

#define SENSOR_TRIS_LED                 _TRISG12
#define SENSOR_LED                      _RG12

#define MOTOR_TRIS_LED                  _TRISG14
#define MOTOR_LED                       _RG14

//#define SENSOR_TRIS_LED               _TRISD0
//#define SENSOR_LED                    _RD0
//
//#define MOTOR_TRIS_LED                _TRISD1
//#define MOTOR_LED                     _RD1

//#define COMP_TRIS_LED                 _TRISE1
//#define COMP_LED                      _RE1

#define EXOARM_STORE_BUF_SIZE       10



/************************/
/*- Structures & Unions -*/
/************************/

        typedef struct _FingerStore {
                short int encoderPos;
                short int encoderVel;
                unsigned short int currentADC;
                unsigned short int seriesElasticADC;
                unsigned short int flexADC;
                unsigned short int presADC;
        } FingerStore;
```

```
typedef struct _WristStore {
	short int encoderPos;
	short int encoderVel;
	short int currentADC;
	short int flexADC;
} WristStore;

typedef struct _ElbowStore {
	short int encoderPos;
	short int encoderVel;
	unsigned short int currentADC;
	unsigned short int seriesElasticADC;
} ElbowStore;

typedef struct _ExoArmStore {
	FingerStore fingers[5];
	WristStore wrist;
	ElbowStore elbow;
} ExoArmStore;

typedef struct _ExoArmStoreBuf {
	unsigned char firstIndex;
	ExoArmStore store[EXOARM_STORE_BUF_SIZE];
} ExoArmStoreBuf;


typedef struct _Manager {
	unsigned char sensorSub[7];
	unsigned char motorSub[7];
	unsigned char motorCtrl[7];

	MainTxSensorRxPacket sensorPkt[7];
	MainTxMotorRxPacket motorPkt[7];
} Manager;
```

```
/************************/
/*- Function Prototypes -*/
/************************/
   void mainBoardInit();

   void initiateSelfTest();

   void mUpdate(Manager *m);

   void addToStore(ExoArmStoreBuf storeBuf[EXOARM_STORE_BUF_SIZE], ExoArmStore *store);

   void setMngrSensorSub(unsigned char id, unsigned char val);



/***************/
/*- Variables -*/
/***************/
   volatile unsigned long tenKTimer;

   volatile unsigned char sensorRecvBuf[SENSOR_TX_MAIN_RX_PACKET_SIZE];
   volatile unsigned char sensorRecvFlag;
   volatile unsigned char sensorRecvCount;

   volatile unsigned char motorRecvBuf[MOTOR_TX_MAIN_RX_PACKET_SIZE];
   volatile unsigned char motorRecvFlag;
   volatile unsigned char motorRecvCount;
```

```
#ifdef   __cplusplus
}
#endif

#endif   /* MAINBOARD_H */
```

b. Main Source File

```
#include "mainBoard.h"


#define server_mode

SensorTxMainRxPacket sensorPkt;
ExoArmStore exoarmStore;
Manager mngr;
unsigned short int myA[7];
//unsigned char commanded = 0;
//unsigned char commandedIndex = 0;

void setMngrMotorSub(unsigned char id, unsigned char val) {
   mngr.motorSub[id] = val;

   MainTxMotorRxPacket pkt;
   pkt.use.header._id = val;

   commonUARTSendData(MOTOR_UART, pkt.stream, MAIN_TX_MOTOR_RX_PACKET_SIZE);
}

void getMngrMotorSub(unsigned char id, unsigned char val) {

}

unsigned char getSensorPktId() {
   return sensorPkt.use.header._id;
}

void setMngrSensorSub(unsigned char id, unsigned char val) {
//    commanded = 1;
//    commandedIndex = id;
```

```
   mngr.sensorSub[id] = val;
//    myA[id] = val;

   MainTxSensorRxPacket pkt;
   pkt.use.header._id = id;
   pkt.use.header._opt = (val)?SEN_OPT_START:SEN_OPT_STOP;
   pkt.use.header._col = 0;

   commonUARTSendData(MAIN_SENSOR_UART, pkt.stream, MAIN_TX_SENSOR_RX_PACKET_SIZE);

//    _RG12 = 1;
}

unsigned char getMngrSensorSub(unsigned char id) {
   return mngr.sensorSub[id];
//    return myA[id];
}

void setFlex(unsigned char id, unsigned short f) {
   exoarmStore.fingers[id].flexADC = f;
}

unsigned short int getFlex(unsigned char id) {
   return exoarmStore.fingers[id].flexADC;
}

void setPres(unsigned char id, unsigned short p) {
   exoarmStore.fingers[id].presADC = p;
}

unsigned short int getPres(unsigned char id) {
   return exoarmStore.fingers[id].presADC;
}
```

```
short int getWristFlex() {
   return exoarmStore.wrist.flexADC;
}

void main(void) {
   SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);




   /***********************************/
   /** BEGIN - System Initialization **/
   /***********************************/
   SYSTEMConfig(SYS_CLK_FRQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

   mainBoardInit();




   sensorPkt.use.header._id = 0;
   RecvManager sensorRM;
   commonUARTNewRecvManager(&sensorRM, SENSOR_TX_MAIN_RX_PACKET_SIZE, 5000);

   MotorTxMainRxPacket motorPkt;
   motorPkt.use.header._id = 0;
   RecvManager motorRM;
   commonUARTNewRecvManager(&motorRM, MOTOR_TX_MAIN_RX_PACKET_SIZE, 5000);

   ExecTimer tenKExecTimer;
   commonTimerNewExecTimer(&tenKExecTimer, &tenKTimer, 10);

//    commonUARTInit(COMPUTER_UART, SYS_CLK_FRQ, BAUD_RATE);
```

```
  commonUARTInit(MAIN_SENSOR_UART, SYS_CLK_FRQ, BAUD_RATE);
  commonUARTInit(MOTOR_UART, SYS_CLK_FRQ, BAUD_RATE);
  commonTimerInit(TEN_K_TIMER, &tenKTimer, 1000, T2_PS_1_8);
  /*********************************/
  /** END - System Initialization **/
  /*********************************/



  /*************************************************/
  /** BEGIN - Startup Interboard Communication Test **/
  /*************************************************/
  // wait some time to ensure all PICs are running
  // before beginning test
  commonTimerDelay(&tenKTimer, 25000);
//   SENSOR_LED = LOW;
//   MOTOR_LED = LOW;
//   COMP_LED = LOW;
//   SENSOR_LED = LOW;
//   MOTOR_LED = LOW;
  PORTGCLR = (BIT_12 | BIT_14);
//   COMP_LED = LOW;
  //      _RD0 = 0; _RD1 = 0; _RD2 = 0;
  //      _RD0 = 0; _RD1 = 0; _RD2 = 0;


  sensorRecvCount = 0;
  motorRecvCount = 0;
  initiateSelfTest();

  unsigned char motorTestId = 0, sensorTestId = 0;
  BOOLEAN compTestFail = TRUE, motorTestFail = TRUE, sensorTestFail = TRUE;

  ExecTimer breakTimer;
```

```
commonTimerNewExecTimer(&breakTimer, &tenKTimer, 25000);


while (1) {

   if (commonTimerExecEvery(&breakTimer, &tenKTimer) /*|| (!compTestFail && !motorTestFail && !sensorTestFail)*/) {
      break;
   }

   /** Main Board <---> Sensor Board Connection Test **/
   if (sensorRecvFlag) {
      sensorRecvFlag = FALSE;

      unsigned char i;
      for (i = 0; i < sensorRecvCount; i++) {
         commonUARTRecvData(&sensorRM, &tenKTimer, &(sensorRecvBuf[i]), sensorPkt.stream);
      }

      sensorRecvCount = 0;
   }

   if (commonUARTTransferComplete(&sensorRM, &tenKTimer)) {
      if (sensorPkt.use.header._id == sensorTestId) {
         if (sensorPkt.use.header._id == 15) {
            sensorTestFail = FALSE;
         } else {
            sensorTestId++;
            MainTxSensorRxPacket p;
            p.use.header._id = sensorTestId;
            commonUARTSendData(MAIN_SENSOR_UART, p.stream, MAIN_TX_SENSOR_RX_PACKET_SIZE);
         }
      }
   }
```

```
   /** Main Board <---> Motor Board Connection Test **/
   if (motorRecvFlag) {
      motorRecvFlag = FALSE;

      unsigned char i;
      for (i = 0; i < motorRecvCount; i++) {
         commonUARTRecvData(&motorRM, &tenKTimer, &(motorRecvBuf[i]), motorPkt.stream);
      }

      motorRecvCount = 0;
   }

   if (commonUARTTransferComplete(&motorRM, &tenKTimer)) {
      if (motorPkt.use.header._id == motorTestId) {
         if (motorPkt.use.header._id == 15) {
            motorTestFail = FALSE;
         }

         motorTestId++;
         MainTxMotorRxPacket p;
         p.use.header._id = motorTestId;
         commonUARTSendData(MOTOR_UART, p.stream, MAIN_TX_MOTOR_RX_PACKET_SIZE);
      }
   }
}

ExecTimer exit;
commonTimerNewExecTimer(&exit, &tenKTimer, 20000);

// test has failed
while (1) {

   commonTimerDelay(&tenKTimer, 1000);
```

```
    //              if(compTestFail) {
    //                      COMP_LED = !COMP_LED;
    ////                             _RD0 = !_RD0;
    //              } else {
    //                      COMP_LED = HIGH;
    ////                             _RD0 = 1;
    //              }

    if (motorTestFail) {
       MOTOR_LED = !MOTOR_LED;
       //                      _RD1 = !_RD1;
    } else {
       MOTOR_LED = HIGH;
       //                      _RD1 = 1;
    }

    if (sensorTestFail) {
       SENSOR_LED = !SENSOR_LED;
       //                      _RD2 = !_RD2;
    } else {
       SENSOR_LED = HIGH;
       //                      _RD2 = 1;
    }

    // test
    if (commonTimerExecEvery(&exit, &tenKTimer)) {
       if(!motorTestFail && !sensorTestFail) {

       } else {

       }
//        PORTGCLR = (BIT_12 | BIT_14);
                        _RD0 = 0; _RD1 = 0; _RD2 = 0;
                        _RD0 = 0; _RD1 = 0; _RD2 = 0;
```

```
//          SENSOR_LED = LOW;
//          MOTOR_LED = LOW;
//          COMP_LED = LOW;
//          SENSOR_LED = LOW;
//          MOTOR_LED = LOW;
//          COMP_LED = LOW;
          break;
      }
    }
    /*************************************************/
    /** END - Startup Interboard Communication Test **/
    /*************************************************/


    /********************************/
    /** BEGIN - Main Execution Loop **/
    /********************************/

    /** Begin - Bowler Initialization **/
    Bowler_Init();
    char * dev = "NR Sample Serial Port";
    char * ser = "FF00FF00FF00";
    usb_CDC_Serial_Init(dev, ser, 0x04D8, 0x0001);
    BowlerPacket Packet; //Declare a packet
    addNamespaceToList((NAMESPACE_LIST *) getMainBoardNamespace());
    /** End - Bowler Initialization **/

//    ExoArmStoreBuf storeBuf[EXOARM_STORE_BUF_SIZE];
    commonUARTInit(MAIN_SENSOR_UART, SYS_CLK_FRQ, BAUD_RATE);
    commonUARTInit(MOTOR_UART, SYS_CLK_FRQ, BAUD_RATE);
    ExoArmStore exoarmStore;

    ExecTimer sanityExecTimer;
    commonTimerNewExecTimer(&sanityExecTimer, &tenKTimer, 10000);
```

```
   BOOLEAN recvAllSensors[7] = {0, 0, 0, 0, 0, 0, 0};
   sensorRecvCount = 0;
   motorRecvCount = 0;
   while (1) {


      if (commonTimerExecEvery(&sanityExecTimer, &tenKTimer)) {
//          MainTxSensorRxPacket p;
//          p.use.header._id = 1;
//          p.use.header._opt = SEN_OPT_SAMPLE_REQ;
//          p.use.header._col = 0;
//
//          commonUARTSendData(MAIN_SENSOR_UART, p.stream, MAIN_TX_SENSOR_RX_PACKET_SIZE);
         _RE1 = !_RE1;
      }

//       if(commanded) {
//          commanded = 0;
//
//          MainTxSensorRxPacket p;
//          p.use.header._id = commandedIndex;
//          p.use.header._opt = SEN_OPT_START;
//
//          commonUARTSendData(MAIN_SENSOR_UART, p.stream, MAIN_TX_SENSOR_RX_PACKET_SIZE);
//       }

      /** Main Board <---> Sensor Board **/
      if (sensorRecvFlag) {
         sensorRecvFlag = FALSE;

         unsigned char i;
         for (i = 0; i < sensorRecvCount; i++) {
            commonUARTRecvData(&sensorRM, &tenKTimer, &(sensorRecvBuf[i]), sensorPkt.stream);
```

```
        }

        sensorRecvCount = 0;
    }

    /** Main Board <---> Motor Board **/
    if (motorRecvFlag) {
        motorRecvFlag = FALSE;

        unsigned char i;
        for (i = 0; i < motorRecvCount; i++) {
            commonUARTRecvData(&motorRM, &tenKTimer, &(motorRecvBuf[i]), motorPkt.stream);
        }

        motorRecvCount = 0;
    }

    if (commonUARTTransferComplete(&sensorRM, &tenKTimer)) {

        _RG14 = !_RG14;

        // store received sensor data
        unsigned char id = sensorPkt.use.header._id;

        if (getMngrSensorSub(id)) {
            _RG12 = !_RG12;
            // just directly set the incoming data to the data store

            if (id == WRIST_ID) {
                setFlex(id, sensorPkt.use.flexAdc);
//                exoarmStore.wrist.flexADC = sensorPkt.use.flexAdc;

            } else {
                setFlex(id, sensorPkt.use.flexAdc);
```

```
            setPres(id, sensorPkt.use.presAdc);
//              exoarmStore.fingers[id].flexADC = sensorPkt.use.flexAdc;
//              exoarmStore.fingers[id].presADC = sensorPkt.use.presAdc;
          }
//
//          recvAllSensors[id]++;
        }
      }

      if (commonUARTTransferComplete(&motorRM, &tenKTimer)) {
        // store received motor data
        unsigned char id = motorPkt.use.header._id;

        if (mngr.motorSub[id]) {
          if (id == ELBOW_ID) {
            exoarmStore.elbow.encoderPos = motorPkt.use.encoderPos;
            exoarmStore.elbow.encoderVel = motorPkt.use.encoderVel;
            exoarmStore.elbow.currentADC = motorPkt.use.currentAdc;
            exoarmStore.elbow.seriesElasticADC = motorPkt.use.seriesElasticAdc;
          } else if (id == WRIST_ID) {
            exoarmStore.wrist.encoderPos = motorPkt.use.encoderPos;
            exoarmStore.wrist.encoderVel = motorPkt.use.encoderVel;
          } else {
            exoarmStore.fingers[id].encoderPos = motorPkt.use.encoderPos;
            exoarmStore.fingers[id].encoderVel = motorPkt.use.encoderVel;
            exoarmStore.fingers[id].currentADC = motorPkt.use.currentAdc;
            exoarmStore.fingers[id].seriesElasticADC = motorPkt.use.seriesElasticAdc;
          }

          recvAllSensors[id]++;
        }
      }
```

```
//      unsigned char i;
//      BOOLEAN allGood = TRUE;
//      for (i = 0; i < 7; i++) {
//         if (recvAllSensors[i] != 2 && (mngr.sensorSub[i] || mngr.motorSub[i])) {
//            allGood = FALSE;
//         }
//      }
//
//      if (allGood) {
//         addToStore(storeBuf, &tempStore);
//      }
      Bowler_Server(&Packet, FALSE);
   }
   /*******************************/
   /** END - Main Execution Loop **/
   /*******************************/
}

void mNewManager(Manager *m) {
   char i;
   for (i = 0; i < 7; i++) {
      m->sensorSub[i] = 0;
      m->motorCtrl[i] = 0;
      m->motorCtrl[i] = 0;
   }
}

void mStop(Manager *m, unsigned char id) {
   if (m->sensorSub[id]) {
      m->sensorSub[id] = 0;
      MainTxSensorRxPacket p;
      p.use.header._id = id;
      p.use.header._opt = SEN_OPT_STOP;
      p.use.header._col = SEN_LED_OFF;
```

```
      commonUARTSendData(MAIN_SENSOR_UART, p.stream, MAIN_TX_SENSOR_RX_PACKET_SIZE);
   }

   if (m->motorSub[id] || m->motorCtrl[id]) {
      m->motorSub[id] = 0;
      m->motorCtrl[id] = 0;
      MainTxMotorRxPacket p;
      p.use.header._id = id;
      commonUARTSendData(MOTOR_UART, p.stream, MAIN_TX_MOTOR_RX_PACKET_SIZE);
   }
}

//void mStart(Manager *m, unsigned char id) {
//         if(m->sensorSub[id]) {
//
//         }
//}

void mUpdate(Manager *m) {
   unsigned char i;
   for (i = 0; i < 7; i++) {
      if (m->sensorSub[i]) {
         MainTxSensorRxPacket p;
         p.use.header._id = i;
         //                              p.use.header._opt =
      }
   }
}

void addToStore(ExoArmStoreBuf storeBuf[EXOARM_STORE_BUF_SIZE], ExoArmStore *store) {
   if (storeBuf->firstIndex == 0) {
      storeBuf->store[EXOARM_STORE_BUF_SIZE - 1] = *store;
      storeBuf->firstIndex = EXOARM_STORE_BUF_SIZE - 1;
   } else {
```

```
        storeBuf->store[storeBuf->firstIndex - 1] = *store;
        storeBuf->firstIndex--;
    }
}

void mainBoardInit() {
    tenKTimer = 0;


    unsigned char i;
    for (i = 0; i < SENSOR_TX_MAIN_RX_PACKET_SIZE; i++) {
        sensorRecvBuf[i] = 0;
    }
    sensorRecvFlag = FALSE;
    sensorRecvCount = 0;



    for (i = 0; i < MOTOR_TX_MAIN_RX_PACKET_SIZE; i++) {
        motorRecvBuf[i] = 0;
    }
    motorRecvFlag = FALSE;
    motorRecvCount = 0;


    SENSOR_TRIS_LED = OUTPUT;
    MOTOR_TRIS_LED = OUTPUT;
    _TRISE1 = OUTPUT;
    _RE1 = LOW;
    SENSOR_LED = HIGH;
    MOTOR_LED = HIGH;
}

void initiateSelfTest() {
```

```
   MainTxSensorRxPacket sP;
   MainTxMotorRxPacket mP;
//    MainTxComputerRxPacket cP;

   sP.use.header._id = 0;
   mP.use.header._id = 0;
//    cP.use.header._id = 0;

   commonUARTSendData(MAIN_SENSOR_UART, sP.stream, MAIN_TX_SENSOR_RX_PACKET_SIZE);
   commonUARTSendData(MOTOR_UART, mP.stream, MAIN_TX_MOTOR_RX_PACKET_SIZE);
}

void __ISR(MAIN_SENSOR_UART_VECTOR, IPL2SOFT) IntUartSensorHandler(void) {
   // is this an Rx interrupt?
   if (INTGetFlag(INT_SOURCE_UART_RX(MAIN_SENSOR_UART))) {
      while (UARTReceivedDataIsAvailable(MAIN_SENSOR_UART)) {
         sensorRecvBuf[sensorRecvCount] = UARTGetDataByte(MAIN_SENSOR_UART);
         sensorRecvCount++;
      }
      sensorRecvFlag = TRUE;
      INTClearFlag(INT_SOURCE_UART_RX(MAIN_SENSOR_UART)); // clear RX interrupt flag
   }

   // We don't care about TX interrupt
   if (INTGetFlag(INT_SOURCE_UART_TX(MAIN_SENSOR_UART))) {
      INTClearFlag(INT_SOURCE_UART_TX(MAIN_SENSOR_UART)); // clear TX interrupt flag
//       _RG14 = 1;

   }

//    _RE1 = 1 ;
}

void __ISR(MOTOR_UART_VECTOR, IPL2SOFT) IntUartMotorHandler(void) {
```

```
  // is this an Rx interrupt?
  if (INTGetFlag(INT_SOURCE_UART_RX(MOTOR_UART))) {
    while (UARTReceivedDataIsAvailable(MOTOR_UART)) {
      motorRecvBuf[motorRecvCount] = UARTGetDataByte(MOTOR_UART); // get the received byte
      motorRecvCount++;
    }
    motorRecvFlag = TRUE; // set the received byte flag
    INTClearFlag(INT_SOURCE_UART_RX(MOTOR_UART)); // clear RX interrupt flag
  }

  // We don't care about TX interrupt
  if (INTGetFlag(INT_SOURCE_UART_TX(MOTOR_UART))) {
    INTClearFlag(INT_SOURCE_UART_TX(MOTOR_UART)); // clear TX interrupt flag
  }
}

void __ISR(TEN_K_TIMER_VECTOR, ipl2) tenKTimerHandler(void) {
  INTClearFlag(TEN_K_TIMER_CLEAR_FLAG);
  tenKTimer++;
}

/*******************/
/*- Bowler stuff  -*/
/*******************/

#define initLed()           _TRISD0 = OUTPUT;_TRISD1 = OUTPUT;_TRISD2 = OUTPUT;
#define initButton()        _TRISD6  = INPUT;_TRISD7  = INPUT;_TRISD13 = INPUT;CNPUESET=0x00098000;
#define isPressed()         ( _RD6==0 || _RD7==0 || _RD13==0)
#define setLed(a,b,c)       _RD0=a;_RD1=b;_RD2=c;

//These are the call backs. Fill these in to catch custom packets.

BYTE UserGetRPCs(BowlerPacket *Packet) {
  return 0;
```

```
}

BYTE UserPostRPCs(BowlerPacket *Packet) {
   return 0;
}

BYTE UserCriticalRPCs(BowlerPacket *Packet) {
   return 0;
}

BYTE SPITransceve(BYTE b) {
   SpiChnPutC(2, b); // send data on the master channel, SPI1
   Delay1us(10);
   return SpiChnGetC(2); // get the received data
}

BYTE get(BYTE b) {
   BYTE back = SPITransceve(b);
   //Delay10us(60);
   return back;
}
```

c. Communication Protocol Source File

```
#include "Bowler/Bowler.h"
#include "commonLib.h"
//#include "commonUART.h"
#include "commonStructs.h"

static const unsigned char cartNSName[]  = "exo.arm.*;0.3;;";
//volatile unsigned char buf;


BOOL sampleAsyncEventCallback(BowlerPacket * Packet,BOOL (*pidAsyncCallbackPtr)(BowlerPacket *Packet)){

   // This gets called by the stack when the system is serviced. If you need to use async,
   // fill the provided packet and push it
   return FALSE;
}

BOOL setSensor(BowlerPacket *p) {
   unsigned short int id = get16bit(p, 0);
   unsigned short int s = get16bit(p, 2);

   setMngrSensorSub(id, s);

   set16bit(p, getMngrSensorSub(id), 0);
}

BOOL getSensor(BowlerPacket *p) {
   set16bit(p, getFlex(THUMB_ID), 0);
   set16bit(p, getPres(THUMB_ID), 2);

   set16bit(p, getFlex(INDEX_ID), 4);
   set16bit(p, getPres(INDEX_ID), 6);
```

```
   set16bit(p, getFlex(MIDDLE_ID), 8);
   set16bit(p, getPres(MIDDLE_ID), 10);

   set16bit(p, getFlex(RING_ID), 12);
   set16bit(p, getPres(RING_ID), 14);

   set16bit(p, getFlex(LITTLE_ID), 16);
   set16bit(p, getPres(LITTLE_ID), 18);

   set16bit(p, getWristFlex(), 20);
   p->use.head.DataLegnth = 4 + (5*(2 + 2)) + 2;

   MainTxSensorRxPacket pt;
   pt.use.header._id = 1;
   pt.use.header._opt = SEN_OPT_SAMPLE_REQ;
   pt.use.header._col = 0;

   commonUARTSendData(UART2, pt.stream, MAIN_TX_SENSOR_RX_PACKET_SIZE);
}

BOOL setMotorSub(BowlerPacket *p) {
   unsigned short int id = get16bit(p, 0);
//    unsigned short int
}

BOOL setMotorControl(BowlerPacket *p) {

}

BOOL getMotor(BowlerPacket *p) {

}

static RPC_LIST setSensorList={   BOWLER_POST,// Method
```

```
                "SenS",//RPC as string, 4 bytes ONLY
                &setSensor,//function pointer to a packet parsinf function
                ((const char [3]){      BOWLER_I16,// from java to device
                                BOWLER_I16,//
                                        0}),// NULL TERMINATOR
                BOWLER_POST,// response method
                ((const char [2]){    BOWLER_I16,
                                0}),//
                NULL //Termination
};

static RPC_LIST getSensorList={  BOWLER_POST,// Method
                "SenG",//RPC as string, 4 bytes ONLY
                &getSensor,//function pointer to a packet parsinf function
                ((const char [1]){0}),// NULL TERMINATOR
                BOWLER_POST,// response method
                ((const char [12]){     BOWLER_I16,// from device to java
                                BOWLER_I16,
                                BOWLER_I16,
                                BOWLER_I16,
                                BOWLER_I16,
                                BOWLER_I16,
                                BOWLER_I16,
                                BOWLER_I16,
                                BOWLER_I16,
                                BOWLER_I16,
                                BOWLER_I16,
                                        0}),//
                NULL //Termination
};

// ONLY ONE OF THESE: static NAMESPACE_LIST bcsSample
static NAMESPACE_LIST bcsSample ={    cartNSName,// The string defining the namespace
                NULL,// the first element in the RPC list
```

```
                    &sampleAsyncEventCallback,// async for this namespace
                    NULL// no initial elements to the other namesapce field.
};

static BOOL namespcaedAdded = FALSE;
NAMESPACE_LIST * getMainBoardNamespace(){
        if(!namespcaedAdded){
         //POST
         addRpcToNamespace(&bcsSample, &setSensorList);
         addRpcToNamespace(&bcsSample, &getSensorList);
         namespcaedAdded =TRUE;
        }

        return &bcsSample;//Return pointer to the struct
}
```

H. Sensor Board Software

a. Main Header File

```
#ifndef SENSORBOARRD_H
#define          SENSORBOARRD_H

#ifdef   __cplusplus
extern "C" {
#endif

/************************/
/*- Pragma Directives -*/
/************************/
#ifdef USB_A0_SILICON_WORK_AROUND
        #pragma config UPLLEN   = OFF      // USB PLL Enabled (A0 bit inverted)
#else
        #pragma config UPLLEN   = ON       // USB PLL Enabled
#endif

#pragma config FPLLMUL  = MUL_20       // PLL Multiplier
#pragma config UPLLIDIV = DIV_2        // USB PLL Input Divider
#pragma config FPLLIDIV = DIV_2        // PLL Input Divider
#pragma config FPLLODIV = DIV_1        // PLL Output Divider
#pragma config FPBDIV   = DIV_1        // Peripheral Clock divisor
#pragma config FWDTEN   = OFF          // Watchdog Timer
#pragma config WDTPS    = PS1          // Watchdog Timer Postscale
#pragma config FCKSM    = CSDCMD       // Clock Switching & Fail Safe Clock Monitor
#pragma config OSCIOFNC = OFF          // CLKO Enable
#pragma config POSCMOD  = HS           // Primary Oscillator
```

```
#pragma config IESO     = ON          // Internal/External Switch-over
#pragma config FSOSCEN  = ON          // Secondary Oscillator Enable (KLO was off)
#pragma config FNOSC    = PRIPLL      // Oscillator Selection
#pragma config CP       = OFF         // Code Protect
#pragma config BWP      = OFF         // Boot Flash Write Protect
#pragma config PWP      = OFF         // Program Flash Write Protect
#pragma config ICESEL   = ICS_PGx2    // ICE/ICD Comm Channel Select
#pragma config DEBUG    = OFF         // Background Debugger Enable

#include "commonLib.h"
#include "commonADC.h"
#include "sensorBoardAcquire.h"
//#include "commonSPI.h"
#include "commonTimer.h"
#include "commonUART.h"
#include "LED.h"


#define MAIN_UART                       UART3
#define MAIN_UART_VECTOR                _UART_3_VECTOR

//#define MAIN_UART                             UART2
//#define MAIN_UART_VECTOR              _UART_2_VECTOR

#define TEN_K_TIMER                             2
#define TEN_K_TIMER_VECTOR              _TIMER_2_VECTOR
#define TEN_K_TIMER_CLEAR_FLAG INT_T2

#define SYS_CLK_FREQ                    80000000ul
#define BAUD_RATE                       9600
```

```
        void initSensorBoard();

/***************/
/*- Variables -*/
/***************/
        volatile unsigned long tenKTimer;

   volatile unsigned char recvBuf[MAIN_TX_SENSOR_RX_PACKET_SIZE];
   volatile BOOLEAN recvFlag;
        volatile unsigned char recvCount;

        unsigned short int sampleCount;

#ifdef  __cplusplus
}
#endif

#endif  /* SENSORBOARRD_H */
```

b. Main Source File

```
#include "sensorBoard.h"


void main() {
//        _RG14 = 1;
        /***********************************/
        /** BEGIN - System Initialization **/
        /***********************************/
        SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

        initSensorBoard();

        MainTxSensorRxPacket mainBrdPkt;

        SensorTxMainRxPacket sendPkts[SENSOR_GROUPS];

        RecvManager mainRecvMngr;
        commonUARTNewRecvManager(&mainRecvMngr, MAIN_TX_SENSOR_RX_PACKET_SIZE, 5000);

        ExecTimer tenKExecTimer;
        commonTimerNewExecTimer(&tenKExecTimer, &tenKTimer, 10);

        SensorBoardStore store[SAMPLES];

        Manager pubMngr;
        mNewManager(&pubMngr);



        commonUARTInit(MAIN_UART, SYS_CLK_FREQ, BAUD_RATE);
```

```
//      commonUARTInit(COMP_UART, SYS_CLK_FREQ, BAUD_RATE);
        commonTimerInit(TEN_K_TIMER, &tenKTimer, 1000, T2_PS_1_8);

        commonADCSetupSensorBoard(5, 5);

        setupSPI_LED();
        /*********************************/
        /** END - System Initialization **/
        /*********************************/


        /************************************************/
        /** BEGIN - Startup Interboard Communication Test **/
        /************************************************/
//      _TRISD1 = 0; _RD1 = 1;
//      _TRISD0 = 0; _RD0 = 1;
        recvCount = 0;
        while(1) {
                if(recvFlag) {
                        recvFlag = FALSE;

                        unsigned char i;
                        for(i = 0; i < recvCount; i++) {
                                commonUARTRecvData(&mainRecvMngr, &tenKTimer, &(recvBuf[i]), mainBrdPkt.stream);
                        }

                        recvCount = 0;
                }

                if(commonUARTTransferComplete(&mainRecvMngr, &tenKTimer)) {
                        SensorTxMainRxPacket p;
```

```
                        unsigned char id = mainBrdPkt.use.header._id;
                        p.use.header._id = id;
                        p.use.header._aux = 0;
                        p.use.flexAdc = 0;
                        p.use.presAdc = 0;

//                      _RD1 = !_RD1;
                        commonUARTSendData(MAIN_UART, p.stream, SENSOR_TX_MAIN_RX_PACKET_SIZE);

                        if(id == 15) {
                                break;
                        }
                }
        }
        /*************************************************/
        /** END - Startup Interboard Communication Test **/
        /*************************************************/



        /*********************************/
        /** BEGIN - Main Execution Loop **/
        /*********************************/
        recvCount = 0;
        while(1) {

//              if(commonTimerExecEvery(&tenKExecTimer, &tenKTimer)) {
                        unsigned char i;
                        for(i = 0; i < SENSOR_GROUPS-1; i++) {
                                if(sendPkts[i].use.presAdc < 500) {
                                        setLEDColor(i, 0);
```

```
                        } else if(sendPkts[i].use.presAdc < 1000) {
                                setLEDColor(i, 1);
                        } else if(sendPkts[i].use.presAdc >= 1000) {
                                setLEDColor(i, 2);
                        }
//                      setLEDColor(i, 1);
//                      commonTimerDelay(&tenKExecTimer, 1);
                        int r = 0;
                        while(r<1000){
                                r++;
                        }
                }
//      }


        if(recvFlag) {
                recvFlag = FALSE;

                unsigned char i;
                for(i = 0; i < recvCount; i++) {
                        commonUARTRecvData(&mainRecvMngr, &tenKTimer, &(recvBuf[i]), mainBrdPkt.stream);
                }

                recvCount = 0;
        }

        if(commonUARTTransferComplete(&mainRecvMngr, &tenKTimer)) {
                unsigned char id = mainBrdPkt.use.header._id;
                unsigned char opt = mainBrdPkt.use.header._opt;
                unsigned char col = mainBrdPkt.use.header._col;
```

```
//                      commonUARTSendByte(MAIN_UART, mainBrdPkt.stream[0]);


                        if(opt == SEN_OPT_SAMPLE_REQ) {
                                readSensors(store);
                                filterSensors(store, sendPkts);

                                mSend(&pubMngr, MAIN_UART, sendPkts);




//                              sampleCount++;

//                              if(sampleCount >= 10) {
//                                      pmPublish(&pubMngr, UART1);
//                              }


                        } else if(id >= 0 && id <= WRIST_ID) {
                                if(opt == SEN_OPT_START) {
                                pubMngr.sensors[id] = 1;
//                                      pmStartPublish(&pubMngr, id);
                                } else if(opt == SEN_OPT_STOP) {
                                        pubMngr.sensors[id] = 0;
//                                      pmStopPublish(&pubMngr, id);
                                }
                        }

//                      setLEDColor(id, col);
                }
```

```
        }
        /*********************************/
        /** END - Main Execution Loop **/
        /*********************************/
}

void initSensorBoard() {

        int i;
        for(i = 0; i <  MAIN_TX_SENSOR_RX_PACKET_SIZE; i++) {
                recvBuf[i] = 0;
        }
        recvFlag = FALSE;
        recvCount = 0;

        sampleCount = 0;

//      readSensorsFlag = FALSE;
//      readSensorChangeCount = 0;

//      _TRISG14 = 0;
//      _RG14;

//      _TRISD2 = 0;
//      _TRISD1 = 0;
//      _RD1 = 0;

        // setup pin change interrupts

//      PORTSetPinsDigitalIn(IOPORT_D, BIT_7);
//      mCNOpen(CN_ON, CN16_ENABLE, CN16_PULLUP_ENABLE);
```

```
//        pinChangeDummy = mPORTDRead();
//        ConfigIntCN(CHANGE_INT_ON | CHANGE_INT_PRI_2);
}

// this interrupt is used to send/receive data to/from the main board
// UART 2 interrupt handler
// it is set at priority level 2 with software context saving
void __ISR(MAIN_UART_VECTOR, IPL2SOFT) IntUartHandler(void) {
   // is this an Rx interrupt?
   if(INTGetFlag(INT_SOURCE_UART_RX(MAIN_UART))) {
      while(UARTReceivedDataIsAvailable(MAIN_UART)) {
         recvBuf[recvCount] = UARTGetDataByte(MAIN_UART); // get the received byte
                              recvCount++;
      }
                  recvFlag = TRUE; // set the received byte flag
      INTClearFlag(INT_SOURCE_UART_RX(MAIN_UART)); // clear RX interrupt flag
   }

   // We don't care about TX interrupt
   if (INTGetFlag(INT_SOURCE_UART_TX(MAIN_UART))) {
      INTClearFlag(INT_SOURCE_UART_TX(MAIN_UART)); // clear TX interrupt flag
   }
}

void __ISR(TEN_K_TIMER_VECTOR, ipl2) Timer2Handler(void) {
   // clear the interrupt flag
   INTClearFlag(TEN_K_TIMER_CLEAR_FLAG);
   tenKTimer++;
}
```

c. Acquire Header File

```
#ifndef ACQUIRE_H
#define          ACQUIRE_H

#ifdef   __cplusplus
extern "C" {
#endif

#include "commonStructs.h"

#define SAMPLES 100
#define SENSORS 11


#define M_STOP

        typedef struct _Manager {
                unsigned char sensors[SENSOR_GROUPS];
        } Manager;

   typedef struct _FingerStore {
      unsigned short int flexAdc;
      unsigned short int presAdc;
   } FingerStore;

   typedef struct _SensorBoardStore {
      FingerStore thumb;
      FingerStore index;
      FingerStore middle;
      FingerStore ring;
      FingerStore little;
```

```
    unsigned short int wristFlexAdc;
  } SensorBoardStore;


        void readSensors(SensorBoardStore store[SAMPLES]);

        void filterSensors(SensorBoardStore store[SAMPLES], SensorTxMainRxPacket *result);



#ifdef  __cplusplus
}
#endif

#endif  /* ACQUIRE_H */
```

d. Acquire Source File

```
#include "sensorBoardAcquire.h"

void readSensors(SensorBoardStore store[SAMPLES]) {
   // samples data from all the sensors multiple times
   // pointer represents an array of SensorBoardStore
   // use #define THUMB_ID, INDEX_ID, etc for array element locations
   // in SensorBoardStore

        unsigned short int dataFlex[8];
        unsigned short int dataPres[8];
        unsigned char i;
        for(i = 0; i < SAMPLES; i++) {
                commonADCReadAllFlex(4, dataFlex);
                commonADCReadAllPres(4, dataPres);

                store[i].thumb.flexAdc = dataFlex[0];
                store[i].thumb.presAdc = dataPres[4];

                store[i].index.flexAdc = dataFlex[1];
                store[i].index.presAdc = dataPres[5];

                store[i].middle.flexAdc = dataFlex[2];
                store[i].middle.presAdc = dataPres[0];

                store[i].ring.flexAdc = dataFlex[3];
                store[i].ring.presAdc = dataPres[1];

                store[i].little.flexAdc = dataFlex[4];
                store[i].little.presAdc = dataPres[2];
```

```
                store[i].wristFlexAdc = dataFlex[5];
        }
}

void filterSensors(SensorBoardStore store[SAMPLES], SensorTxMainRxPacket *result) { // sensorPublishManager
  // store is an array
  // result is not an array

  unsigned long sum[SENSORS];

  unsigned short int i;
        for(i = 0; i < SENSORS; i++) {
                sum[i] = 0;
        }

  for(i = 0; i < SAMPLES; i++) {
    sum[THUMB_ID*2] += (unsigned long)(store[i].thumb.flexAdc);
    sum[THUMB_ID*2+1] += (unsigned long)(store[i].thumb.presAdc);

    sum[INDEX_ID*2] += (unsigned long)(store[i].index.flexAdc);
    sum[INDEX_ID*2+1] += (unsigned long)(store[i].index.presAdc);

    sum[MIDDLE_ID*2] += (unsigned long)(store[i].middle.flexAdc);
    sum[MIDDLE_ID*2+1] += (unsigned long)(store[i].middle.presAdc);

    sum[RING_ID*2] += (unsigned long)(store[i].ring.flexAdc);
    sum[RING_ID*2+1] += (unsigned long)(store[i].ring.presAdc);

    sum[LITTLE_ID*2] += (unsigned long)(store[i].little.flexAdc);
    sum[LITTLE_ID*2+1] += (unsigned long)(store[i].little.presAdc);
```

```
        sum[WRIST_ID*2] += (unsigned long)(store[i].wristFlexAdc);
    }

        for(i = 0; i < SENSOR_GROUPS; i++) {
            if(i != WRIST_ID) {
                result[i].use.header._id = i;
                result[i].use.flexAdc = (unsigned short int)(sum[i*2]/((unsigned long)SAMPLES));
                result[i].use.presAdc = (unsigned short int) (sum[i*2+1]/((unsigned long)SAMPLES));
            } else {
                result[i].use.header._id = i;
                result[i].use.flexAdc = (unsigned short int)(sum[i*2]/((unsigned long)SAMPLES));
            }
        }
}

void mSend(Manager *m, unsigned char u, SensorTxMainRxPacket *p) {
        unsigned char i;
        for(i = 0; i < SENSOR_GROUPS; i++) {
            if(m->sensors[i]) {
                p[i].use.header._aux = 0;
                commonUARTSendData(u, p[i].stream, SENSOR_TX_MAIN_RX_PACKET_SIZE);
            }
        }
}

void mNewManager(Manager *m) {
        char i;
        for(i = 0;i < SENSOR_GROUPS; i++) {
            m->sensors[i] = 0;
        }
}
```

e. LED Header File

```
#ifndef LED_H
#define          LED_H

#ifdef   __cplusplus
extern "C" {
#endif

#include "commonLib.h"

#define sendByte1  0b10000100
#define sendByte2  0b00100001
#define sendByte3  0b00001000
#define sendByte4  0b01000010
#define sendByte5  0b00010000
#define sendByte6  0b00001110
#define sendByte7  0b10000000

#define LED_SPI                  (SPI_CHANNEL3)

void setupSPI_LED();
void setColorRed();
void setColorGreen();
void setColorBlue();
void setColorYellow();
void setLedOff();
void setLEDColor(unsigned char id, unsigned char color); // takes in 0-4 for 5 fingers id. 0-2 for color id.

#ifdef   __cplusplus
}
#endif
```

```
#endif  /* LED_H */
```

f. LED Source File

```
#include "LED.h"


void setupSPI_LED(){
          SpiOpenFlags oFlags = SPI_OPEN_MODE8 | SPI_OPEN_MSTEN;
          SpiChnOpen(LED_SPI, oFlags, 20);
}

void setLEDColor(unsigned char id, unsigned char color) {

//  SpiChnPutC(LED_SPI, sendByte7);
   switch(id){

      case(0b100) : {switch(color){
         case(0b00): setColorRed();
//                setLedOff();
//                setLedOff();
//                setLedOff();
               break;
         case(0b01): setColorGreen();
//                setLedOff();
//                setLedOff();
//                setLedOff();
               break;
         case(0b10): setColorBlue();
//                setLedOff();
//                setLedOff();
//                setLedOff();
               break;
      }}break;
```

```
      case(0b11) : {switch(color){
        case(0b00): setLedOff();
              setColorRed();
//              setLedOff();
//              setLedOff();
              break;
        case(0b01): setLedOff();
              setColorGreen();
//              setLedOff();
//              setLedOff();
              break;
        case(0b10): setLedOff();
              setColorBlue();
//              setLedOff();
//              setLedOff();
              break;
      }}break;
      case(0b10) : {switch(color){
        case(0b00): setLedOff();
              setLedOff();
              setColorRed();
//              setLedOff();
              break;
        case(0b01): setLedOff();
              setLedOff();
              setColorGreen();
//              setLedOff();
              break;
        case(0b10): setLedOff();
              setLedOff();
              setColorBlue();
```

```
//                setLedOff();
                  break;
        }}break;
        case(0b01) : {switch(color){
           case(0b00): setLedOff();
                  setLedOff();
                  setLedOff();
                  setColorRed();
                  break;
           case(0b01): setLedOff();
                  setLedOff();
                  setLedOff();
                  setColorGreen();
                  break;
           case(0b10): setLedOff();
                  setLedOff();
                  setLedOff();
                  setColorBlue();
                  break;
        }}break;
        case(0b00) : {switch(color){
           case(0): setLedOff();
                  setColorRed();
                  setLedOff();
                  setLedOff();
                  break;
           case(1): setLedOff();
                  setColorGreen();
                  setLedOff();
                  setLedOff();
                  break;
```

```
        case(2): setLedOff();
                 setColorBlue();
                 setLedOff();
                 setLedOff();
                 break;
      }}break;
   }

}


// sends data to turn off a color on LED
void sendOffByte(){
   SpiChnPutC(LED_SPI, sendByte1);
   SpiChnPutC(LED_SPI, sendByte2);
   SpiChnPutC(LED_SPI, sendByte3);
   SpiChnPutC(LED_SPI, sendByte4);
   SpiChnPutC(LED_SPI, sendByte5);

}

// sends data to turn on a color LED
void sendOnByte(){

   SpiChnPutC(LED_SPI, sendByte1);
   SpiChnPutC(LED_SPI, sendByte2);
   SpiChnPutC(LED_SPI, sendByte6);
   SpiChnPutC(LED_SPI, sendByte4);
   SpiChnPutC(LED_SPI, sendByte5);

}
```

```
void sendOByte(){

   SpiChnPutC(LED_SPI, sendByte7);
   SpiChnPutC(LED_SPI, sendByte7);
   SpiChnPutC(LED_SPI, sendByte7);
   SpiChnPutC(LED_SPI, sendByte7);
   SpiChnPutC(LED_SPI, sendByte7);

}

void setColorRed(){
    sendOffByte();  //green if on
    sendOnByte();   //red if on
    sendOffByte();  //blue if on
}

void setColorGreen(){
    sendOnByte();  //green if on
    sendOffByte();   //red if on
    sendOffByte();  //blue if on
}

void setColorBlue(){
    sendOffByte();  //green if on
    sendOffByte();   //red if on
    sendOnByte();  //blue if on
}

void setColorYellow(){
    sendOByte();  //green if on
    sendOByte();   //red if on
```

```
    sendOByte();  //blue if on
}

void setLedOff(){
      sendOffByte();  //green if on
      sendOffByte();   //red if on
      sendOffByte();  //blue if on
}
```

I. Motor Board Software

a. Main Header File

```
#ifndef MOTORBOARD_H
#define MOTORBOARD_H

#ifdef  __cplusplus
extern "C" {
#endif

/************************/
/*- Pragma Directives -*/
/************************/
#ifdef USB_A0_SILICON_WORK_AROUND
        #pragma config UPLLEN   = OFF      // USB PLL Enabled (A0 bit inverted)
#else
        #pragma config UPLLEN   = ON       // USB PLL Enabled
#endif

#pragma config FPLLMUL  = MUL_20       // PLL Multiplier
#pragma config UPLLIDIV = DIV_2        // USB PLL Input Divider
#pragma config FPLLIDIV = DIV_2        // PLL Input Divider
#pragma config FPLLODIV = DIV_1        // PLL Output Divider
#pragma config FPBDIV   = DIV_1        // Peripheral Clock divisor
#pragma config FWDTEN   = OFF          // Watchdog Timer
#pragma config WDTPS    = PS1          // Watchdog Timer Postscale
#pragma config FCKSM    = CSDCMD       // Clock Switching & Fail Safe Clock Monitor
#pragma config OSCIOFNC = OFF          // CLKO Enable
#pragma config POSCMOD  = HS           // Primary Oscillator
#pragma config IESO     = ON           // Internal/External Switch-over
#pragma config FSOSCEN  = ON           // Secondary Oscillator Enable (KLO was off)
#pragma config FNOSC    = PRIPLL       // Oscillator Selection
#pragma config CP       = OFF          // Code Protect
```

```
#pragma config BWP      = OFF          // Boot Flash Write Protect
#pragma config PWP      = OFF          // Program Flash Write Protect
#pragma config ICESEL   = ICS_PGx2     // ICE/ICD Comm Channel Select
#pragma config DEBUG    = OFF          // Background Debugger Enable



/************************/
/*- Include Directives -*/
/************************/
#include <stdio.h>
#include <stdlib.h>
//#include <stdint.h>
#include "commonLib.h"
#include "commonUART.h"
#include "commonTimer.h"
#include "controlManager.h"
#include "pwmGeneratorDriver.h"
#include "publishManager.h"
#include "decoder.h"
#include "digiPotDriver.h"

/************************/
/*- Macro Definitions -*/
/************************/
#define initLed()           _TRISD0 = 0;_TRISD1 = 0;_TRISD2 = 0;
#define setLed(a,b,c)     _RD0=a;_RD1=b;_RD2=c;
#define initMainLed()   _TRISG13 = 0;

#define MAIN_UART                       UART1
#define MAIN_UART_VECTOR    _UART_1_VECTOR

#define TEN_K_TIMER                             1
#define TEN_K_TIMER_VECTOR          _TIMER_1_VECTOR
```

```
#define TEN_K_TIMER_CLEAR_FLAG      INT_T1


/*************************/
/*- Structures & Unions -*/
/*************************/



/*************************/
/*- Function Prototypes -*/
/*************************/
   void motorBoardInit();


/***************/
/*- Variables -*/
/***************/
   volatile unsigned char recvBuf[MAIN_TX_MOTOR_RX_PACKET_SIZE];
   volatile BOOL recvFlag;
         volatile unsigned char recvCount;

         volatile unsigned long tenKTimer;


#ifdef   __cplusplus
}
#endif

#endif   /* MOTORBOARD_H */
```

b. Main Source File

```
#include "motorBoard.h"


void main() {
        /***********************************/
        /** BEGIN - System Initialization **/
        /***********************************/
        SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

        motorBoardInit();

  initMainLed();
  initLed();

  ControlInstanceManager cm;
  cmNewControlManager(&cm);

  MotorPublishManager pm;
        pmNewManager(&pm);

  RecvManager mainRecvMngr;
  commonUARTNewRecvManager(&mainRecvMngr, MAIN_TX_MOTOR_RX_PACKET_SIZE, 5000);

  MainTxMotorRxPacket mainBrdPkt;

  ExecTimer tenKExecTimer;
  commonTimerNewExecTimer(&tenKExecTimer, &tenKTimer, 10); // executes at 1,000 Hz

  commonUARTInit(MAIN_UART, 80000000ul, 9600);
  commonTimerInit(TEN_K_TIMER, &tenKTimer, 1000, T2_PS_1_8);
//   _RG13 = 1;
```

```
	/*********************************/
	/** END - System Initialization **/
	/*********************************/


	/*************************************************/
	/** BEGIN - Startup Interboard Communication Test **/
	/*************************************************/
	recvCount = 0;
	while(1) {
		if(recvFlag) {
			recvFlag = FALSE;

			_RD0 = !_RD0;

			unsigned char i;
			for(i = 0; i < recvCount; i++) {
				commonUARTRecvData(&mainRecvMngr, &tenKTimer, &(recvBuf[i]), mainBrdPkt.stream);
			}

			recvCount = 0;
		}

		if(commonUARTTransferComplete(&mainRecvMngr, &tenKTimer)) {
			MotorTxMainRxPacket p;
			unsigned char id = mainBrdPkt.use.header._id;
			p.use.header._id = id;
//			p.use.header._opt = 0;
//			p.use.header._aux = 0;
//			p.use.encoderPos = 0;
//			p.use.encoderVel = 0;
//			p.use.currentAdc = 0;
//			p.use.seriesElasticAdc = 0;
```

```
//                          _RD1 = !_RD1;
                            commonUARTSendData(MAIN_UART, p.stream, MOTOR_TX_MAIN_RX_PACKET_SIZE);

                            if(id == 15) {
                                    break;
                            }
                    }
            }
            /*************************************************/
            /** END - Startup Interboard Communication Test **/
            /*************************************************/

//          AD1PCFGSET = 0x0040;
//          AD1PCFGSET = 0x0080;
//          _TRISB7 = OUTPUT;
//          _TRISE8 = OUTPUT;
//
//          _RB7 = HIGH;
//          _RE8 = HIGH;
////        initDigiPots();
////        setDigiPot0(128);
////        setDigiPot1(128);
//
//          initPwmGeneratorDriver();
//
////        int i = 0, j = 0;
//          while(1) {
////                  i++;
////                  j++;
////                  if(i > 1000) {
////                          i = 0;
////                  }
////                  if(j > 255) {
```

```
////                        j = 0;
////                }
////                setDigiPot0(j);
////                setDigiPot1(j);
//                  setAllGS(10);
////                setGS(0, 50);
//                  updateGS();
//
//        }


        /*********************************/
        /** BEGIN - Main Execution Loop **/
        /*********************************/
        recvCount = 0;
   while(1) {
//                  updateGS();

      if(commonTimerExecEvery(&tenKExecTimer, &tenKTimer)) {

      }

      if(recvFlag) {
         recvFlag = FALSE;

                              unsigned char i;
                              for(i = 0; i < recvCount; i++) {
                                       commonUARTRecvData(&mainRecvMngr, &tenKTimer, &(recvBuf[i]), mainBrdPkt.stream);
                              }

                              recvCount = 0;
      }

      // received a packet from the main board
```

```
    if(commonUARTTransferComplete(&mainRecvMngr, &tenKTimer)) {
                        unsigned char id = mainBrdPkt.use.header._id;
                        unsigned char opt = mainBrdPkt.use.header._opt;
                        unsigned char ctrl = mainBrdPkt.use.header._controlType;

                        if(id == 15) { // execute update
                                cmStartInstance(&cm, &mainBrdPkt);
                        } else { // new publish/command requests
                                if(opt & MTR_OPT_START) {
                                        if(opt & MTR_OPT_PUB) {
                                                pmStartPublish(&pm,    id);
                                        }
                                        if(opt & MTR_OPT_CTRL) {
                                                cmStartInstance(&cm, &mainBrdPkt);
                                        }
                                } else {
                                        if(opt & MTR_OPT_PUB) {
                                                pmStopPublish(&pm, id);
                                        }
                                        if(opt & MTR_OPT_CTRL) {
                                                cmStopInstance(&cm, &mainBrdPkt);
                                        }
                                }
                        }
    }
  }
        /*******************************/
        /** END - Main Execution Loop **/
        /*******************************/
}

void motorBoardInit() {
        DDPCONbits.TROEN = 0;
        DDPCONbits.JTAGEN = 0;
```

```
        int i;
        for(i = 0; i <  MAIN_TX_MOTOR_RX_PACKET_SIZE; i++) {
                recvBuf[i] = 0;
        }
   recvFlag = FALSE;
        recvCount = 0;

        tenKTimer = 0;
}

// this interrupt is used to send/receive data to/from the main board
// UART 2 interrupt handler
// it is set at priority level 2 with software context saving
void __ISR(MAIN_UART_VECTOR, IPL2SOFT) IntUartHandler(void) {
   // is this an Rx interrupt?
   if(INTGetFlag(INT_SOURCE_UART_RX(MAIN_UART))) {
      while(UARTReceivedDataIsAvailable(MAIN_UART)) {
         recvBuf[recvCount] = UARTGetDataByte(MAIN_UART); // get the received byte
                        recvCount++;
      }
                recvFlag = TRUE; // set the received byte flag
      INTClearFlag(INT_SOURCE_UART_RX(MAIN_UART)); // clear RX interrupt flag
   }

   // We don't care about TX interrupt
   if (INTGetFlag(INT_SOURCE_UART_TX(MAIN_UART))) {
      INTClearFlag(INT_SOURCE_UART_TX(MAIN_UART)); // clear TX interrupt flag
   }
}

void __ISR(TEN_K_TIMER_VECTOR, ipl2) Timer2Handler(void) {
   // clear the interrupt flag
   INTClearFlag(TEN_K_TIMER_CLEAR_FLAG);
```

```
    tenKTimer++;
}
```

c. PWM Generator Header File

```
/**
* @file pwmGeneratorDriver.h
* @author        Jason Klein (jason.e.klein@wpi.edu)
*
* Driver to control the TLC5940. This implementation uses I/O built-in
* features on the PIC32MX795F512L. When using this driver, make sure the
* following built-in features are not being used by other parts of your
* program. If they are being used, check to see if there are conflicts with
* Timer, Output Compare, and SPI setup arguments are the same, or if not,
* change this driver to work. By default, the following are the built-in features used
* by this driver.
*  Timer 3
*  Timer 4
*  Output Compare 3
*  SPI 1
*
* Driver created using refrence programming flow chart provided by Texas
* Instruments:
* http://www.ti.com/lit/sw/slvc106/slvc106.pdf
*
* @bug                    It works but only if the values assigned are between 0 and ~200
*/

#ifndef PWMGENERATORDRIVER_H
#define PWMGENERATORDRIVER_H

#ifdef    __cplusplus
extern "C" {
#endif

/**********************/
/*- Pragma Directives -*/
```

```
/***********************/



/***********************/
/*- Include Directives -*/
/***********************/
#include "commonLib.h"



/***********************/
/*- Macro Definitions -*/
/***********************/
////////////////// DO NOT MODIFY
#define DC_DATA_SIZE    12
#define GS_DATA_SIZE    24

#define BOOLEAN       unsigned char
#define TRUE          1
#define FALSE         0

////////////////// AS NEEDED, MODIFY ONLY THESE #define SETTINGS BELOW
//#define TLC5940_SPI_CHANNEL (SPI_CHANNEL4)
//
//#define GSCLK_TRIS        (_TRISD3)
//#define DCPRG_TRIS        (_TRISA2)
//#define VPRG_TRIS         (_TRISB12)
//#define XLAT_TRIS         (_TRISA9)
//#define BLANK_TRIS        (_TRISA1)
//
//#define GSCLK_PIN         (_RD3)
//#define DCPRG_PIN         (_RA2)
//#define VPRG_PIN          (_RB12)
```

```
//#define XLAT_PIN          (_RA9)
//#define BLANK_PIN         (_RA1)

          // PIC32 ETHERNET STARTER KIT
#define TLC5940_SPI_CHANNEL (SPI_CHANNEL1)

#define GSCLK_TRIS        (_TRISD3)
#define DCPRG_TRIS        (_TRISA2)
#define VPRG_TRIS         (_TRISB12)
#define XLAT_TRIS         (_TRISC2)
#define BLANK_TRIS        (_TRISE5)

#define BLANK_PIN         (_RE5)
#define BLANK_HIGH                  (mPORTESetBits(BIT_5))
#define BLANK_LOW                   (mPORTESetBits(BIT_5))
#define BLANK_TOGGLE                (mPORTEToggleBits(BIT_5))

//#define GSCLK_TOGGLE              (mPORTDToggleBits(3))
//#define GSCLK_PIN                 (3)

#define GSCLK_PIN         (_RD3)
#define DCPRG_PIN         (_RA2)
#define VPRG_PIN          (_RB12)
#define XLAT_PIN          (_RC2)


#define GSCLK_OC_R                  (OC3RS)

// DMA - Direct Memory Access
#define DMA3              (3)
#define PWM_PERIOD        (40)
#define CYCLE_ARRAY_SIZE    (sizeof(pwmDutyCycles))

#define DC_DATA_INITIAL_VALUE   (0xFF)
```

```
#define GS_DATA_INITIAL_VALUE   (0x00)



/************************/
/*- Structures & Unions -*/
/************************/



/************************/
/*- Function Prototypes -*/
/************************/
   void initPwmGeneratorDriver();

   void initGSCLK();
   void initSCLK();
   void initDCPRG();
   void initVPRG();
   void initXLAT();
   void initBLANK();

   void setGS(unsigned char pin, unsigned short int value);
   void setDC();

   void setAllGS(unsigned short int value);
   void setAllDC();

   void updateDC();
   void updateGS();



/***************/
```

```
/*- Variables -*/
/***************/
   BOOLEAN firstCycleFlag;
   volatile unsigned long counterGSCLK;
   unsigned long dataCounter;

   unsigned short pwmDutyCycles[1];  // set to half of PWM_PERIOD
                                     // to achieve a 50% duty cycle
   unsigned char srcSize;   // number of bytes for each
                            // transfer
   unsigned char cellSize; // size of (bytes) per transaction (16-bit PWM
                           // values requires 2 bytes
   unsigned char dstSize;  // size (bytes) of the destination (16-bit
                           // OC3RS register)
   unsigned short *volatile pDma3Dst;

   unsigned short *pDmaSrc;

   unsigned char dcData[DC_DATA_SIZE];

   unsigned char gsData[GS_DATA_SIZE];



#ifdef   __cplusplus
}
#endif

#endif   /* PWMGENERATORDRIVER_H */
```

d. PWM Generator Source File

```
#include "pwmGeneratorDriver.h"

void setMotor(unsigned char id, char direction, unsigned char power) {
        switch(id) {

        }
}

void initMotors() {

}

void initPwmGeneratorDriver() {
   firstCycleFlag = TRUE;
   counterGSCLK = 0;
   dataCounter = 0;
   pwmDutyCycles[0] = 20;
   srcSize = CYCLE_ARRAY_SIZE;
   cellSize = 2;
   dstSize = 2;
   pDma3Dst = (void *) &GSCLK_OC_R;
   pDmaSrc = pwmDutyCycles;

   int i;
   for(i = 0; i < DC_DATA_SIZE; i++) {
      dcData[i] = DC_DATA_INITIAL_VALUE;
   }

   setAllGS(GS_DATA_INITIAL_VALUE);


        DDPCONbits.TROEN = 0;
```

```
        DDPCONbits.JTAGEN = 0;
    GSCLK_TRIS = OUTPUT;
    GSCLK_PIN = LOW;
    initSCLK();
//    initDCPRG();
//    initVPRG();
    initXLAT();
    initBLANK();

    updateDC();
    initGSCLK();
    updateGS();
}

void initGSCLK() {
    // initialize to LOW
    // Step #1 - open DMA channel_0 and channel_1
    // Use channel chaining mode to start next channel when previous channel is finished
    DmaChnOpen(DMA3, 0, DMA_OPEN_CHAIN_LOW);
//    DmaChnOpen(DMA1, 0, DMA_OPEN_CHAIN_HI);


    // Step #2 - setup DMA even triggers using TIMER 4 period match interrupt
    DmaChnSetEventControl(DMA3, DMA_EV_START_IRQ(_TIMER_4_IRQ));
//    DmaChnSetEventControl(DMA1, DMA_EV_START_IRQ(_TIMER_4_IRQ));


    // Step #3 - setup the transfer pointers and transfer sizes
    DmaChnSetTxfer(DMA3, pDmaSrc, pDma3Dst, srcSize, dstSize, cellSize);
//    DmaChnSetTxfer(DMA1, pDmaSrc, pDma_1_Dst, srcSize, dstSize, cellSize);


    // Step #4 - Configure the Output Compare channels for PWM mode using Timer3
    // setup output compare channel #1 - RD3
```

```
    OpenOC3(OC_ON | OC_TIMER_MODE16 | OC_TIMER3_SRC | OC_PWM_FAULT_PIN_DISABLE, 0, 0);

    // setup output compare channel #2 - RD4
//    OpenOC4(OC_ON | OC_TIMER_MODE16 | OC_TIMER3_SRC | OC_PWM_FAULT_PIN_DISABLE, 0, 0);


    // Step #5 - Configure Timer3 and Timer4
    // Timer 3 generates 20ms period for PWMs
    OpenTimer3(T3_ON | T3_PS_1_1, PWM_PERIOD);

    // Timer4 generates 32ms triggers for DMA transfer
    OpenTimer4(T4_ON | T4_PS_1_256, 10000);


    // Step #6 - Enale DMA channel 0 to star the transfers.
    // Channel chaining will take over automatically
    DmaChnEnable(DMA3);
}

void initSCLK() {
    // initialize to LOW
    SpiOpenFlags oFlags = SPI_OPEN_MODE8 | SPI_OPEN_MSTEN;
    SpiChnOpen(TLC5940_SPI_CHANNEL, oFlags, 800);
}

void initDCPRG() {
    // initialize to LOW
    DCPRG_TRIS     =   OUTPUT;
    DCPRG_PIN      =   LOW;
}

void initVPRG() {
    // initialize to HIGH
    VPRG_TRIS    =   OUTPUT;
```

```
   VPRG_PIN   =   HIGH;
}

void initXLAT() {
   // initialize to LOW
   XLAT_TRIS    =   OUTPUT;
   XLAT_PIN   =   LOW;
}

void initBLANK() {
   // initialize to HIGH
   BLANK_TRIS   =   OUTPUT;
   BLANK_PIN  =   LOW;
}

void setGS(unsigned char pin, unsigned short int value) {
   value = (value > 4095)?4095:value;
   value = (value < 0)?0:value;

   unsigned short i;
   i = (pin * 12) / 8;

   if(i * 8 == pin * 12) {
      gsData[i] = 0x00;
      gsData[i+1] &= ~(0b11110000);
      gsData[i] = (unsigned char)(value >> 4);
      gsData[i+1] |= (unsigned char)(value << 4);
   } else {
      gsData[i] &= ~(0b00001111);
      gsData[i+1] = 0x00;
      gsData[i] |= (unsigned char)(value >> 8);
      gsData[i+1] = (unsigned char)(value);
   }
}
```

```
void setAllGS(unsigned short int value) {
   char i;
   for(i = 0; i < 16; i++) {
      setGS(i, value);
   }
}

void updateDC() {
//    setPin(DCPRG_PIN, HIGH);
//    setPin(VPRG_PIN, HIGH);

   unsigned char *p = dcData;
   int dump = 0;

   while(p < dcData + DC_DATA_SIZE) {
      SpiChnPutC(TLC5940_SPI_CHANNEL, *p++);
      dump = SpiChnGetC(TLC5940_SPI_CHANNEL);
      SpiChnPutC(TLC5940_SPI_CHANNEL, *p++);
      dump = SpiChnGetC(TLC5940_SPI_CHANNEL);
      SpiChnPutC(TLC5940_SPI_CHANNEL, *p++);
      dump = SpiChnGetC(TLC5940_SPI_CHANNEL);
   }

   while(!SpiChnTxBuffEmpty(TLC5940_SPI_CHANNEL));

   pulsePin(XLAT_PIN);
}

void updateGS() {
   if(VPRG_PIN == HIGH) {
//       setPin(VPRG_PIN, LOW);
   }
```

```
    counterGSCLK = 0;

//          BLANK_HIGH;
//          BLANK_LOW;
//    setPin(BLANK_PIN, HIGH);
//          setPin(BLANK_PIN, LOW);
//          BLANK_TOGGLE;
            pulsePin(BLANK_PIN);
            pulsePin(XLAT_PIN);

    unsigned char *p = gsData;
    int dump = 0;

    while(p < gsData + GS_DATA_SIZE && counterGSCLK < 4096) {
        SpiChnPutC(TLC5940_SPI_CHANNEL, *p++);
        dump = SpiChnGetC(TLC5940_SPI_CHANNEL);
        SpiChnPutC(TLC5940_SPI_CHANNEL, *p++);
        dump = SpiChnGetC(TLC5940_SPI_CHANNEL);
        SpiChnPutC(TLC5940_SPI_CHANNEL, *p++);
        dump = SpiChnGetC(TLC5940_SPI_CHANNEL);
    }

    while(!SpiChnTxBuffEmpty(TLC5940_SPI_CHANNEL));

//    setPin(BLANK_PIN, LOW);
//          pulsePin(XLAT_PIN);


    if(firstCycleFlag) {
        SpiChnEnable(TLC5940_SPI_CHANNEL, 0);
        TRISDbits.TRISD10 = 0;
//        PORTDbits.RD10 = 1;
//        PORTDbits.RD10 = !PORTDbits.RD10;
                    mPORTDToggleBits(10);
```

```
      SpiChnEnable(TLC5940_SPI_CHANNEL, 1);
   }

   firstCycleFlag = FALSE;
}

void __ISR(_OUTPUT_COMPARE_4_VECTOR, ipl7) OC4_IntHandler (void) {
//    IFS0CLR = 0x4000;
   counterGSCLK++;
   INTClearFlag(INT_OC3);
}

void __ISR(_TIMER_3_VECTOR, ipl2) Timer3Handler(void) {
   // clear the interrupt flag
   INTClearFlag(INT_T3);
}
```

e. Digi-Pot Driver Header File

```
#ifndef DIGIPOTDRIVER_H
#define DIGIPOTDRIVER_H

#ifdef    __cplusplus
extern "C" {
#endif

#include "commonLib.h"

//#define DIGI_LAT_TRIS         (_TRISG0)
//#define DIGI_LAT_PIN                  (_RG0)
#define DIGI_0_CS_TRIS                  (_TRISB5)
#define DIGI_0_CS_PIN                   (_RB5)
#define DIGI_1_CS_TRIS                  (_TRISA0)
#define DIGI_1_CS_PIN                   (_RA0)

#define DIGI_POT_SPI                    (SPI_CHANNEL2)

void initDigiPot();
void setDigiPot0(BYTE value);
void setDigiPot1(BYTE value);



#ifdef    __cplusplus
}
#endif

#endif   /* DIGIPOTDRIVER_H */
```

f. Digi-Pot Driver Source File

```
#include "digiPotDriver.h"

void initDigiPots() {
	SpiOpenFlags oFlags = SPI_OPEN_MODE16 | SPI_OPEN_MSTEN;
	SpiChnOpen(DIGI_POT_SPI, oFlags, 20);

	WORD setUpConAddr1 = 0b0100000001111111;

//	setPin(DIGI_LAT_TRIS, OUTPUT);
	AD1PCFGSET = 0x0020;
	setPin(DIGI_0_CS_TRIS, OUTPUT);
	setPin(DIGI_1_CS_PIN, OUTPUT);


//	setPin(DIGI_LAT_PIN, HIGH);
	setPin(DIGI_0_CS_PIN, HIGH);
	setPin(DIGI_1_CS_PIN, HIGH);




	setPin(DIGI_0_CS_PIN, LOW);

	SpiChnPutC(DIGI_POT_SPI, setUpConAddr1);
	SpiChnGetC(DIGI_POT_SPI);

	while(!SpiChnTxBuffEmpty(DIGI_POT_SPI));

	setPin(DIGI_0_CS_PIN, HIGH);
```

```
	setPin(DIGI_1_CS_PIN, LOW);

	SpiChnPutC(DIGI_POT_SPI, setUpConAddr1);
	SpiChnGetC(DIGI_POT_SPI);

	while(!SpiChnTxBuffEmpty(DIGI_POT_SPI));

	setPin(DIGI_1_CS_PIN, HIGH);
}

void setDigiPot0(BYTE value) {
	WORD Byte1 = (0b0000000000000000 | value);
	WORD Byte2 = 0b0000110000001111;

	setPin(DIGI_0_CS_PIN, LOW);
//	setPin(DIGI_LAT_PIN, LOW);

	//	SpiChnPutC(ADC_SENSOR_SPI, Byte1);
	//	SpiChnGetC(ADC_SENSOR_SPI);
	SpiChnPutC(DIGI_POT_SPI, Byte1);
	//	SpiChnPutC(ADC_SENSOR_SPI, Byte2);
	Byte2 = SpiChnGetC(DIGI_POT_SPI);

	while(!SpiChnTxBuffEmpty(DIGI_POT_SPI));

//	setPin(DIGI_LAT_PIN, HIGH);
	setPin(DIGI_0_CS_PIN, HIGH);
}

void setDigiPot1(BYTE value) {
	WORD Byte1 = (0b0000000000000000 | value);
```

```
        WORD Byte2 = 0b0000110000001111;

        setPin(DIGI_1_CS_PIN, LOW);
//      setPin(DIGI_LAT_PIN, LOW);

        //      SpiChnPutC(ADC_SENSOR_SPI, Byte1);
        //      SpiChnGetC(ADC_SENSOR_SPI);
        SpiChnPutC(DIGI_POT_SPI, Byte1);
        //      SpiChnPutC(ADC_SENSOR_SPI, Byte2);
        Byte2 = SpiChnGetC(DIGI_POT_SPI);

        while(!SpiChnTxBuffEmpty(DIGI_POT_SPI));

//      setPin(DIGI_LAT_PIN, HIGH);
        setPin(DIGI_1_CS_PIN, HIGH);
}
```

g. Control Manager Header File

```
#ifndef CONTROLMANAGER_H
#define CONTROLMANAGER_H

#ifdef    __cplusplus
extern "C" {
#endif

#include "commonLib.h"
#include "commonStructs.h"


#define MAX_CONTROL_INSTANCES   7

#define THUMB_MOTOR_PIN     0
#define THUMB_MOTOR_DIR     0
#define THUMB_ENCODER_PIN   0
#define THUMB_CURRENT_PIN   0
#define THUMB_SER_EL_PIN    0

#define THUMB_MOTOR_PIN     0
#define THUMB_MOTOR_DIR     0
#define THUMB_ENCODER_PIN   0
#define THUMB_CURRENT_PIN   0
#define THUMB_SER_EL_PIN    0


   typedef struct _Motor {
      unsigned char motorPin;
      signed char direction;
      unsigned char encoderPin;
      unsigned char currentPin;
      unsigned char seriesElasticPin;
```

```
} Motor;

typedef struct _ControlInstance {
    unsigned char controlType;
    unsigned char status;
    short int desiredEncoderPos;
    short int desiredEncoderVel;
    unsigned short int desiredCurrentAdc;
    unsigned short int desiredSeriesElasticAdc;
} ControlInstance;

typedef struct _ControlInstanceManager {
    unsigned char runningInstances;
    unsigned char controlIndex[MAX_CONTROL_INSTANCES];
    ControlInstance controlInstances[MAX_CONTROL_INSTANCES];
    Motor motors[MAX_CONTROL_INSTANCES];
} ControlInstanceManager;



#ifdef  __cplusplus
}
#endif

#endif  /* CONTROLMANAGER_H */
```

h. Control Manager Source File

```
#include "controlManager.h"
#include "pwmGeneratorDriver.h"

void cmNewControlManager(ControlInstanceManager *cm) {
   cm->runningInstances = 0;
}

void cmNewControlInstance(ControlInstance *ci, MainTxMotorRxPacket *p) {
   ci->controlType = p->use.header._controlType;
   ci->status = 0;
   ci->desiredEncoderPos = p->use.encoderPos;
   ci->desiredEncoderVel = p->use.encoderVel;
   ci->desiredCurrentAdc = p->use.currentAdc;
   ci->desiredSeriesElasticAdc = p->use.seriesElasticAdc;
}

void cmExecControl(ControlInstanceManager *cm) {

        unsigned char i;
        for(i = 0; i < cm->runningInstances; i++) {
                ControlInstance ci = cm->controlInstances[cm->controlIndex[i]];

                switch(ci.controlType) {
                        case 0:
                                if(ci.desiredEncoderPos > 0) {
                                        setAllGS(ci.desiredEncoderPos);
                                } else if(ci.desiredEncoderPos < 0) {

                                } else {

                                }
                                break;
```

```
                }
            }

            updateGS();
}

void cmInitMotors() {
//    motors[0].motorPin = 0;
   // ...
            initPwmGeneratorDriver();
}

void cmStartInstance(ControlInstanceManager *cm, MainTxMotorRxPacket *p) {
   // check to see if existing instance is already running
   unsigned char i;
   unsigned char id = 0;
   id = p->use.header._id;

   for(i = 0; i < cm->runningInstances; i++) {
      if(cm->controlIndex[i] == id) { // if instance is already running
//          cm->controlInstances[id] = *ci;
         cmNewControlInstance(&(cm->controlInstances[id]), p);
         return;
      }
   }

   // if none exists, create a new control instance
   cm->controlIndex[cm->runningInstances] = id;
   cm->runningInstances++;
//    cm->controlInstances[id] = *ci;
   cmNewControlInstance(&(cm->controlInstances[id]), p);
}

void cmStopInstance(ControlInstanceManager *cm, MainTxMotorRxPacket *p) {
```

```
    unsigned char i;
    unsigned char id = 0;
    id = p->use.header._id;
    BOOL slideOver = FALSE;

    for(i = 0; i < cm->runningInstances; i++) {
        if(id == cm->controlIndex[i]) {
            slideOver = TRUE;
        }

        if(slideOver) {
            if(i+1 < cm->runningInstances) {
                cm->controlIndex[i] = cm->controlIndex[i+1];
            } else {
                cm->runningInstances--;
                break;
            }
        }
    }
}
```

i. Publish Manager Header File

```
#ifndef PUBLISHMANAGER_H
#define PUBLISHMANAGER_H

#ifdef    __cplusplus
extern "C" {
#endif

#include "commonLib.h"
#include "commonStructs.h"

        typedef struct _MotorPublishManager {
                unsigned char runningPublishers;
      unsigned char publishIndex[MOTOR_GROUPS];
      MotorTxMainRxPacket packets[MOTOR_GROUPS];
        } MotorPublishManager;



#ifdef    __cplusplus
}
#endif

#endif   /* PUBLISHMANAGER_H */
```

j. Publish Manager Source File

```
#include "publishManager.h"

void pmNewManager(MotorPublishManager *pm) {
	pm->runningPublishers = 0;
	unsigned char i;
	for(i = 0; i < MOTOR_GROUPS; i++) {
		pm->publishIndex[i] = 0;
	}
}

void pmStartPublish(MotorPublishManager *pm, unsigned char id) {
	unsigned char i;

	if(pm->runningPublishers == 0) {
		pm->publishIndex[0] = id;
		pm->runningPublishers++;
		return;
	}

	// check if id already exists
  for(i = 0; i < pm->runningPublishers; i++) {
    if(pm->publishIndex[i] == id) {
      return;
    }
  }

	// if not exists, add
	// this keeps the indicies in order
	for(i = 0; i < pm->runningPublishers; i++) {
		if(id > pm->publishIndex[i]) {
			unsigned char j;
			for(j = i+1; j < pm->runningPublishers; j++) {
```

```
                        pm->publishIndex[j+1] = pm->publishIndex[j];
                    }

                    pm->publishIndex[i+1] = id;
                    pm->runningPublishers++;
                    break;
                }
            }
}

void pmStopPublish(MotorPublishManager *pm, unsigned char id) {
        unsigned char i;
        // check if there is an id to stop
        for(i = 0; i < pm->runningPublishers; i++) {
            if(pm->publishIndex[i] == id) {
                unsigned char j;
                pm->runningPublishers--;
                for(j = i; j < pm->runningPublishers; j++) {
                    pm->publishIndex[j] = pm->publishIndex[j+1];
                }

                break;
            }
        }
}


void pmPublish(MotorPublishManager *pm, unsigned char u) {
        unsigned char i;
   for(i = 0; i < pm->runningPublishers; i++) {
      commonUARTSendData(u, pm->packets[pm->publishIndex[i]].stream, MOTOR_TX_MAIN_RX_PACKET_SIZE);
   }
}
```

J. Shared Libraries

a. Common UART Header File

```
/**
 * @author Jason Klein (jason.e.klein@wpi.edu)
 *
 *      Setup UART communication on the PIC32. When receiving, this implementation
 * expects to receive a packet of a specific size on a regular basis. It is able
 * to detect when a packet is not received and then perform some action on it.
 * It uses struct RecvManager to ensure data is received in the specified time.
 * A RecvManager is used for each UART connection. There should be equal numbers
 * of UART connections used as RecvManagers used.
 *
 * @pre Requires a timer to be running. Start a timer with commonTimer.h
 *
 * @note                The timer can be used for other task and is not just dedicated
 *                              to run the UART RecvManager. It accounts for timer overflow.
 */

#ifndef COMMONUART_H
#define COMMONUART_H

#ifdef   __cplusplus
extern "C" {
#endif

        /************************/
        /*- Include Directives -*/
        /************************/
#include "commonConfig.h"
#include "commonStructs.h"
```

```
    /************************/
    /*- Function Prototypes -*/
    /************************/
    /**
     * Initializes a UART port
     *
     * @param uID                   The UART ID of the port to initialize
     * @param systemClock           The clock frequency of this microntroller
     * @param desiredBaudRate       The desirted baud rate for this port
     */
    void commonUARTInit(UART_MODULE uID, unsigned long systemClock, unsigned long desiredBaudRate);

    /**
     * Sends a byte
     *
* @param uID                      The UART ID of the port to use
* @param c                        The byte to send
*/
    void commonUARTSendByte(UART_MODULE uID, const BYTE c);

    /**
     * Sends a byte array
     *
* @param uID                      The UART ID of the port to use
* @param data                     The byte array to send
* @param size                     The length of the byte array
*/
    void commonUARTSendData(UART_MODULE uID, const BYTE *data, unsigned int size);

    /**
     * Sends a character of the corresponding byte
     *
* @param uID                      The UART ID of the port to use
```

```
* @param c                    The HEX value to send, must be 0-15 inclusive
*/
    void commonUARTSendHexByte(UART_MODULE uID, const BYTE c);

    /**
     * Initializes a new struct RecvManager
     *
* @param rm                   The RecvManager to initialize
* @param bytesExpected        The number of bytes to expect in each packet
* @param timeout              The number of clock ticks in which a packet should be received
*/
    void commonUARTNewRecvManager(RecvManager *rm, unsigned char bytesExpected, unsigned long timeout);

    /**
     * Clears the Receive Manger to await for another packet
     *
* @param rm                   The RecvManager to clear
*/
    void commonUARTClearRecvManager(RecvManager *rm);

    /**
     * Checks to see if the entire packet has been received
     *
* @param rm                   The RecvManager to use
* @param timer                The timer incrementer
* @return                     1 if transfer is complete, 0 otherwise
     *
     * @note                        Must run in a loop
*/
    BOOL commonUARTTransferComplete(RecvManager *rm, volatile unsigned long *timer);

    /**
     * Adds received bytes to an array
     *
```

```
    * @param rm                    The RecvManager to use
    * @param timer                 The timer incrementer
    * @param dataByte              The data byte just received
    * @param dataStream            The data array to add the byte to
    */
        void commonUARTRecvData(RecvManager *rm, volatile unsigned long *timer, volatile unsigned char *dataByte, unsigned char
*dataStream);

#ifdef   __cplusplus
}
#endif

#endif   /* COMMONUART_H */
```

b. Common UART Source File

```
/**
 * @author      Jason Klein      (jason.e.klein@wpi.edu)
 * @param uID
 * @param systemClock
 * @param desiredBaudRate
 */

#include "commonUART.h"

void commonUARTInit(UART_MODULE uID, unsigned long systemClock, unsigned long desiredBaudRate) {
    // Configure the device for maximum performance but do not change the PBDIV
    // Given the options, this function will change the flash wait states, RAM
    // wait state and enable prefetch cache but will not change the PBDIV.
    // The PBDIV value is already set via the pragma FPBDIV option above..
//    SYSTEMConfig(systemClock, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

    UARTConfigure(uID, UART_ENABLE_PINS_TX_RX_ONLY);
    UARTSetFifoMode(uID, UART_INTERRUPT_ON_TX_NOT_FULL | UART_INTERRUPT_ON_RX_NOT_EMPTY);
    UARTSetLineControl(uID, UART_DATA_SIZE_8_BITS | UART_PARITY_NONE | UART_STOP_BITS_1);
    UARTSetDataRate(uID, systemClock, desiredBaudRate);
    UARTEnable(uID, UART_ENABLE_FLAGS(UART_PERIPHERAL | UART_RX | UART_TX));

    // Configure UART_MODULE_ID RX Interrupt
    INTEnable(INT_SOURCE_UART_RX(uID), INT_ENABLED);
    INTSetVectorPriority(INT_VECTOR_UART(uID), INT_PRIORITY_LEVEL_2);
    INTSetVectorSubPriority(INT_VECTOR_UART(uID), INT_SUB_PRIORITY_LEVEL_0);

    // configure for multi-vectored mode
    INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);

    // enable interrupts
    INTEnableInterrupts();
```

```
}

void commonUARTSendByte(UART_MODULE uID, const BYTE c) {
   while (!UARTTransmitterIsReady(uID));

   UARTSendDataByte(uID, c);

   while (!UARTTransmissionHasCompleted(uID));
}

void commonUARTSendData(UART_MODULE uID, const BYTE *data, unsigned int size) {
   while(size) {
      while (!UARTTransmitterIsReady(uID));

      UARTSendDataByte(uID, *data);

      data++;
      size--;
   }
   while (!UARTTransmissionHasCompleted(uID));
}

void commonUARTSendHexByte(UART_MODULE uID, const BYTE c) {
         switch(c) {
                  case 58:
                           commonUARTSendByte(uID, 'A');
                           break;
                  case 59:
                           commonUARTSendByte(uID, 'B');
                           break;
                  case 60:
                           commonUARTSendByte(uID, 'C');
                           break;
                  case 61:
```

```
                commonUARTSendByte(uID, 'D');
                break;
            case 62:
                commonUARTSendByte(uID, 'E');
                break;
            case 63:
                commonUARTSendByte(uID, 'F');
                break;
            default:
                commonUARTSendByte(uID, c);
                break;
        }
}

void commonUARTNewRecvManager(RecvManager *rm, unsigned char bytesExpected, unsigned long timeout) {
   rm->firstByteTime = 0;
   rm->timout = timeout;
   rm->timeoutTime = 0;
   rm->overflow = FALSE;
   rm->bytesExpected = bytesExpected;
   rm->bytesReceived = 0;
   rm->transferComplete = FALSE;
}

void commonUARTClearRecvManager(RecvManager *rm) {
        rm->firstByteTime = 0;
        rm->timeoutTime = 0;
        rm->overflow = FALSE;
        rm->bytesReceived = 0;
        rm->transferComplete = FALSE;
}

BOOL commonUARTTransferComplete(RecvManager *rm, volatile unsigned long *timer) {
   if(rm->transferComplete) {
```

```
//      commonUARTClearRecvManager(rm);
                rm->transferComplete = FALSE;
      return TRUE;
   }

   if(rm->bytesReceived) {
      // did the timeout overflow?
      if(rm->overflow) {
         // did timer overflow and has is timer still below the timeout
         if(!(*timer < rm->firstByteTime && *timer < rm->timeoutTime)) { // timeout limit has been exceeded
            // took too long to receive data
            rm->bytesReceived = 0;
//             _RD1 = 0x1;
         }
      } else {
         if(!(*timer < rm->timeoutTime)) { // timeout limit has been exceeded
            // took too long to receive data
            rm->bytesReceived = 0;
//             _RD1 = 0x1; // turn LED on
         }
      }
   }

   return FALSE;
}

void commonUARTRecvData(RecvManager *rm, volatile unsigned long *timer, volatile unsigned char *dataByte, unsigned char *dataStream) {
   // is this the first byte received?
   if(!rm->bytesReceived) { //rm->bytesReceived == 0
//      PORTDbits.w &= ~(0x1 | 0x2); // turn both LEDs off
      rm->firstByteTime = *timer; // mark down time received
      rm->timeoutTime = *timer + rm->timout; // mark down time stream should be received by

//            rm->bytes[rm->bytesReceived] = UARTGetDataByte(UART_MODULE_ID); // get received byte
```

```
      dataStream[rm->bytesReceived] = *dataByte; // get received byte
      rm->bytesReceived++; // increment received byte counter

//                  if(rm->bytesExpected == 1 && rm->bytesReceived == rm->bytesExpected) {
//                        rm->transferComplete = TRUE;
//                  }

      // will the timeout time overflow because time received is near max variable value?
      if(rm->timeoutTime > *timer) {
         rm->overflow = FALSE;
      } else {
         rm->overflow = TRUE;
      }
   } else { // not the first byte received
      // did the timeout overflow?
      if(rm->overflow) {
         // did timer overflow and has is timer still below the timeout
         if(*timer < rm->firstByteTime && *timer < rm->timeoutTime) {
            dataStream[rm->bytesReceived] = *dataByte; // get received byte
            rm->bytesReceived++;
         } else { // timeout limit has been exceeded
            // took too long to receive data
            rm->bytesReceived = 0;
//             _RD1 = 0x1;
         }
      } else {
         if(*timer < rm->timeoutTime) {
            dataStream[rm->bytesReceived] = *dataByte; // get received byte
            rm->bytesReceived++;
         } else { // timeout limit has been exceeded
            // took too long to receive data
            rm->bytesReceived = 0;
//             _RD1 = 0x1;
         }
```

```
    }
  }

        if(rm->bytesReceived == rm->bytesExpected) {
                // assign rm->bytes[] --> _Tx_RxPacket
                // reset values of rBuf
//          _RD0 = 0x1;
                rm->transferComplete = TRUE;
                rm->bytesReceived = 0;
        }
}
```

c. Common Timer Header File

```
/**
 * @author Jason Klein (jason.e.klein@wpi.edu)
 *
 *      Timer functions designed to be simple to use on the PIC32. This code
 * is used to execute a section of code at a desired frequency.
 *
 * @example
 *                              volatile unsigned long myIncrementer; // must be global
 *
 *                              void main() {
 *                                      ExecTimer myExecTimer;
 *                                      commonTimerNewExecTimer(&myExecTimer, &myIncrementer, 1001);
 *
 *                                      commonTimerInit(2, &myIncrementer, 312, T2_PS_1_256);
 *
 *                                      while(1) {
 *                                              if(commonTimerExecEvery(&myExecTimer, &myIncrementer)) {
 *                                                      // do something here aboue every 1 second
 *                                              }
 *                                      }
 *                              }
 *
 *                              void __ISR(_TIMER_2_VECTOR, ipl2) Timer2Handler(void) {
 *                                      INTClearFlag(INT_T2);    // clear the interrupt flag
 *                                      myIncrementer++;
 *                              }
 *
 * @TODO                Add improvement to account for overflow of the timer variable
 */

#ifndef COMMONTIMER_H
#define COMMONTIMER_H
```

```
#ifdef   __cplusplus
extern "C" {
#endif

/************************/
/*- Include Directives -*/
/************************/
#include "commonConfig.h"
#include "commonStructs.h"



/************************/
/*- Macro Definitions -*/
/************************/
#define BOOLEAN unsigned char
#define TRUE 1
#define FALSE 0



/*************************/
/*- Function Prototypes -*/
/*************************/
        /**
         * Initializes a new timer
         *
         * @param timer                 The desired timer to use
         * @param cmnTmr      The global variable inside the ISR that is to be
         *                                      incremented when the ISR fires
         * @param period        The period for this timer
         * @param prescaler     The prescaler for this timer
         *
```

```
     * @example
     *                                  volatile unsigned long myIncrementer; // must be global
     *                                  commonTimerInit(1, &myIncrementer, 312, T2_PS_1_256);
     *                                  // this example fires at about 1KHz
     *
     * @note                    Calculation to determine the frequency:
     *                                  frequency=(clock frequency)/(prescaler * period)
     *                                  frequency=(80,000,000)/(256 * 312)
     *                                  frequency = ~1001.6Hz
  *
  * @TODO          This function can be improved by converting it
  *              to a macro expansion
     */
    void commonTimerInit(unsigned char timer, volatile unsigned long *cmnTmr, unsigned short int period, unsigned short int prescaler);

    /**
     * Creates a new ExecTimer, the struct ExecTimer is used in conjunction
     * with commonTimerExecEvery() to execute a section of code a a specific frequency
     *
* @param et          The ExecTimer to initialize
* @param cmnTmr   The global variable inside the ISR that is to be
     *                                  incremented when the ISR fires
* @param every             The number of increments to pass before
     *                                  commonTimerExecEvery() returns true.
     *
     * @example                 ExecTimer myExecTimer;
     *                                  commonTimerNewExecTimer(&myExecTimer, &myIncrementer, 1001);
     *                                  // this will initialize a struct and will execute every
     *                                  // 1001 ticks or about 1Hz when building off the example
     *                                  // in commonTimerInit()
*/
    void commonTimerNewExecTimer(ExecTimer *et, volatile unsigned long *cmnTmr, unsigned long every);

    /**
```

```
         * This is used to execute a section of code at a specific frequency.
         * Used in conjuction with commonTimerInit() and commonTimerNewExecTimer()
         *
    * @param et            The ExecTimer to use that was initialized by
         *                                      commonTimerNewExecTimer()
    * @param cmnTmr    The global variable inside the ISR that is to be
         *                                      incremented when the ISR fires
    * @return                      1 if *cmnTmr exceeds et->timerSave
         *                                      0 if not
         *
         * @example                     if(commonTimerExecEvery(&myExecTimer, &myIncrementer)) {
         *                                              // execute code here every ~1 second
         *                                      }
         *
         * @note                        must run in a loop
    */
        BOOLEAN commonTimerExecEvery(ExecTimer *et, volatile unsigned long *cmnTmr);

        /**
         * Delays code execution for the given number of clock ticks
    * @param cmnTmr    The global variable inside the ISR that is to be
         *                                      incremented when the ISR fires
    * @param t                     The number of increments to pass before code exection
         *                                      continutes past this point
    */
        void commonTimerDelay(volatile unsigned long *cmnTmr, unsigned long t);


#ifdef   __cplusplus
}
#endif

#endif   /* COMMONTIMER_H */
```

d. Common Timer Source File

```
#include "commonTimer.h"


void commonTimerInit(unsigned char timer, volatile unsigned long *cmnTmr, unsigned short int period, unsigned short int prescaler) {
   *cmnTmr = 1;

        switch(timer) {
                case 1:
                        // Configure Timer using PBCLK as input, prescaler
                        // Period matches the Timer frequency, so the interrupt handler will trigger
                        OpenTimer1(T1_ON | T1_SOURCE_INT | prescaler, period);

                        // Set up the timer interrupt with a priority of 2
                        INTEnable(INT_T1, INT_ENABLED);
                        INTSetVectorPriority(INT_TIMER_1_VECTOR, INT_PRIORITY_LEVEL_2);
                        INTSetVectorSubPriority(INT_TIMER_1_VECTOR, INT_SUB_PRIORITY_LEVEL_0);

                        // Enable multi-vector interrupts
                        INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);
                        INTEnableInterrupts();
                        break;
                case 2:
                        // Configure Timer using PBCLK as input, prescaler
                        // Period matches the Timer frequency, so the interrupt handler will trigger
                        OpenTimer2(T2_ON | T2_SOURCE_INT | prescaler, period);

                        // Set up the timer interrupt with a priority of 2
                        INTEnable(INT_T2, INT_ENABLED);
                        INTSetVectorPriority(INT_TIMER_2_VECTOR, INT_PRIORITY_LEVEL_2);
                        INTSetVectorSubPriority(INT_TIMER_2_VECTOR, INT_SUB_PRIORITY_LEVEL_0);

                        // Enable multi-vector interrupts
```

```
        INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);
        INTEnableInterrupts();
        break;
case 3:
        // Configure Timer using PBCLK as input, prescaler
        // Period matches the Timer frequency, so the interrupt handler will trigger
        OpenTimer3(T3_ON | T3_SOURCE_INT | prescaler, period);

        // Set up the timer interrupt with a priority of 2
        INTEnable(INT_T3, INT_ENABLED);
        INTSetVectorPriority(INT_TIMER_3_VECTOR, INT_PRIORITY_LEVEL_2);
        INTSetVectorSubPriority(INT_TIMER_3_VECTOR, INT_SUB_PRIORITY_LEVEL_0);

        // Enable multi-vector interrupts
        INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);
        INTEnableInterrupts();
        break;
case 4:
        // Configure Timer using PBCLK as input, prescaler
        // Period matches the Timer frequency, so the interrupt handler will trigger
        OpenTimer4(T4_ON | T4_SOURCE_INT | prescaler, period);

        // Set up the timer interrupt with a priority of 2
        INTEnable(INT_T4, INT_ENABLED);
        INTSetVectorPriority(INT_TIMER_4_VECTOR, INT_PRIORITY_LEVEL_2);
        INTSetVectorSubPriority(INT_TIMER_4_VECTOR, INT_SUB_PRIORITY_LEVEL_0);

        // Enable multi-vector interrupts
        INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);
        INTEnableInterrupts();
        break;
case 5:
        // Configure Timer using PBCLK as input, prescaler
        // Period matches the Timer frequency, so the interrupt handler will trigger
```

```
                        OpenTimer5(T5_ON | T5_SOURCE_INT | prescaler, period);

                        // Set up the timer interrupt with a priority of 2
                        INTEnable(INT_T5, INT_ENABLED);
                        INTSetVectorPriority(INT_TIMER_5_VECTOR, INT_PRIORITY_LEVEL_2);
                        INTSetVectorSubPriority(INT_TIMER_5_VECTOR, INT_SUB_PRIORITY_LEVEL_0);

                        // Enable multi-vector interrupts
                        INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);
                        INTEnableInterrupts();
                        break;
        }
}

void commonTimerNewExecTimer(ExecTimer *et, volatile unsigned long *cmnTmr, unsigned long every) {
   et->every = every;
   et->timerSave = *cmnTmr + every;
}

BOOLEAN commonTimerExecEvery(ExecTimer *et, volatile unsigned long *cmnTmr) {
   if(*cmnTmr == et->timerSave) {
      et->timerSave = *cmnTmr + et->every;
      return TRUE;
   }
   return FALSE;
}

void commonTimerDelay(volatile unsigned long *cmnTmr, unsigned long t) {
        ExecTimer et;
        commonTimerNewExecTimer(&et, cmnTmr, t);

        while(1) {
                if(commonTimerExecEvery(&et, cmnTmr)) {
                        break;
```

```
                }
            }
        }
```

e. Common ADC Header File

```
#ifndef COMMONADC_H
#define COMMONADC_H

#ifdef    __cplusplus
extern "C" {
#endif

#include "commonConfig.h"
#include "commonStructs.h"

/***********************/
/*- Macro Definitions -*/
/***********************/
////////////////// DO NOT MODIFY
#define BOOLEAN       unsigned char
#define TRUE          1
#define FALSE         0

////////////////// AS NEEDED, MODIFY ONLY THESE #define SETTINGS BELOW
#define ADC_SENSOR_SPI                  (SPI_CHANNEL1)

#define ADC_FLEX_CS_TRIS       (_TRISA4)
#define ADC_FLEX_CS_PIN                 (_RA4)

#define ADC_PRES_CS_TRIS       (_TRISA5)
#define ADC_PRES_CS_PIN                 (_RA5)

#define ADC_MOTOR_SPI                   (SPI_CHANNEL1)

#define ADC_SE_CS_TRIS                  (_TRISD12)
#define ADC_SE_CS_PIN                   (_RD12)
```

```
#define ADC_CURR_CS_TRIS      (_TRISC1)
#define ADC_CURR_CS_PIN               (_RC1)


        void commonADCSetupSensorBoard(unsigned char flexChannelEnd, unsigned char presChannelEnd);

        void commonADCReadAllFlex(unsigned char channelEnd, unsigned  short int readData[8]);
        unsigned short int commonADCReadFlex(unsigned char channelEnd);

        void commonADCReadAllPres(unsigned char channelEnd, unsigned  short int readData[8]);
        unsigned short int commonADCReadPres(unsigned char channelEnd);



        void commonADCSetupMotorBoard(unsigned char seChannelEnd, unsigned char currChannelEnd);

        void commonADCReadAllSe(unsigned char channelEnd, unsigned  short int readData[8]);
        unsigned short int commonADCReadSe(unsigned char channelEnd);

        void commonADCReadAllCurr(unsigned char channelEnd, unsigned  short int readData[8]);
        unsigned short int commonADCReadCurr(unsigned char channelEnd);

#ifdef   __cplusplus
}
#endif

#endif   /* COMMONADC_H */
```

f. Common ADC Source File

```
#include "commonADC.h"


/*************************************/
/** BEGIN - Common ADC - Sensor Board **/
/*************************************/
void commonADCSetupSensorBoard(unsigned char flexChannelEnd, unsigned char presChannelEnd) {
	BYTE recvByte1 = 0;
	BYTE recvByte2 = 0;

//	setPin(ADC_FLEX_CS_TRIS, OUTPUT);
//	setPin(ADC_PRES_CS_TRIS, OUTPUT);

	DDPCONbits.TROEN = 0;
	DDPCONbits.JTAGEN = 0;

	TRISACLR |= (BIT_4 | BIT_5);

//	setPin(ADC_FLEX_CS_PIN, HIGH);
//	setPin(ADC_PRES_CS_PIN, HIGH);

	PORTASET |= (BIT_4 | BIT_5);

	SpiOpenFlags oFlags = SPI_OPEN_MODE8 | SPI_OPEN_MSTEN;
	SpiChnOpen(ADC_SENSOR_SPI, oFlags, 256);


	// setup flex adc
	BYTE setupByte1f = 0b11000011;
	setupByte1f |= (flexChannelEnd << 2);
	BYTE setupByte2f = 0b10010000;
```

```
        unsigned short int valf = commonADCReadFlex(flexChannelEnd);

        setPin(ADC_FLEX_CS_PIN, LOW);

        SpiChnPutC(ADC_SENSOR_SPI, setupByte1f);
        recvByte1 = SpiChnGetC(ADC_SENSOR_SPI);

        SpiChnPutC(ADC_SENSOR_SPI, setupByte2f);
        recvByte2 = SpiChnGetC(ADC_SENSOR_SPI);

        while(!SpiChnTxBuffEmpty(ADC_SENSOR_SPI));

        setPin(ADC_FLEX_CS_PIN, HIGH);

        // setup pres adc
        BYTE setupByte1p = 0b11000011;
        setupByte1p |= (presChannelEnd << 2);
        BYTE setupByte2p = 0b10010000;

        unsigned short int valp = commonADCReadPres(presChannelEnd);

        setPin(ADC_PRES_CS_PIN, LOW);

        SpiChnPutC(ADC_SENSOR_SPI, setupByte1p);
        recvByte1 = SpiChnGetC(ADC_SENSOR_SPI);

        SpiChnPutC(ADC_SENSOR_SPI, setupByte2p);
        recvByte2 = SpiChnGetC(ADC_SENSOR_SPI);

        while(!SpiChnTxBuffEmpty(ADC_SENSOR_SPI));

        setPin(ADC_PRES_CS_PIN, HIGH);
}
```

```
unsigned short int commonADCReadFlex(unsigned char channelEnd) {
	BYTE sendByte1 = 0b01000011;
	sendByte1 |= (channelEnd << 2);
	BYTE sendByte2 = 0b10010000;
	BYTE recvByte1 = 0;
	BYTE recvByte2 = 0;
	unsigned short int recvVal = 0;

	setPin(ADC_FLEX_CS_PIN, LOW);

	SpiChnPutC(ADC_SENSOR_SPI, sendByte1);
	recvByte1 = SpiChnGetC(ADC_SENSOR_SPI);

	SpiChnPutC(ADC_SENSOR_SPI, sendByte2);
	recvByte2 = SpiChnGetC(ADC_SENSOR_SPI);

	while(!SpiChnTxBuffEmpty(ADC_SENSOR_SPI));

	recvVal = recvByte1;
	recvVal = recvVal << 8;
	recvVal |= recvByte2;

	setPin(ADC_FLEX_CS_PIN, HIGH);

	return recvVal;
}

void commonADCReadAllFlex(unsigned char channelEnd, unsigned short int readData[8]) {
	unsigned char i;
	for(i = 0; i <= channelEnd; i++) {
		unsigned short int data = commonADCReadFlex(channelEnd);
		unsigned char id = (unsigned char)(data >> 12);
		readData[id] = data & ~(0b1111000000000000);
	}
```

```
}

unsigned short int commonADCReadPres(unsigned char channelEnd) {
	BYTE sendByte1 = 0b01000011;
	sendByte1 |= (channelEnd << 2);
	BYTE sendByte2 = 0b10010000;
	BYTE recvByte1 = 0;
	BYTE recvByte2 = 0;
	unsigned short int recvVal = 0;

	setPin(ADC_PRES_CS_PIN, LOW);

	SpiChnPutC(ADC_SENSOR_SPI, sendByte1);
	recvByte1 = SpiChnGetC(ADC_SENSOR_SPI);

	SpiChnPutC(ADC_SENSOR_SPI, sendByte2);
	recvByte2 = SpiChnGetC(ADC_SENSOR_SPI);

	while(!SpiChnTxBuffEmpty(ADC_SENSOR_SPI));

	recvVal = recvByte1;
	recvVal = recvVal << 8;
	recvVal |= recvByte2;

	setPin(ADC_PRES_CS_PIN, HIGH);

	return recvVal;
}

void commonADCReadAllPres(unsigned char channelEnd, unsigned short int readData[8]) {
	unsigned char i;
	for(i = 0; i <= channelEnd; i++) {
		unsigned short int data = commonADCReadPres(channelEnd);
		unsigned char id = (unsigned char)(data >> 12);
```

```
                readData[id] = data & ~(0b1111000000000000);
        }
}
/*************************************/
/** END - Common ADC - Sensor Board **/
/*************************************/



/*************************************/
/** BEGIN - Common ADC - Motor Board **/
/*************************************/
void commonADCSetupMotorBoard(unsigned char seChannelEnd, unsigned char currChannelEnd) {
        BYTE recvByte1 = 0;
        BYTE recvByte2 = 0;

        setPin(ADC_SE_CS_TRIS, OUTPUT);
        setPin(ADC_CURR_CS_TRIS, OUTPUT);

        setPin(ADC_SE_CS_PIN, HIGH);
        setPin(ADC_CURR_CS_PIN, HIGH);

        SpiOpenFlags oFlags = SPI_OPEN_MODE8 | SPI_OPEN_MSTEN;
        SpiChnOpen(ADC_MOTOR_SPI, oFlags, 256);


        // setup flex adc
        BYTE setupByte1s = 0b11000011;
        setupByte1s |= (seChannelEnd << 2);
        BYTE setupByte2s = 0b10010000;

        unsigned short int valf = commonADCReadSe(seChannelEnd);

        setPin(ADC_SE_CS_PIN, LOW);
```

```
	SpiChnPutC(ADC_MOTOR_SPI, setupByte1s);
	recvByte1 = SpiChnGetC(ADC_MOTOR_SPI);

	SpiChnPutC(ADC_MOTOR_SPI, setupByte2s);
	recvByte2 = SpiChnGetC(ADC_MOTOR_SPI);

	while(!SpiChnTxBuffEmpty(ADC_MOTOR_SPI));

	setPin(ADC_SE_CS_PIN, HIGH);

	// setup pres adc
	BYTE setupByte1c = 0b11000011;
	setupByte1c |= (currChannelEnd << 2);
	BYTE setupByte2c = 0b10010000;

	unsigned short int valp = commonADCReadCurr(currChannelEnd);

	setPin(ADC_CURR_CS_PIN, LOW);

	SpiChnPutC(ADC_MOTOR_SPI, setupByte1c);
	recvByte1 = SpiChnGetC(ADC_MOTOR_SPI);

	SpiChnPutC(ADC_MOTOR_SPI, setupByte2c);
	recvByte2 = SpiChnGetC(ADC_MOTOR_SPI);

	while(!SpiChnTxBuffEmpty(ADC_MOTOR_SPI));

	setPin(ADC_CURR_CS_PIN, HIGH);
}

unsigned short int commonADCReadSe(unsigned char channelEnd) {
	BYTE sendByte1 = 0b01000011;
	sendByte1 |= (channelEnd << 2);
```

```
	BYTE sendByte2 = 0b10010000;
	BYTE recvByte1 = 0;
	BYTE recvByte2 = 0;
	unsigned short int recvVal = 0;

	setPin(ADC_SE_CS_PIN, LOW);

	SpiChnPutC(ADC_MOTOR_SPI, sendByte1);
	recvByte1 = SpiChnGetC(ADC_MOTOR_SPI);

	SpiChnPutC(ADC_MOTOR_SPI, sendByte2);
	recvByte2 = SpiChnGetC(ADC_MOTOR_SPI);

	while(!SpiChnTxBuffEmpty(ADC_MOTOR_SPI));

	recvVal = recvByte1;
	recvVal = recvVal << 8;
	recvVal |= recvByte2;

	setPin(ADC_SE_CS_PIN, HIGH);

	return recvVal;
}

void commonADCReadAllSe(unsigned char channelEnd, unsigned short int readData[8]) {
	unsigned char i;
	for(i = 0; i <= channelEnd; i++) {
		readData[i] = commonADCReadSe(channelEnd);
	}
}

unsigned short int commonADCReadCurr(unsigned char channelEnd) {
	BYTE sendByte1 = 0b01000011;
	sendByte1 |= (channelEnd << 2);
```

```
	BYTE sendByte2 = 0b10010000;
	BYTE recvByte1 = 0;
	BYTE recvByte2 = 0;
	unsigned short int recvVal = 0;

	setPin(ADC_CURR_CS_PIN, LOW);

	SpiChnPutC(ADC_MOTOR_SPI, sendByte1);
	recvByte1 = SpiChnGetC(ADC_MOTOR_SPI);

	SpiChnPutC(ADC_MOTOR_SPI, sendByte2);
	recvByte2 = SpiChnGetC(ADC_MOTOR_SPI);

	while(!SpiChnTxBuffEmpty(ADC_MOTOR_SPI));

	recvVal = recvByte1;
	recvVal = recvVal << 8;
	recvVal |= recvByte2;

	setPin(ADC_CURR_CS_PIN, HIGH);

	return recvVal;
}

void commonADCReadAllCurr(unsigned char channelEnd, unsigned short int readData[8]) {
	unsigned char i;
	for(i = 0; i <= channelEnd; i++) {
		readData[i] = commonADCReadCurr(channelEnd);
	}
}
/***********************************/
/** END - Common ADC - Motor Board **/
/***********************************/
```

g. Common Configuration Header File

```
#ifndef COMMONCONFIG_H
#define COMMONCONFIG_H

#ifdef    __cplusplus
extern "C" {
#endif

#include <plib.h>


// Configuration Bit settings
// SYSCLK = 80 MHz (8MHz Crystal / FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 80 MHz (SYSCLK / FPBDIV)
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are don't care

/*****************************************************************/
/*  BEGIN - comment out when not building from project commonLibs  */
/*****************************************************************/
//#pragma config FPLLMUL  = MUL_20       // PLL Multiplier
//#pragma config UPLLIDIV = DIV_2        // USB PLL Input Divider
//#pragma config FPLLIDIV = DIV_2        // PLL Input Divider
//#pragma config FPLLODIV = DIV_1        // PLL Output Divider
//#pragma config FPBDIV   = DIV_1        // Peripheral Clock divisor
//#pragma config FWDTEN   = OFF          // Watchdog Timer
//#pragma config WDTPS    = PS1          // Watchdog Timer Postscale
//#pragma config FCKSM    = CSDCMD       // Clock Switching & Fail Safe Clock Monitor
//#pragma config OSCIOFNC = OFF          // CLKO Enable
//#pragma config POSCMOD  = HS           // Primary Oscillator
//#pragma config IESO     = ON           // Internal/External Switch-over
//#pragma config FSOSCEN  = ON           // Secondary Oscillator Enable (KLO was off)
```

```
//#pragma config FNOSC    = PRIPLL       // Oscillator Selection
//#pragma config CP       = OFF          // Code Protect
//#pragma config BWP      = OFF          // Boot Flash Write Protect
//#pragma config PWP      = OFF          // Program Flash Write Protect
//#pragma config ICESEL   = ICS_PGx2     // ICE/ICD Comm Channel Select
//#pragma config DEBUG    = OFF          // Background Debugger Enable
/*****************************************************************/
/*  END - comment out when not building from project commonLibs    */
/*****************************************************************/

//#define       GetPeripheralClock()            (SYS_FREQ/(1 << OSCCONbits.PBDIV))
//#define       GetInstructionClock()           (SYS_FREQ)

#define SYS_FREQ (80000000L)


#define THUMB_ID        0
#define INDEX_ID        1
#define MIDDLE_ID       2
#define RING_ID                 3
#define LITTLE_ID       4
#define WRIST_ID        5
#define ELBOW_ID        6

#define BOOLEAN unsigned char
#define TRUE 1
#define FALSE 0

#define OUTPUT                  0
#define INPUT           1

#define HIGH            1
#define LOW                     0
```

```
#define SENSOR_GROUPS          6
#define MOTOR_GROUPS           7

#define pulsePin(pin)   pin=HIGH;pin=LOW;
#define setPin(pin, value)  pin=value;

//#define MAIN_SENSOR_UART             UART2
//#define MAIN_SENSOR_UART_VECTOR                    _UART_2_VECTOR

#ifdef    __cplusplus
}
#endif

#endif   /* COMMONCONFIG_H */
```

h. Common Configuration Source File

```
#ifndef COMMONLIB_H
#define COMMONLIB_H

#ifdef    __cplusplus
extern "C" {
#endif


#include "commonConfig.h"



#ifdef    __cplusplus
}
#endif

#endif   /* COMMONLIB_H */
```

i. Common Structures Header File

```
/** @file commonStructs.h
   @brief     Common Structures
   @details   Structures used by more than one microcontroller
   @author    Jason Klein - jason.e.klein@wpi.edu
   @date      2013-2014
 */

#ifndef COMMONSTRUCTS_H
#define COMMONSTRUCTS_H

#ifdef   __cplusplus
extern "C" {
#endif

#include "commonLib.h"

   // used by commonTimer
   typedef struct _ExecTimer {
      unsigned long timerSave;
      unsigned long every;
   } ExecTimer;

   // used by commonUART
   typedef struct _RecvManager {
      unsigned long firstByteTime;
      unsigned long timout;
      unsigned long timeoutTime;
      BOOL overflow;
      unsigned char bytesExpected;
      unsigned char bytesReceived;
      BOOL transferComplete;
   } RecvManager;
```

```
#define CMP_OPT_STR_SUB      0
#define CMP_OPT_STP_SUB      1
#define CMP_OPT_STR_CMD      2
#define CMP_OPT_STP_CMD      3

#define CMP_CT_POS_ENC       0
#define CMP_CT_POS_FLX       1

#define COMPUTER_TX_MAIN_RX_PACKET_SIZE 4

        typedef struct __attribute__((__packed__)) _ComputerTxMainRxHeader {
                unsigned _id:4;
                unsigned _opt:2;          // subscribe, unsubscribe, command, uncommand
                unsigned _ctrlType:4;
                unsigned _col:2;
        } ComputerTxMainRxHeader;

        typedef union _ComputerTxMainRxPacket {
                unsigned char stream[COMPUTER_TX_MAIN_RX_PACKET_SIZE];

                struct {
                        ComputerTxMainRxHeader       header;
                        short int encoderPos;
                } use;
        } ComputerTxMainRxPacket;


#define MAIN_TX_COMPUTER_RX_PACKET_SIZE 5

        typedef struct __attribute__((__packed__)) _MainTxComputerRxHeader {
                unsigned _id:4;
                unsigned _aux:4;
        } MainTxComputerRxHeader;
```

```
        typedef union _MainTxComputerRxPacket {
                unsigned char stream[MAIN_TX_COMPUTER_RX_PACKET_SIZE];

                struct {
                        MainTxComputerRxHeader header;
                        unsigned short int flexADC;
                        unsigned short int presADC;
                } use;
        } MainTxComputerRxPacket;


//#define MTR_OPT_START             0
//#define MTR_OPT_STOP              1
//#define MTR_OPT_SUBSCRIBE2
//#define MTR_OPT_UNSUBSCRIBE       3

#define MTR_OPT_START             4
#define MTR_OPT_STOP              0
#define MTR_OPT_PUB               1
#define MTR_OPT_CTRL              2

#define MTR_CTRL_FLEX_POS    0
#define MTR_CTRL_ENC_POS     1


#define MAIN_TX_MOTOR_RX_PACKET_SIZE      10

/***********************************************/
/*- Main Board --> Motor Board: Header & Packet -*/
/***********************************************/
   typedef struct __attribute__((__packed__)) _MainTxMotorRxHeader {
      unsigned _id : 4; // #define XXXXX_ID
      unsigned _opt : 3; //(start/update)/stop/remove/subscribe?/unsubscribe?
      unsigned _controlType : 4;
```

```
    unsigned _aux : 5;
  } MainTxMotorRxHeader;

  typedef union _MainTxMotorRxPacket {
    unsigned char stream[MAIN_TX_MOTOR_RX_PACKET_SIZE];

    struct {
      MainTxMotorRxHeader header;
      short int encoderPos;
      short int encoderVel;
      unsigned short int currentAdc;
      unsigned short int seriesElasticAdc;
    } use;
  } MainTxMotorRxPacket;



#define MOTOR_TX_MAIN_RX_PACKET_SIZE     9

/***********************************************/
/*- Motor Board --> Main Board: Header & Packet -*/
/***********************************************/
  typedef struct __attribute__((__packed__)) _MotorTxMainRxHeader {
    unsigned _id : 4; ///< Subset number
    unsigned _opt : 3; // error, update,
    unsigned _aux : 1;
  } MotorTxMainRxHeader;

  typedef union _MotorTxMainRxPacket {
    unsigned char stream[MOTOR_TX_MAIN_RX_PACKET_SIZE];

    struct {
      MotorTxMainRxHeader header;
      short int encoderPos;
```

```
        short int encoderVel;
        unsigned short int currentAdc;
        unsigned short int seriesElasticAdc;
      } use;
   } MotorTxMainRxPacket;



#define SEN_OPT_START                   0
#define SEN_OPT_STOP            1
#define SEN_OPT_SAMPLE_REQ2
#define SEN_OPT_UPDATE                  3

#define SEN_LED_OFF                     0
#define SEN_LED_RED                     1
#define SEN_LED_BLUE            2
#define SEN_LED_GREEN                   3

#define MAIN_TX_SENSOR_RX_PACKET_SIZE     1

/*************************************************/
/*- Main Board --> Sensor Board: Header & Packet -*/
/*************************************************/
   typedef struct __attribute__((__packed__)) _MainTxSensorRxHeader {
      unsigned _id    : 4;   // 0b00 = thumb, ...
      unsigned _opt         : 2;        // 0b00 = start, 0b01 = stop, 0b10 = update all, 0b11 = update individual
                     unsigned _col    : 2;        // 0b00 = off, 0b01 = red, 0b10 = blue, 0b11 = green
   } MainTxSensorRxHeader;

   typedef union _MainTxSensorRxPacket {
      unsigned char stream[MAIN_TX_SENSOR_RX_PACKET_SIZE];

      struct {
         MainTxSensorRxHeader header;
```

```
    } use;
  } MainTxSensorRxPacket;



#define SENSOR_TX_MAIN_RX_PACKET_SIZE     5

/************************************************/
/*- Sensor Board --> Main Board: Header & Packet -*/
/************************************************/
  typedef struct __attribute__((__packed__)) _SensorTxMainRxHeader {
    unsigned _id    :4;
    unsigned _aux   :4;
  } SensorTxMainRxHeader;

  typedef union __attribute__((__packed__)) _SensorTxMainRxPacket {
    unsigned char stream[SENSOR_TX_MAIN_RX_PACKET_SIZE];

    struct __attribute__((__packed__)) {
      SensorTxMainRxHeader header;
      unsigned short int flexAdc;
      unsigned short int presAdc;
    } use;
  } SensorTxMainRxPacket;

#ifdef   __cplusplus
}
#endif

#endif   /* COMMONSTRUCTS_H */
```

K. User Interface Software

a. Exoarm User Interface Java File

```
package exoarmui;

import com.neuronrobotics.sdk.common.BowlerMethod;
import com.neuronrobotics.sdk.ui.ConnectionDialog;
import pic32.Pic32USB;



public class ExoarmUI extends javax.swing.JFrame {

   static Pic32USB p;
   Thread t;
   /** Creates new form Antenna */
   public ExoarmUI() {
          p = new Pic32USB();
          t = new Thread(p);
          boolean device = ConnectionDialog.getBowlerDevice(p);
          System.out.println(device);



          if(device) {

             initComponents();

             p.setJTextField(0, jTextField7);
             p.setJTextField(1, jTextField10);
             p.setJTextField(2, jTextField3);
             p.setJTextField(3, jTextField4);
             p.setJTextField(4, jTextField13);
```

```
            p.setJTextField(5, jTextField14);
            p.setJTextField(6, jTextField5);
            p.setJTextField(7, jTextField6);
            p.setJTextField(8, jTextField11);
            p.setJTextField(9, jTextField12);
        }
}

/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
// <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-BEGIN:initComponents
private void initComponents() {

    jPanel2 = new javax.swing.JPanel();
    jTextField3 = new javax.swing.JTextField();
    jTextField4 = new javax.swing.JTextField();
    jTextField5 = new javax.swing.JTextField();
    jTextField6 = new javax.swing.JTextField();
    jTextField7 = new javax.swing.JTextField();
    jLabel12 = new javax.swing.JLabel();
    jLabel13 = new javax.swing.JLabel();
    jLabel14 = new javax.swing.JLabel();
    jTextField10 = new javax.swing.JTextField();
    jLabel15 = new javax.swing.JLabel();
    jCheckBox3 = new javax.swing.JCheckBox();
    jCheckBox2 = new javax.swing.JCheckBox();
    jCheckBox4 = new javax.swing.JCheckBox();
    jCheckBox5 = new javax.swing.JCheckBox();
    jCheckBox6 = new javax.swing.JCheckBox();
    jTextField11 = new javax.swing.JTextField();
    jTextField12 = new javax.swing.JTextField();
```

```
jTextField13 = new javax.swing.JTextField();
jTextField14 = new javax.swing.JTextField();
jButton4 = new javax.swing.JButton();

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
setTitle("Exoarm");

jPanel2.setBorder(javax.swing.BorderFactory.createTitledBorder("Flex & Pressure Sensors"));

jTextField5.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(java.awt.event.ActionEvent evt) {
      jTextField5ActionPerformed(evt);
   }
});

jTextField7.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(java.awt.event.ActionEvent evt) {
      jTextField7ActionPerformed(evt);
   }
});

jLabel12.setText("Fingers");

jLabel13.setText("Flex");

jLabel14.setText("Pressure");

jCheckBox3.setText("Thumb:");
jCheckBox3.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(java.awt.event.ActionEvent evt) {
      jCheckBox3ActionPerformed(evt);
   }
});
```

```
jCheckBox2.setText("Index:");
jCheckBox2.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(java.awt.event.ActionEvent evt) {
      jCheckBox2ActionPerformed(evt);
   }
});

jCheckBox4.setText("Middle:");
jCheckBox4.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(java.awt.event.ActionEvent evt) {
      jCheckBox4ActionPerformed(evt);
   }
});

jCheckBox5.setText("Ring:");
jCheckBox5.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(java.awt.event.ActionEvent evt) {
      jCheckBox5ActionPerformed(evt);
   }
});

jCheckBox6.setText("Little:");
jCheckBox6.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(java.awt.event.ActionEvent evt) {
      jCheckBox6ActionPerformed(evt);
   }
});

jTextField11.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(java.awt.event.ActionEvent evt) {
      jTextField11ActionPerformed(evt);
   }
});
```

```
    org.jdesktop.layout.GroupLayout jPanel2Layout = new org.jdesktop.layout.GroupLayout(jPanel2);
    jPanel2.setLayout(jPanel2Layout);
    jPanel2Layout.setHorizontalGroup(
       jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
       .add(jPanel2Layout.createSequentialGroup()
          .add(56, 56, 56)
          .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.TRAILING)
             .add(jLabel12)
             .add(jCheckBox3)
             .add(jCheckBox2)
             .add(jCheckBox4)
             .add(jCheckBox5)
             .add(jCheckBox6))
          .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
          .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
             .add(jPanel2Layout.createSequentialGroup()
                .add(17, 17, 17)
                .add(jLabel13)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
             .add(jPanel2Layout.createSequentialGroup()
                .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
                   .add(org.jdesktop.layout.GroupLayout.TRAILING, jPanel2Layout.createSequentialGroup()
                      .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING, false)
                         .add(jTextField3)
                         .add(jTextField7, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, 86, Short.MAX_VALUE))
                      .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                      .add(jLabel15))
                   .add(jPanel2Layout.createSequentialGroup()
                      .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
                         .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.TRAILING, false)
                            .add(org.jdesktop.layout.GroupLayout.LEADING, jTextField5, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, 85,
Short.MAX_VALUE)
```

```
                        .add(org.jdesktop.layout.GroupLayout.LEADING, jTextField11, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 1,
Short.MAX_VALUE))
                    .add(jTextField13, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 85,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
                .add(0, 22, Short.MAX_VALUE)))
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)))
        .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.TRAILING, false)
            .add(org.jdesktop.layout.GroupLayout.LEADING, jTextField6)
            .add(org.jdesktop.layout.GroupLayout.LEADING, jTextField4)
            .add(org.jdesktop.layout.GroupLayout.LEADING, jPanel2Layout.createSequentialGroup()
                .add(26, 26, 26)
                .add(jLabel14))
            .add(org.jdesktop.layout.GroupLayout.LEADING, jTextField14)
            .add(jTextField12)
            .add(jTextField10, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 99, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
        .addContainerGap(21, Short.MAX_VALUE))
    );
    jPanel2Layout.setVerticalGroup(
        jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
        .add(jPanel2Layout.createSequentialGroup()
            .addContainerGap()
            .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.BASELINE)
                .add(jLabel12)
                .add(jLabel13)
                .add(jLabel14))
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
            .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
                .add(org.jdesktop.layout.GroupLayout.TRAILING, jTextField7, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 28,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.BASELINE)
                    .add(jTextField10, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                    .add(jLabel15)
                    .add(jCheckBox3)))
```

```
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
            .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.BASELINE)
               .add(jTextField3, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
               .add(jTextField4, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
               .add(jCheckBox2))
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED, 9, Short.MAX_VALUE)
            .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.BASELINE)
               .add(jCheckBox4)
               .add(jTextField13, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
               .add(jTextField14, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
            .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.BASELINE)
               .add(jTextField5, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
               .add(jTextField6, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
               .add(jCheckBox5))
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
            .add(jPanel2Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.BASELINE)
               .add(jCheckBox6)
               .add(jTextField11, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
               .add(jTextField12, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .addContainerGap(7, Short.MAX_VALUE))
      );

      jButton4.setText("Begin Sampling");
      jButton4.addActionListener(new java.awt.event.ActionListener() {
         public void actionPerformed(java.awt.event.ActionEvent evt) {
```

```
            jButton4ActionPerformed(evt);
        }
    });

    org.jdesktop.layout.GroupLayout layout = new org.jdesktop.layout.GroupLayout(getContentPane());
    getContentPane().setLayout(layout);
    layout.setHorizontalGroup(
        layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
        .add(layout.createSequentialGroup()
            .addContainerGap()
            .add(layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
                .add(jPanel2, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                .add(org.jdesktop.layout.GroupLayout.TRAILING, layout.createSequentialGroup()
                    .add(0, 0, Short.MAX_VALUE)
                    .add(jButton4)))
            .addContainerGap())
    );
    layout.setVerticalGroup(
        layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
        .add(layout.createSequentialGroup()
            .addContainerGap()
            .add(jPanel2, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,
org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .add(jButton4)
            .addContainerGap())
    );

    pack();
  }// </editor-fold>//GEN-END:initComponents

  private void jCheckBox6ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jCheckBox6ActionPerformed
    // INDEX
        if(jCheckBox6.isSelected()) {
```

```
            p.subscribe(4);
        } else {
            p.unsubscribe(4);
        }
}//GEN-LAST:event_jCheckBox6ActionPerformed

private void jCheckBox4ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jCheckBox4ActionPerformed
   // MIDDLE
        if(jCheckBox4.isSelected()) {
            p.subscribe(2);
        } else {
            p.unsubscribe(2);
        }
}//GEN-LAST:event_jCheckBox4ActionPerformed

private void jCheckBox2ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jCheckBox2ActionPerformed
        // INDEX
        if(jCheckBox2.isSelected()) {
            p.subscribe(1);
        } else {
            p.unsubscribe(1);
        }
}//GEN-LAST:event_jCheckBox2ActionPerformed

private void jCheckBox3ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jCheckBox3ActionPerformed
        // THUMB
        if(jCheckBox3.isSelected()) {
            p.subscribe(0);
        } else {
            p.unsubscribe(0);
        }
}//GEN-LAST:event_jCheckBox3ActionPerformed

private void jTextField7ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jTextField7ActionPerformed
```

```
        // TODO add your handling code here:
    }//GEN-LAST:event_jTextField7ActionPerformed

    private void jTextField5ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jTextField5ActionPerformed

    }//GEN-LAST:event_jTextField5ActionPerformed

    private void jTextField11ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jTextField11ActionPerformed
        // TODO add your handling code here:
    }//GEN-LAST:event_jTextField11ActionPerformed

    private void jCheckBox5ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jCheckBox5ActionPerformed
        // RING
            if(jCheckBox5.isSelected()) {
                p.subscribe(3);
            } else {
                p.unsubscribe(3);
            }
    }//GEN-LAST:event_jCheckBox5ActionPerformed

    private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jButton4ActionPerformed
        // TODO add your handling code here:
//          int data[] = p.readSensorBoard();
//          jTextField7.setText(data[1] + "");


            t.start();
    }//GEN-LAST:event_jButton4ActionPerformed

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        /* Set the Nimbus look and feel */
```

```
        //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
        /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
         * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
         */
        try {
            javax.swing.UIManager.LookAndFeelInfo[] installedLookAndFeels=javax.swing.UIManager.getInstalledLookAndFeels();
            for (int idx=0; idx<installedLookAndFeels.length; idx++)
                if ("Nimbus".equals(installedLookAndFeels[idx].getName())) {
                    javax.swing.UIManager.setLookAndFeel(installedLookAndFeels[idx].getClassName());
                    break;
                }
        } catch (ClassNotFoundException ex) {
            java.util.logging.Logger.getLogger(ExoarmUI.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
        } catch (InstantiationException ex) {
            java.util.logging.Logger.getLogger(ExoarmUI.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
        } catch (IllegalAccessException ex) {
            java.util.logging.Logger.getLogger(ExoarmUI.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
        } catch (javax.swing.UnsupportedLookAndFeelException ex) {
            java.util.logging.Logger.getLogger(ExoarmUI.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
        }
        //</editor-fold>

//        Pic32USB p = new Pic32USB();
//        boolean device = ConnectionDialog.getBowlerDevice(p);
//        System.out.println(device);
//        if(device) {
//            initComponents();
//            ExoarmUI e = new ExoarmUI();
//            e.setVisible(true);
//        }


        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
```

```
        public void run() {
            new ExoarmUI().setVisible(true);
        }
      });
    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JButton jButton4;
    private javax.swing.JCheckBox jCheckBox2;
    private javax.swing.JCheckBox jCheckBox3;
    private javax.swing.JCheckBox jCheckBox4;
    private javax.swing.JCheckBox jCheckBox5;
    private javax.swing.JCheckBox jCheckBox6;
    private javax.swing.JLabel jLabel12;
    private javax.swing.JLabel jLabel13;
    private javax.swing.JLabel jLabel14;
    private javax.swing.JLabel jLabel15;
    private javax.swing.JPanel jPanel2;
    private javax.swing.JTextField jTextField10;
    private javax.swing.JTextField jTextField11;
    private javax.swing.JTextField jTextField12;
    private javax.swing.JTextField jTextField13;
    private javax.swing.JTextField jTextField14;
    private javax.swing.JTextField jTextField3;
    private javax.swing.JTextField jTextField4;
    private javax.swing.JTextField jTextField5;
    private javax.swing.JTextField jTextField6;
    private javax.swing.JTextField jTextField7;
    // End of variables declaration//GEN-END:variables

    private Runnable Pic32USB() {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.
    }
}
```

b. Exoarm User Interface Form File

```
<?xml version="1.0" encoding="UTF-8" ?>

<Form version="1.3" maxVersion="1.5" type="org.netbeans.modules.form.forminfo.JFrameFormInfo">
 <Properties>
  <Property name="defaultCloseOperation" type="int" value="3"/>
  <Property name="title" type="java.lang.String" value="Exoarm"/>
 </Properties>
 <SyntheticProperties>
  <SyntheticProperty name="formSizePolicy" type="int" value="1"/>
  <SyntheticProperty name="generateCenter" type="boolean" value="false"/>
 </SyntheticProperties>
 <AuxValues>
  <AuxValue name="FormSettings_autoResourcing" type="java.lang.Integer" value="0"/>
  <AuxValue name="FormSettings_autoSetComponentName" type="java.lang.Boolean" value="false"/>
  <AuxValue name="FormSettings_generateFQN" type="java.lang.Boolean" value="true"/>
  <AuxValue name="FormSettings_generateMnemonicsCode" type="java.lang.Boolean" value="false"/>
  <AuxValue name="FormSettings_i18nAutoMode" type="java.lang.Boolean" value="false"/>
  <AuxValue name="FormSettings_layoutCodeTarget" type="java.lang.Integer" value="2"/>
  <AuxValue name="FormSettings_listenerGenerationStyle" type="java.lang.Integer" value="0"/>
  <AuxValue name="FormSettings_variablesLocal" type="java.lang.Boolean" value="false"/>
  <AuxValue name="FormSettings_variablesModifier" type="java.lang.Integer" value="2"/>
 </AuxValues>

 <Layout>
  <DimensionLayout dim="0">
   <Group type="103" groupAlignment="0" attributes="0">
     <Group type="102" attributes="0">
       <EmptySpace max="-2" attributes="0"/>
       <Group type="103" groupAlignment="0" attributes="0">
         <Component id="jPanel2" alignment="0" max="32767" attributes="0"/>
         <Group type="102" alignment="1" attributes="0">
           <EmptySpace min="0" pref="0" max="32767" attributes="0"/>
```

```
              <Component id="jButton4" min="-2" max="-2" attributes="0"/>
            </Group>
          </Group>
          <EmptySpace max="-2" attributes="0"/>
        </Group>
      </Group>
    </DimensionLayout>
    <DimensionLayout dim="1">
      <Group type="103" groupAlignment="0" attributes="0">
        <Group type="102" alignment="0" attributes="0">
          <EmptySpace min="-2" max="-2" attributes="0"/>
          <Component id="jPanel2" min="-2" max="-2" attributes="0"/>
          <EmptySpace max="32767" attributes="0"/>
          <Component id="jButton4" min="-2" max="-2" attributes="0"/>
          <EmptySpace max="-2" attributes="0"/>
        </Group>
      </Group>
    </DimensionLayout>
  </Layout>
  <SubComponents>
    <Container class="javax.swing.JPanel" name="jPanel2">
      <Properties>
        <Property name="border" type="javax.swing.border.Border" editor="org.netbeans.modules.form.editors2.BorderEditor">
          <Border info="org.netbeans.modules.form.compat2.border.TitledBorderInfo">
            <TitledBorder title="Flex &amp;amp; Pressure Sensors"/>
          </Border>
        </Property>
      </Properties>

      <Layout>
        <DimensionLayout dim="0">
          <Group type="103" groupAlignment="0" attributes="0">
            <Group type="102" alignment="0" attributes="0">
              <EmptySpace min="-2" pref="56" max="-2" attributes="0"/>
```

```
<Group type="103" groupAlignment="1" attributes="0">
    <Component id="jLabel12" min="-2" max="-2" attributes="0"/>
    <Component id="jCheckBox3" min="-2" max="-2" attributes="0"/>
    <Component id="jCheckBox2" min="-2" max="-2" attributes="0"/>
    <Component id="jCheckBox4" min="-2" max="-2" attributes="0"/>
    <Component id="jCheckBox5" min="-2" max="-2" attributes="0"/>
    <Component id="jCheckBox6" min="-2" max="-2" attributes="0"/>
</Group>
<EmptySpace max="-2" attributes="0"/>
<Group type="103" groupAlignment="0" attributes="0">
    <Group type="102" alignment="0" attributes="0">
        <EmptySpace min="-2" pref="17" max="-2" attributes="0"/>
        <Component id="jLabel13" min="-2" max="-2" attributes="0"/>
        <EmptySpace min="66" max="32767" attributes="0"/>
    </Group>
    <Group type="102" attributes="0">
        <Group type="103" groupAlignment="0" attributes="0">
            <Group type="102" alignment="1" attributes="0">
                <Group type="103" groupAlignment="0" max="-2" attributes="0">
                    <Component id="jTextField3" max="32767" attributes="0"/>
                    <Component id="jTextField7" pref="86" max="32767" attributes="0"/>
                </Group>
                <EmptySpace max="32767" attributes="0"/>
                <Component id="jLabel15" min="-2" max="-2" attributes="0"/>
            </Group>
            <Group type="102" attributes="0">
                <Group type="103" groupAlignment="0" attributes="0">
                    <Group type="103" groupAlignment="1" max="-2" attributes="0">
                        <Component id="jTextField5" alignment="0" pref="85" max="32767" attributes="0"/>
                        <Component id="jTextField11" alignment="0" pref="0" max="32767" attributes="0"/>
                    </Group>
                    <Component id="jTextField13" alignment="0" min="-2" pref="85" max="-2" attributes="0"/>
                </Group>
                <EmptySpace min="0" pref="22" max="32767" attributes="0"/>
```

```
              </Group>
            </Group>
            <EmptySpace max="-2" attributes="0"/>
          </Group>
        </Group>
        <Group type="103" groupAlignment="1" max="-2" attributes="0">
          <Component id="jTextField6" alignment="0" max="32767" attributes="0"/>
          <Component id="jTextField4" alignment="0" max="32767" attributes="0"/>
          <Group type="102" alignment="0" attributes="0">
            <EmptySpace min="-2" pref="26" max="-2" attributes="0"/>
            <Component id="jLabel14" min="-2" max="-2" attributes="0"/>
          </Group>
          <Component id="jTextField14" alignment="0" max="32767" attributes="0"/>
          <Component id="jTextField12" max="32767" attributes="0"/>
          <Component id="jTextField10" alignment="1" min="-2" pref="99" max="-2" attributes="0"/>
        </Group>
        <EmptySpace pref="21" max="32767" attributes="0"/>
      </Group>
    </Group>
  </DimensionLayout>
  <DimensionLayout dim="1">
    <Group type="103" groupAlignment="0" attributes="0">
      <Group type="102" alignment="0" attributes="0">
        <EmptySpace max="-2" attributes="0"/>
        <Group type="103" groupAlignment="3" attributes="0">
          <Component id="jLabel12" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jLabel13" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jLabel14" alignment="3" min="-2" max="-2" attributes="0"/>
        </Group>
        <EmptySpace max="-2" attributes="0"/>
        <Group type="103" groupAlignment="0" attributes="0">
          <Component id="jTextField7" alignment="1" min="-2" pref="28" max="-2" attributes="0"/>
          <Group type="103" groupAlignment="3" attributes="0">
            <Component id="jTextField10" alignment="3" min="-2" max="-2" attributes="0"/>
```

```
            <Component id="jLabel15" alignment="3" min="-2" max="-2" attributes="0"/>
            <Component id="jCheckBox3" alignment="3" min="-2" max="-2" attributes="0"/>
          </Group>
        </Group>
        <EmptySpace max="-2" attributes="0"/>
        <Group type="103" groupAlignment="3" attributes="0">
          <Component id="jTextField3" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jTextField4" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jCheckBox2" alignment="3" min="-2" max="-2" attributes="0"/>
        </Group>
        <EmptySpace pref="9" max="32767" attributes="0"/>
        <Group type="103" groupAlignment="3" attributes="0">
          <Component id="jCheckBox4" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jTextField13" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jTextField14" alignment="3" min="-2" max="-2" attributes="0"/>
        </Group>
        <EmptySpace max="-2" attributes="0"/>
        <Group type="103" groupAlignment="3" attributes="0">
          <Component id="jTextField5" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jTextField6" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jCheckBox5" alignment="3" min="-2" max="-2" attributes="0"/>
        </Group>
        <EmptySpace max="-2" attributes="0"/>
        <Group type="103" groupAlignment="3" attributes="0">
          <Component id="jCheckBox6" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jTextField11" alignment="3" min="-2" max="-2" attributes="0"/>
          <Component id="jTextField12" alignment="3" min="-2" max="-2" attributes="0"/>
        </Group>
        <EmptySpace pref="7" max="32767" attributes="0"/>
      </Group>
    </Group>
  </DimensionLayout>
</Layout>
<SubComponents>
```

```
      <Component class="javax.swing.JTextField" name="jTextField3">
      </Component>
      <Component class="javax.swing.JTextField" name="jTextField4">
      </Component>
      <Component class="javax.swing.JTextField" name="jTextField5">
        <Events>
          <EventHandler event="actionPerformed" listener="java.awt.event.ActionListener" parameters="java.awt.event.ActionEvent"
handler="jTextField5ActionPerformed"/>
        </Events>
      </Component>
      <Component class="javax.swing.JTextField" name="jTextField6">
      </Component>
      <Component class="javax.swing.JTextField" name="jTextField7">
        <Events>
          <EventHandler event="actionPerformed" listener="java.awt.event.ActionListener" parameters="java.awt.event.ActionEvent"
handler="jTextField7ActionPerformed"/>
        </Events>
      </Component>
      <Component class="javax.swing.JLabel" name="jLabel12">
        <Properties>
          <Property name="text" type="java.lang.String" value="Fingers"/>
        </Properties>
      </Component>
      <Component class="javax.swing.JLabel" name="jLabel13">
        <Properties>
          <Property name="text" type="java.lang.String" value="Flex"/>
        </Properties>
      </Component>
      <Component class="javax.swing.JLabel" name="jLabel14">
        <Properties>
          <Property name="text" type="java.lang.String" value="Pressure"/>
        </Properties>
      </Component>
      <Component class="javax.swing.JTextField" name="jTextField10">
```

```
    </Component>
    <Component class="javax.swing.JLabel" name="jLabel15">
    </Component>
    <Component class="javax.swing.JCheckBox" name="jCheckBox3">
      <Properties>
        <Property name="text" type="java.lang.String" value="Thumb:"/>
      </Properties>
      <Events>
        <EventHandler event="actionPerformed" listener="java.awt.event.ActionListener" parameters="java.awt.event.ActionEvent"
handler="jCheckBox3ActionPerformed"/>
      </Events>
    </Component>
    <Component class="javax.swing.JCheckBox" name="jCheckBox2">
      <Properties>
        <Property name="text" type="java.lang.String" value="Index:"/>
      </Properties>
      <Events>
        <EventHandler event="actionPerformed" listener="java.awt.event.ActionListener" parameters="java.awt.event.ActionEvent"
handler="jCheckBox2ActionPerformed"/>
      </Events>
    </Component>
    <Component class="javax.swing.JCheckBox" name="jCheckBox4">
      <Properties>
        <Property name="text" type="java.lang.String" value="Middle:"/>
      </Properties>
      <Events>
        <EventHandler event="actionPerformed" listener="java.awt.event.ActionListener" parameters="java.awt.event.ActionEvent"
handler="jCheckBox4ActionPerformed"/>
      </Events>
    </Component>
    <Component class="javax.swing.JCheckBox" name="jCheckBox5">
      <Properties>
        <Property name="text" type="java.lang.String" value="Ring:"/>
      </Properties>
```

```
        <Events>
          <EventHandler event="actionPerformed" listener="java.awt.event.ActionListener" parameters="java.awt.event.ActionEvent"
handler="jCheckBox5ActionPerformed"/>
        </Events>
      </Component>
      <Component class="javax.swing.JCheckBox" name="jCheckBox6">
        <Properties>
          <Property name="text" type="java.lang.String" value="Little:"/>
        </Properties>
        <Events>
          <EventHandler event="actionPerformed" listener="java.awt.event.ActionListener" parameters="java.awt.event.ActionEvent"
handler="jCheckBox6ActionPerformed"/>
        </Events>
      </Component>
      <Component class="javax.swing.JTextField" name="jTextField11">
        <Events>
          <EventHandler event="actionPerformed" listener="java.awt.event.ActionListener" parameters="java.awt.event.ActionEvent"
handler="jTextField11ActionPerformed"/>
        </Events>
      </Component>
      <Component class="javax.swing.JTextField" name="jTextField12">
      </Component>
      <Component class="javax.swing.JTextField" name="jTextField13">
      </Component>
      <Component class="javax.swing.JTextField" name="jTextField14">
      </Component>
    </SubComponents>
  </Container>
  <Component class="javax.swing.JButton" name="jButton4">
    <Properties>
      <Property name="text" type="java.lang.String" value="Begin Sampling"/>
    </Properties>
    <Events>
```

```
      <EventHandler event="actionPerformed" listener="java.awt.event.ActionListener" parameters="java.awt.event.ActionEvent"
handler="jButton4ActionPerformed"/>
     </Events>
   </Component>
  </SubComponents>
</Form>
```

c. PIC32 USB Java File

```
package pic32;

import com.neuronrobotics.sdk.common.BowlerAbstractDevice;
import com.neuronrobotics.sdk.common.BowlerDatagram;
import com.neuronrobotics.sdk.common.BowlerMethod;
import java.nio.ByteBuffer;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JTextField;

/**
 *
 * @author jasonklein
 */
public class Pic32USB extends BowlerAbstractDevice implements Runnable {

   JTextField textFields[];
   short sensorBoardData[];

   public Pic32USB() {
         sensorBoardData = new short[11];
         textFields = new JTextField[10];
   }

   public void setJTextField(int i, JTextField f) {
         this.textFields[i] = f;
   }

   public void subscribe(int id) {
         Object[] args = send("exo.arm.*",
```

```
                BowlerMethod.POST,
                "SenS",
                new Object[]{(short)id, 1}, 5);
    }

    public void unsubscribe(int id) {
        Object[] args = send("exo.arm.*",
                BowlerMethod.POST,
                "SenS",
                new Object[]{(short)id, 0}, 5);
    }

    private synchronized short[] readSensorBoard() {
        short[] data = new short[11];

        Object[] args = send("exo.arm.*",
                BowlerMethod.POST,
                "SenG",
                new Object[]{}, 5);

        System.out.println(args.length);

        for(int i = 0; i < 11; i++) {
//          data[i] = (Short)args[i];
//          data[i] = ByteBuffer.wrap(args[i]).getShort();
            data[i] = Short.parseShort(args[i] + "");
//          data[i] = (Integer)args[i];
        }

        return data;
    }
```

```
    @Override
    public void onAsyncResponse(BowlerDatagram data) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools |
Templates.
    }

    @Override
    public void run() {
        while(true) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {
                Logger.getLogger(Pic32USB.class.getName()).log(Level.SEVERE, null, ex);
            }
            sensorBoardData = readSensorBoard();
//          textFields[0].setText(sensorBoardData);
            for(int i = 0; i < 10; i++) {
                textFields[i].setText(sensorBoardData[i] + "");
            }
        }
    }
}
```

L. Flex Sensors Test Data

Degress	V1	V2	diff	Vout	gain
0	2.903	2.371	0.532	1.166	2.191729
5	2.93	2.371	0.559	1.82	3.255814
10	3.01	2.371	0.639	1.99	3.114241
15	3.06	2.371	0.689	2.177	3.159652
20	3.17	2.371	0.799	2.492	3.118899
25	3.24	2.371	0.869	2.701	3.10817
30	3.28	2.371	0.909	2.83	3.113311
35	3.33	2.371	0.959	2.976	3.103233
40	3.402	2.371	1.031	3.211	3.114452
45	3.447	2.371	1.076	3.342	3.105948
50	3.47	2.371	1.099	3.425	3.11647
60	3.555	2.371	1.184	3.698	3.123311
70	3.652	2.371	1.281	3.991	3.115535
80	3.67	2.371	1.299	4.06	3.125481
0	2.705	2.372	0.333	1.086	3.261261
5	2.73	2.372	0.358	1.117	3.120112
10	2.81	2.372	0.438	1.354	3.091324
15	2.868	2.372	0.496	1.525	3.074597
20	2.945	2.372	0.573	1.774	3.095986
25	3.013	2.372	0.641	1.992	3.107644
30	3.085	2.372	0.713	2.211	3.100982
35	3.119	2.372	0.747	2.311	3.093708
40	3.166	2.372	0.794	2.456	3.093199
45	3.218	2.372	0.846	2.616	3.092199
50	3.269	2.372	0.897	2.78	3.09922
60	3.38	2.372	1.008	3.152	3.126984
70	3.469	2.372	1.097	3.415	3.113036
80	3.557	2.372	1.185	3.703	3.124895

M. MATLAB Code for Testing Low-Pass Filter Transfer Function

```
%function Sallen_Key_2nd_LPF_Testing (R1, R2, C1, C2)
function Sallen_Key_2nd_LPF_Testing (R, C)

f = 0:0.1:1024;
w = 2*pi*f;
H = 1./(1+2*R*C*1j*w + (R^2)*(C^2)*((1j*w).^2));    % frequency response
% when we make R1=R2=R, C1=C2=C

plot(f,abs(H))
title('Sallen-Key 2nd LPF');
xlabel('Frequency (Hz)');
ylabel('Amplitude');
return
```