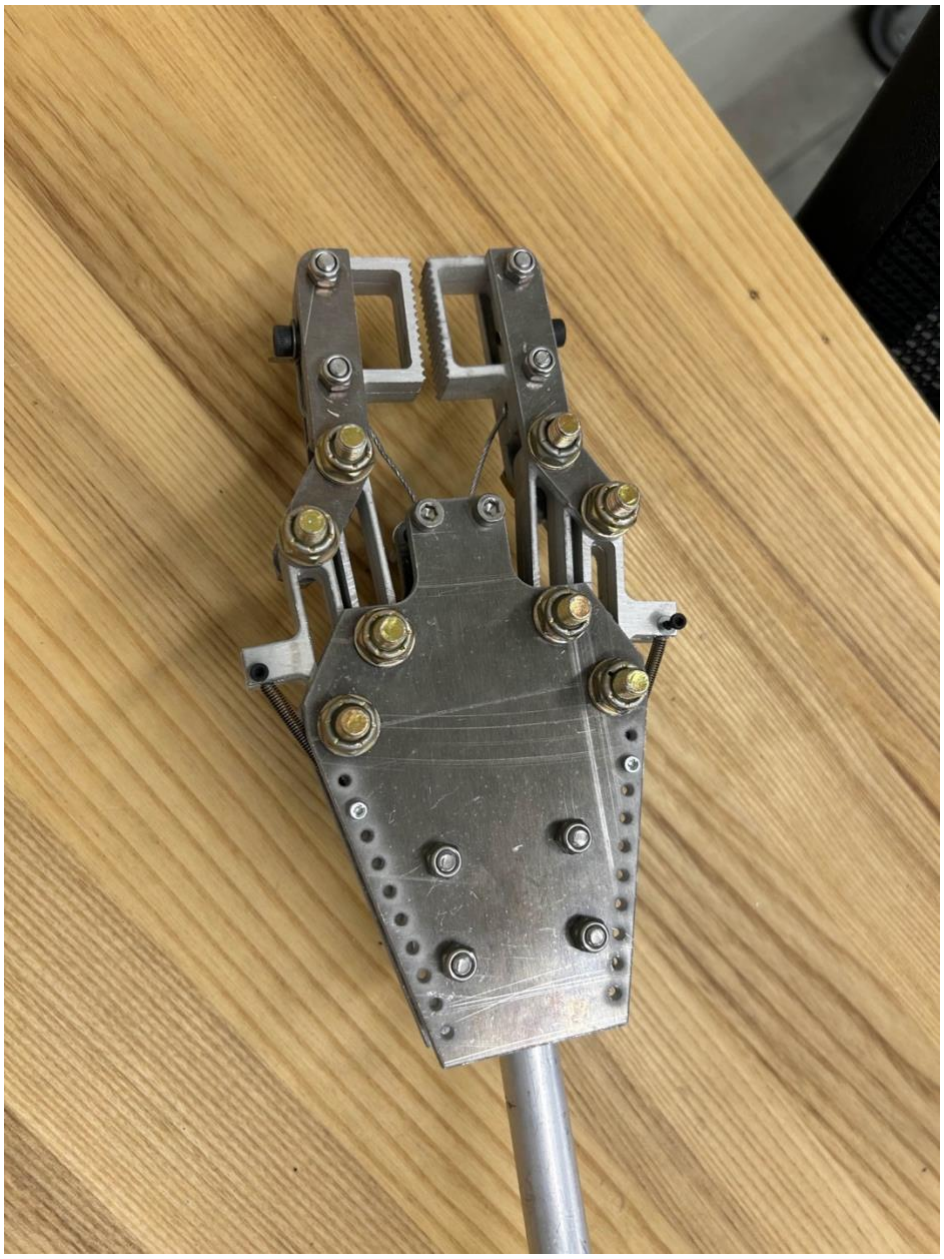
Force Control End Effector



James Quek: jamesqhg@gmail.com

Meet Nandu: meetnandu33@gmail.com

Abstract

This project focuses on the development of an adjustable force control end effector, designed to securely grip various objects while preventing damage or injury during operation. The primary objective was to implement closed-loop current control for precise force adjustments, enabling safe and effective interactions in robotic systems. The end effector features a robust cable-driven mechanical design with an ergonomic gripper, leveraging a linear actuator for force transmission. Built using aluminum plates and steel cables, the gripper ensures durability and consistent performance under heavy loads.

The firmware integrates a proportional-integrator (PI) control loop to dynamically regulate motor currents based on user inputs and feedback from a hall sensor. Analog signals from the force control knob and load cell were mapped into digital values using ADC channels, scaled to reflect real-world forces. Safety features, including a trigger switch and actuator feedback, ensure that the gripper operates within predefined limits to prevent mechanical damage during backtracking. A Python-based graphical user interface (GUI) displays real-time feedback on force, current, and load, allowing users to make precise adjustments.

Comprehensive testing demonstrated the system's capability to handle diverse scenarios, including grip control on fragile objects like a Coke can, lifting a 25-pound dumbbell, and safely holding a human finger without discomfort at low current limits. The results highlighted the importance of current control in achieving precise and safe operation. While the system successfully achieved its goals, challenges such as limited backdrivability in the mechanical setup were noted, with recommendations for future designs to improve actuator performance.

This project showcases the integration of mechanical design, firmware development, and user-friendly software for a safe and reliable force control system, contributing valuable insights for applications in human-robot interactions and adaptive gripping mechanisms.

1. Objectives

Design, build, and program an adjustable force control end effector for use by a person to reach and grab a wide variety of objects. Like a mechanical “Reacher-grabber tool” employed by people to access things on high shelves, this will be an electrically controlled grabber where the user can use a control knob to adjust the force applied to the object.

2. Rationale

The reason for attempting this kind of project is as an exercise to implement closed-loop current control as a means to control the torque or force applied by a robotic end effector. This is useful as robotic developments move towards implementing robots and robotic systems in consumer applications where human-robot interactions must be carefully considered to as to not harm human beings who interact with the robots. One of the easiest ways to limit robotic force application without force-sensors is by limiting current to the electric motors such that the force applied is not enough to harm a person or object.

3. Summary of Functional Requirements

This section summarizes the key functional requirements (FRs) for the adjustable force control end effector and the corresponding hardware and software developed to address each requirement. The interactions between system components and the flow of signals are outlined to provide a comprehensive understanding of the system functionality.

Functional Component	**Functional Requirement**	**Hardware/Software Addressed**
End Effector Mechanical Design	Design an ergonomic, robust mechanical gripper capable of adjusting grip force to securely hold objects without causing damage.	**Hardware**: Mechanically robust gripper with adjustable gripping mechanism.
End Effector Firmware	Develop firmware to implement current-based closed-loop force control, process user inputs, and transmit UART commands for calibration.	**Software**: Microcontroller firmware written in C implementing P control loop for current adjustment and UART communication.
End Effector Electrical Design	Provides stable and reliable electrical power to the motor, hall sensor, and microcontroller, ensuring smooth operation and accurate feedback signals.	**Hardware**: Power supply, motor driver circuits, and sensors for monitoring grip position and force.
End Effector Calibration	Accurately calibrate the relationship between motor current and applied	**Hardware**: Load cell for measuring force.

	force using a load cell for precise force adjustments.	**Software**: Calibration algorithm integrated into firmware to map current values to output forces, using ADC's
End Effector User Software	Create/ Use a user-friendly software interface for force adjustment, calibration to show load mapping	**Software**: C#/ Python based GUI for receiving feedback on operations.

Table 1: Summary of functional requirements

4. 1 Functional Component #1 – End Effector Mechanical Design

End effector mechanical design philosophy was to develop a robust, reliable, mechanical gripper structure that used cable-actuation and a high-reduction linear leadscrew actuator to mimic the kind of force transmission setup that would be present in a cable-driven robotic application. The gripper was expected to need to withstand forces from up to 500N peak and 300N continuous force from the Actuonix P16-P linear actuator and so ¼" and ½" AL 7075 plate was found and used for the structure and linkages. The linkage pivots were tight tolerance AN bolts (similar to SAE grade 8), and were tightened on with nylon locknuts to ensure vibration and load would not loosen the mechanism. The cable was a 1.5mm steel braided cable with a max load capacity of 1.4kN. The 4-bar linkages of the gripper were spring-loaded to ensure return of the gripper to a neutral position.

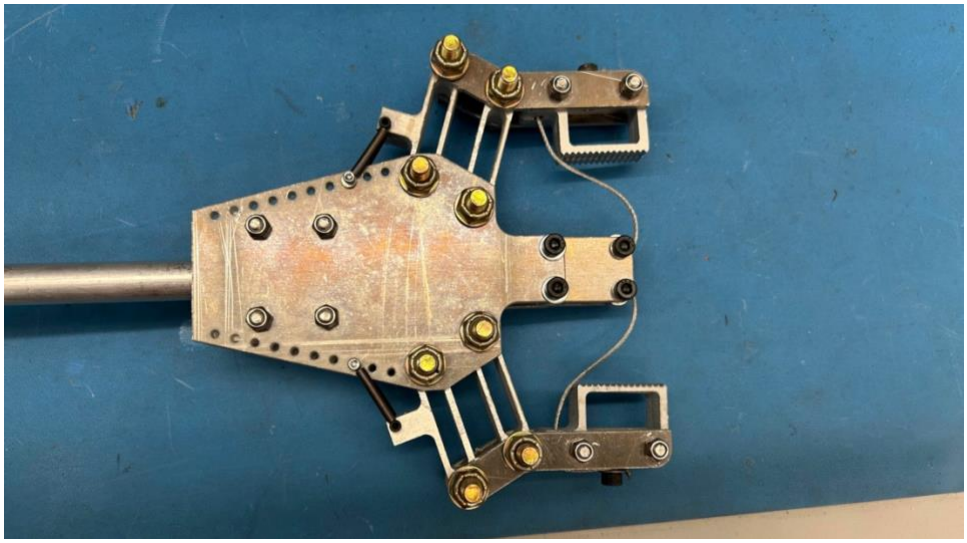


Figure 1: Design of the Gripper

4.1.1 Approach and Design

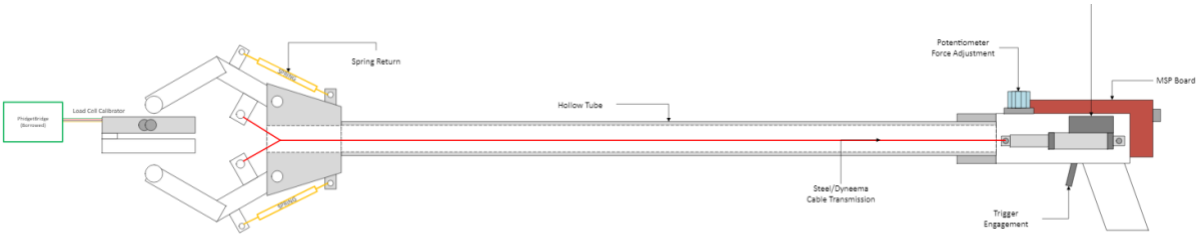
- **Objective**: Design a portable, user-friendly, robust end effector that can grab a variety of objects without causing damage.
- **Design Requirements**: The end effector must withstand regular and off-nominal use, with ergonomic controls for easy human use and durable materials to handle wear and tear.
- **Mechanism**: The mechanical design should allow for controlled, adjustable gripping force that is gentle enough for fragile objects while robust enough for heavier ones.

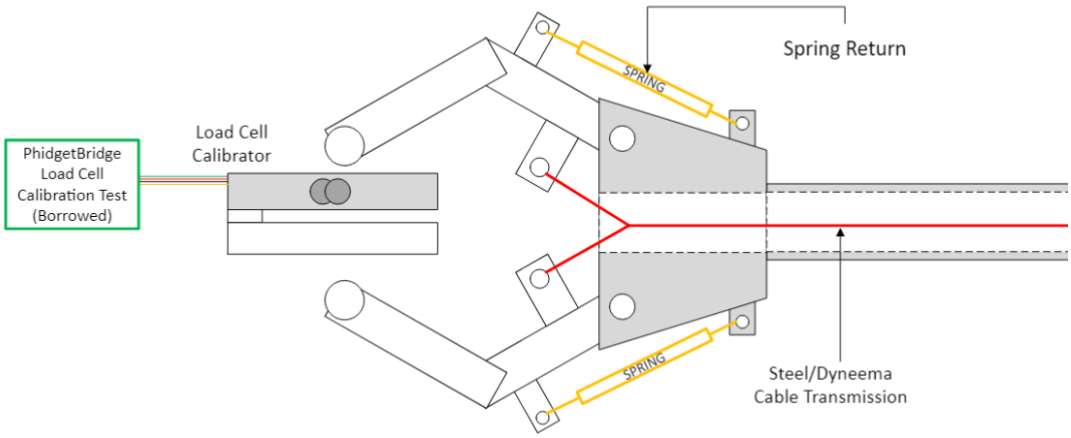
4.1.2 Inputs and Outputs

- **Inputs**:
    - Trigger switch to initiate gripping action
    - Gripper force setpoint adjustment knob
- **Outputs**:
    - Gripping action on the object with controlled force, closed-loop current control of the linear actuator, calibrated for equivalent force control

4.1.3 Parameters

- **Force Range**:
    - Able to grip with minimum of 2N and maximum of 150N
- **Durability Factors**:
    - AN/SAE Grade 8 bolts as linkage pivots
    - Al7075 ¼" and ½" plate construction for linkages and structure
    - 1.5mm steel wire rope for cable drive
- **Grip Shape**: Suitable to grasp multiple object shapes and sizes securely
    - Waterjet gripper texture to assist in gripping variety of surfaces
    - 0-100mm object grip width achievable





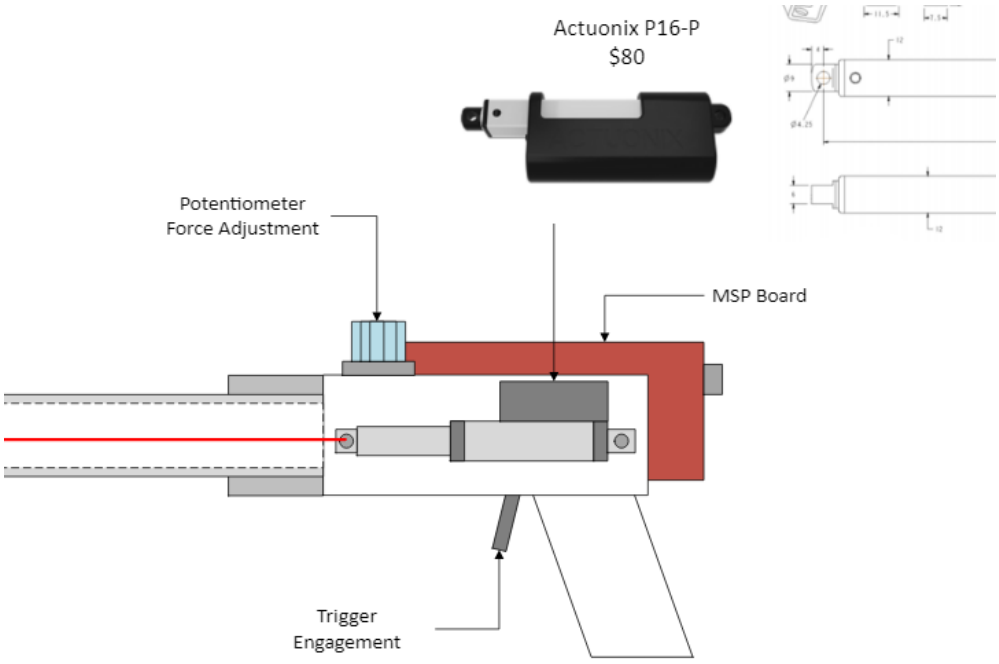


Figure 2,3,4: Mechanical Design

4.2 Functional Component #2 – End Effector Firmware

4.2.1 Approach and Design

The firmware for the end effector was successfully developed to control motor currents, process user inputs. The design achieved dynamic current adjustments based on user input while ensuring safe operation by imposing strict current limits. A closed loop P controller design was implemented to ensure a steady input (PI was to be implemented but did not get time to implement it fully.) Discretization and mapping techniques were employed to convert analog signals from sensors into actionable digital values, ensuring precision and scalability.

The figure below demonstrates the controls logic implemented:

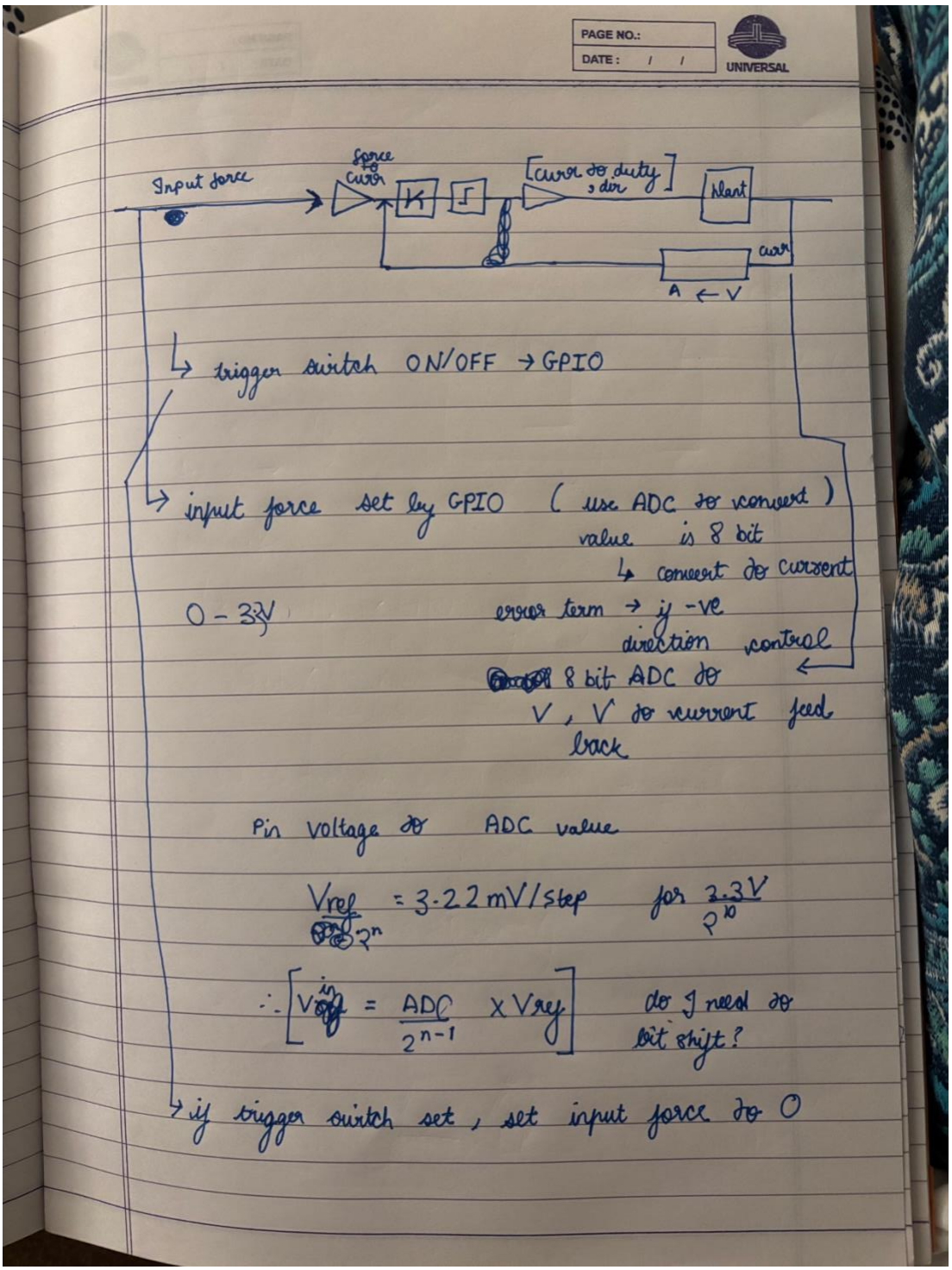


Figure 5: Controls loop for Current Sensing loop

4.2.2 Inputs and Outputs

- **Inputs:**
  - **Force Control Knob:** Analog signal sampled via GPIO pin P1.2.
  - **Current Sensor Feedback:** Analog signal sampled via GPIO pin P1.3.
  - **Trigger Switch Signal:** Digital input from GPIO pin P3.0 to enable or disable the system.
- **Outputs:**
  - **PWM Control Signals:** Generated via Timer B2 to control motor speed and direction (output on P2.1).
  - **UART Data Transmission:** Real-time feedback for force and current values, as well as intermediate calibration commands.

4.2.3 Parameters

- Gains:
- **Current Limits:** Motor current was capped by limiting the PWM duty cycle to a maximum of 1999 (2000-cycle period).
- **Discretization and Scaling:**
  - ADC values were converted from raw 10-bit resolution to 8-bit scaled values.
  - A saturation scaling factor of 2000.0 / 255.0 (corresponding to 2^8 for 8-bit resolution) mapped the scaled digital input values to PWM duty cycle ranges (0–2000).

4.2.4 Implementation

Current Control Logic

We implemented a proportional-integrator control system where the error between the desired force and actual current was computed to generate the motor's control signal. The process included:

- Sampling ADC inputs for force (forcevalue) and current sensor feedback (currentsensorvalue).
- Calculating error as:

```
force_input_voltage = forcevalue * 3.22 / 256;
input_current = force_input_voltage * VoltagetoCurrentGain;
```

```
error = input_current - currentsensorvalue * VoltagetoCurrentGain;

integral_error += error;
if (integral_error > max_integral) {
    integral_error = max_integral;
} else if (integral_error < -max_integral) {
    integral_error = -max_integral;
}
```

- Amplifying the error using a proportional gain and adding Integrator Error to eliminate state error to determine the control output:

```
amplified_curr = (error * Kp) + (integral_error * Ki)
duty_cycle = (unsigned int)abs(amplified_curr * Saturation);
if (duty_cycle > 1999) duty_cycle = 1999;
```

This ensured safe force levels while providing a smooth control response.

Motor Direction Control

The motor's direction was determined based on the sign of the error, using GPIO pins P3.6 and P3.7 for H-Bridge control. A positive error triggered forward motion, while a negative error reversed the motor direction:

```
escape_byte = (error > 0) ? 1 : 0;
control_dc_motor(duty_cycle, escape_byte);
```

UART Communication

UART communication was configured to transmit the current values periodically. Calibration commands sent via UART were acknowledged and applied in real time.

System Activation

The system's operation was gated by a binary input signal on P3.0 connected to the trigger switch. When the signal was high, the control loop executed; otherwise, the system defaulted to a safe state, with no force applied:

```
if (BINARY_INPUT_PORT & BINARY_INPUT_PIN) {
    // Normal operation
} else {
    forcevalue = 0;  // Disable force
```

```
}
```

Full Code can be found in the appendix.

4.2.5 Testing and Results

The current control logic was tested using a known range of force inputs. The proportional control gain value and integrator value was fine-tuned to ensure minimal steady state error and a stable performance with very little overshoot, oscillations and unexpected behaviours. Results confirmed that the motor's current was limited to safe values, with the PWM duty cycle accurately reflecting the desired force – in turn mapping to the degree of grip applied by the end arms on the final product.

4.3 Functional Component #3 - End Effector Electrical Design

The electrical layout of the end-effector is as shown below:

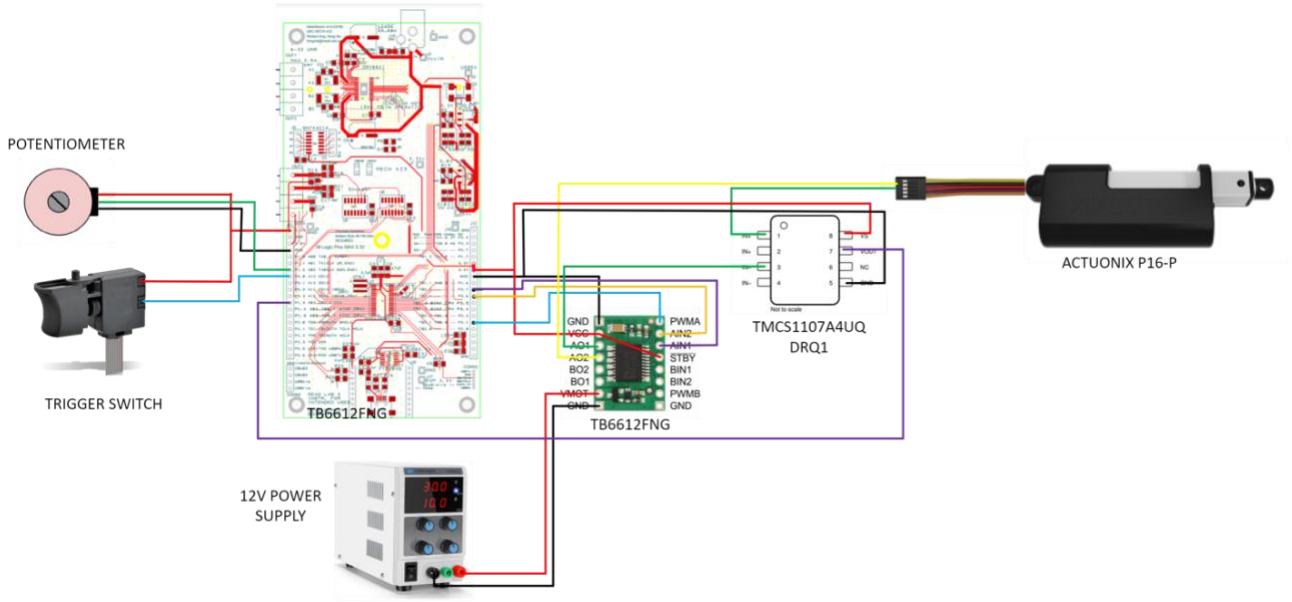


Figure 6: Electrical Schematic of the device

4.3.2 Inputs and Outputs

- **Inputs**:
    - Analog Signal (0-3.3V) from force setpoint adjustment potentiometer
    - Digital I/O from trigger switch

    - Analog Signal (0-3.3V) from TMCS1107A4UQDRQ1 Hall effect current sensor
- **Outputs**:
    - PWM duty cycle (0-100%) and direction to motor controller TB6612FNG

4.3.3 Parameters

- **Power Supply**: Should maintain steady operation for 30 minutes.
    - 12V, 1A max continuous motor power was achievable by the power supply
- **Motor Control Capacity**:
    - Motor controller TB6612FNG capable of 12V, 3A continuous, within spec for linear actuator
    - Aluminium heatsink was attached to protect current sensing IC TMCS1107A4UQDRQ1 and motor controller TB6612FNG
- **Sensor Resolution**:
    - 0-255 counts of resolution for potentiometer setpoint value
    - 8-bit resolution for current sensor

4.4 Functional Component #4 - End Effector Calibration

4.4.1 Objective and Design

The end effector calibration aimed to establish accurate relationships between motor current, applied force, and actuator movement, ensuring precise force adjustments and safe operation. The process used a load cell to measure applied force, a hall sensor for current feedback, a trigger switch to enable or disable the system, and actuator feedback to prevent mechanical snapping during backtracking.

A calibration algorithm integrated into the firmware scaled analog signals to actionable digital values and validated them against theoretical mappings from datasheets. The system also included safety checks to limit actuator movement based on the feedback signal, ensuring it only operated within safe mechanical limits.

4.4.2 Inputs and Outputs

Inputs:

- Load Cell – Potentiometer Feedback: Analog signal sampled via an ADC channel for applied force measurement.
- Hall Sensor Feedback: Analog signal sampled via an ADC channel for motor current sensing.
- Trigger Switch: Digital signal from GPIO pin to enable or disable the system.

- Actuator Feedback: Analog signal sampled via an ADC channel to monitor actuator position during backtracking.

Outputs:

- PWM Control Signals: Dynamically adjusted signals to control motor speed and direction based on inputs.
- UART Data Transmission: Real-time feedback for force, current, actuator position, and intermediate calibration data.

4.4.3 Discretization and Mapping

- The firmware scaled raw ADC values to digital representations and mapped them to physical units to ensure precise control and monitoring:
- Force Input Mapping:
  The analog signal from the load cell was mapped to its digital equivalent:

```
force_input_voltage = forcevalue * 3.22 / 256;
input_current = force_input_voltage * VoltagetoCurrentGain;
```

  The `3.22 mV/step` scaling factor mapped the ADC's voltage range to the digital resolution of 0–255 for 8-bit resolution.

- Hall Sensor Mapping:
  The analog output of the hall sensor was scaled to match datasheet specifications (e.g., 50mV/A sensitivity):

```
current_sensor_analog = currentsensorvalue * 3.22 / 256;
```

- Actuator Feedback Mapping:
  The actuator feedback signal was sampled and scaled to determine the current position during backtracking:

```
actuator_position = sampleADC(ACTUATOR_FEEDBACK_CHANNEL) * 3.22 / 256;
```

  This position was compared to a predefined safe limit to ensure the actuator did not exceed mechanical constraints.

4.4.4 Calibration Process

- Trigger Switch:
  The trigger switch, connected to a GPIO pin, enabled or disabled the system. When the trigger was high, the control logic executed; otherwise, the system defaulted to a safe state with no force applied:

```
if (BINARY_INPUT_PORT & BINARY_INPUT_PIN) {
    // Control logic
} else {
    forcevalue = 0;
    integral_error = 0;  // Reset integral term
}
```

- Force and Current Calibration:
  The load cell and hall sensor were calibrated using known weights and datasheet mappings. The PI control logic minimized the error between the desired force and actual current:

```
error = input_current - (current_sensor_analog - 1.96) *
VoltagetoCurrentGain;
integral_error += error;
amplified_curr = (error * Kp) + (integral_error * Ki);
```

- Actuator Backtracking Safety:
  During backtracking, the actuator feedback signal was continuously monitored to ensure the system operated within safe limits:

- Load Testing:
  The system was tested with varying loads, logging force, current, and actuator feedback via UART. Logged values were compared against theoretical predictions and datasheet mappings to validate accuracy.

4.4.5 Testing and Results

The end effector calibration integrated seamlessly into the firmware, leveraging ADC scaling, PI control logic, and real-time feedback. The trigger switch enabled safe and controlled operation, while the actuator feedback ensured mechanical safety during backtracking.

Discretization and mapping techniques provided precise control over motor speed and direction, validated through testing against datasheet specifications and physical measurements. This comprehensive calibration process ensured reliable operation under dynamic force and actuator conditions, maintaining both precision and safety.

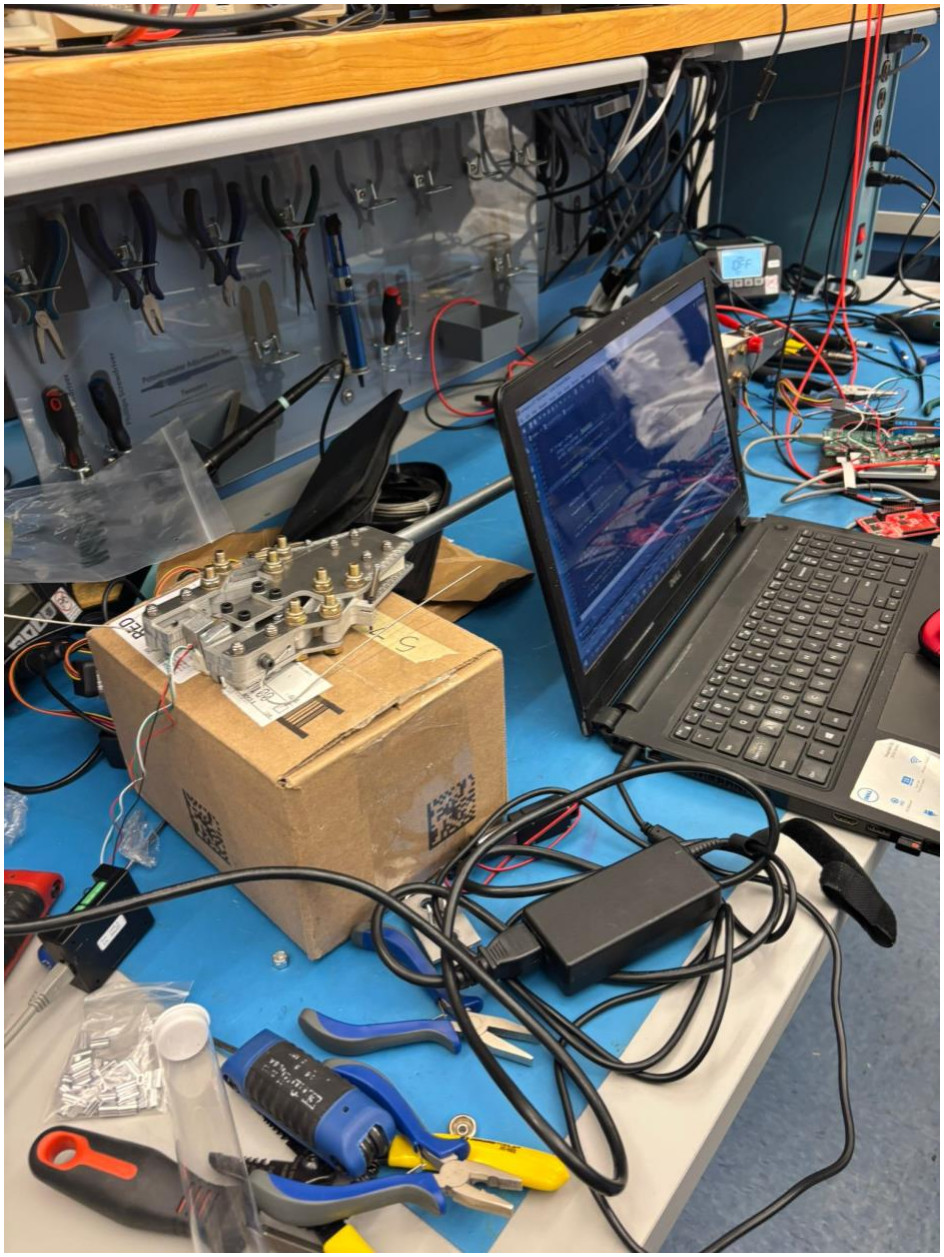


Figure 7: Load to Current calibration

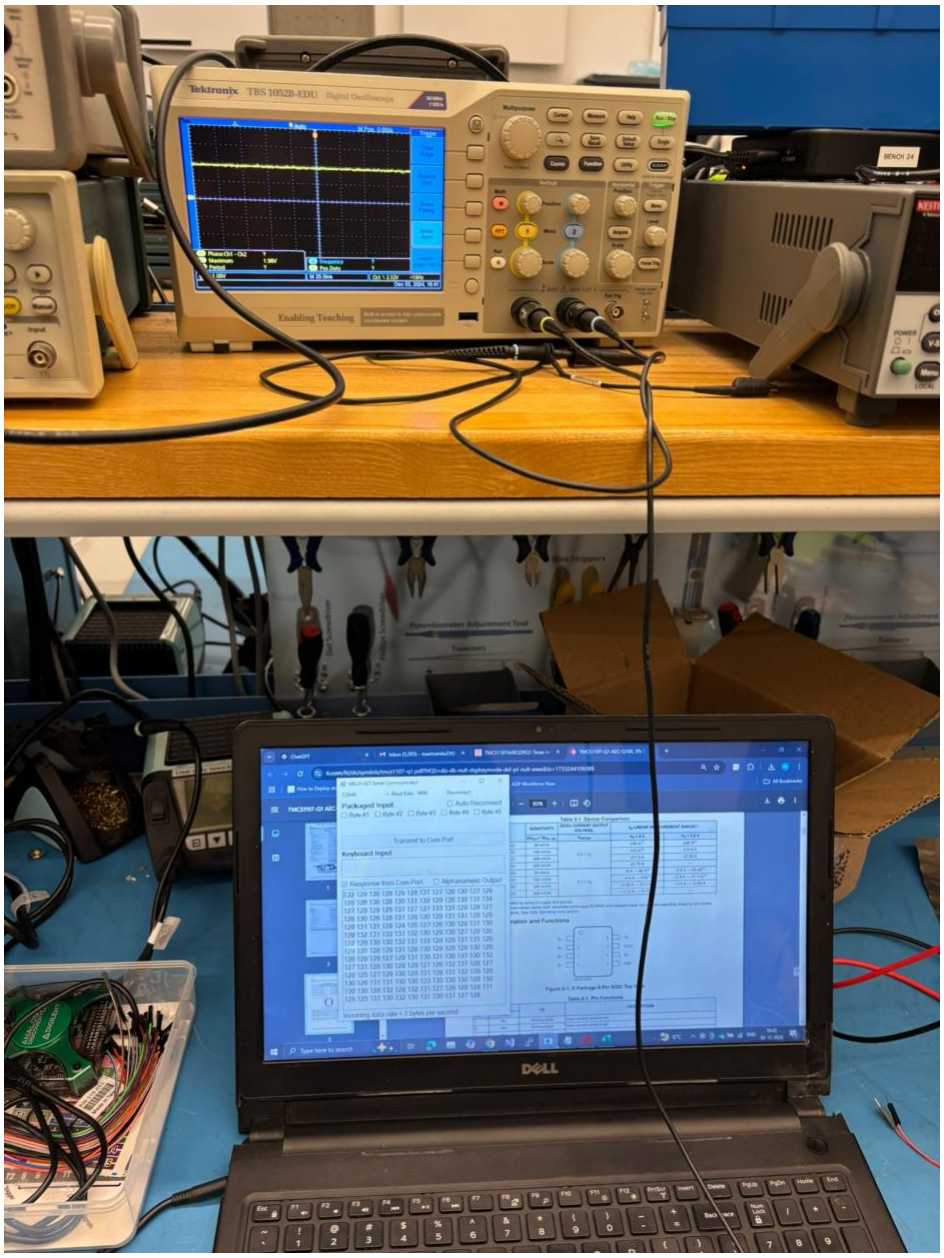


Figure 8: Discretization of input voltage from Hall sensor

4.5 Functional Component #5 - End Effector User Software

4.5.1 Objective and Design

The user interface for the end effector aimed to provide a user-friendly system for real-time force adjustment and calibration, displaying load mapping in terms of Newtons. A load cell connected to a Phidget driver was interfaced with a Python-based Phidget GUI, which displayed the applied force in real time. The GUI allowed users to visualize the current limits imposed on the end effector and the corresponding forces applied.

Given the time constraints, leveraging the existing Phidget GUI proved to be a practical and effective solution. If more time had been available, a custom interface with similar functionality could have been developed in Python or C# using widely available libraries like PyQt or .NET Framework.

4.5.2 Inputs and Outputs

Inputs:

- **Load Cell Feedback**: Analog signal from the load cell processed by the Phidget driver to measure applied force.
- **Current Limit Data**: User input to adjust motor current limits in the system.

Outputs:

- **Load Mapping Visualization**: The GUI displayed the load in Newtons, mapped to the motor current.
- **System Status Feedback**: Real-time updates on force applied, current consumed (current is seen electronically).

4.5.3 Discretization and Mapping

The interface integrated feedback from the load cell and mapped it to a digital display:

- Force Measurement:
  The load cell output was processed by the Phidget driver, which converted the analog voltage into a digital signal.
  The calibration_factor was derived from load cell specifications and validated during testing.

4.5.4 Implementation

- **Load Cell and Phidget Interface**:
  The load cell was connected to a Phidget driver, which interfaced with the Python-based GUI using the Phidget API. The driver converted analog signals from the load cell to digital values, which were scaled to Newtons in the GUI.

- **Calibration Algorithm**:
  The software included a calibration routine to map the load cell output to force. This routine relied on a set of known weights to determine the linear relationship between voltage and force. The process was automated within the Phidget GUI.

- **Real-Time Feedback**:
  The GUI provided real-time feedback on force applied, while current consumed was monitored using a power supply. Users could adjust the motor current limit using potentiometer on the device, which updated the corresponding force in the GUI dynamically.

4.5.5 Testing and Results

The system was thoroughly tested using a load cell with known weights to validate the calibration and scaling process. Real-time feedback was compared to theoretical values derived from datasheet mappings, ensuring accuracy.

The existing Phidget GUI proved to be a robust and effective interface for monitoring and adjusting system parameters. The load mapping and real-time visualization enhanced usability and enabled precise control over the end effector's operation.

While the pre-built Phidget GUI was effective, a custom interface could be developed in Python or C# for more tailored functionality and aesthetic improvements if additional development time were available.

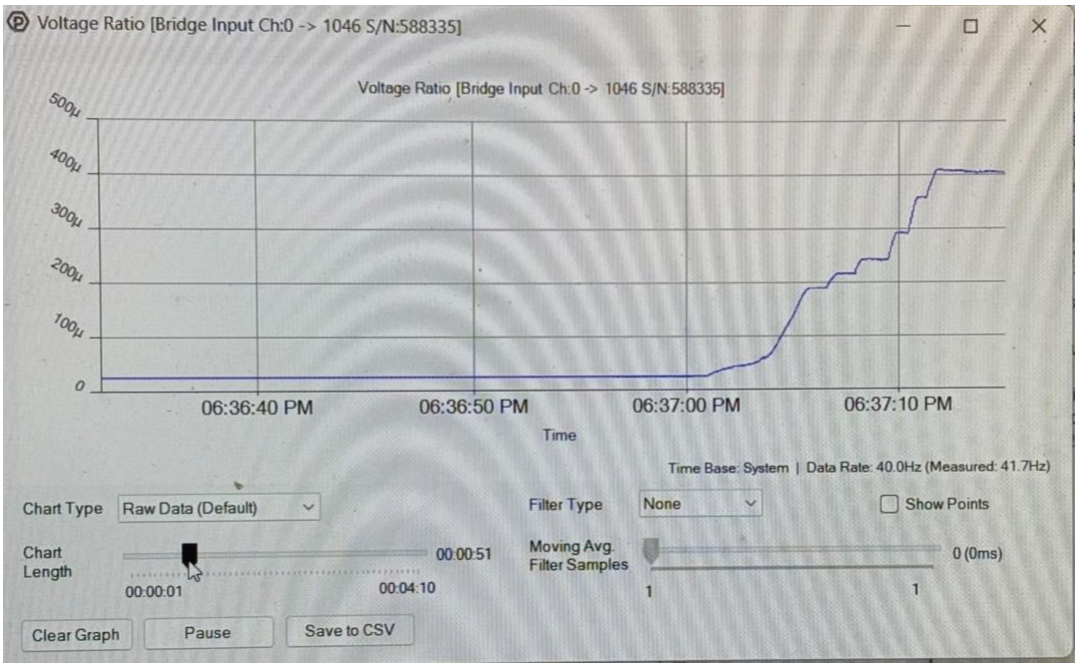


Figure 9: Phidget GUI for mapping force observed by load cell

5. System Evaluation

The system evaluation was conducted to verify the functionality, performance, and safety of the end effector under varying conditions and loads. The calibration process started with the hall sensor, where its analog output was mapped to known current values and validated against datasheet specifications. This calibration ensured accurate current sensing with an error margin of less than 5%, which was critical for maintaining precise force control. The load cell was then used to measure applied forces corresponding to varying current limits. These measurements demonstrated the system's ability to scale forces accurately while adhering to safety constraints. For example, at low current input, the gripper was able to hold a Coke can securely without crushing it. As the current limit was gradually increased, the applied force also increased, eventually causing the can to collapse. This highlighted the system's ability to dynamically adjust forces while maintaining precise control.

Further testing was conducted using a 25-pound dumbbell to evaluate the maximum load capacity of the end effector. At the highest current limit, the gripper successfully lifted the dumbbell, showcasing the system's robustness and capability to handle heavy loads. To assess safety features, a finger test was performed where a finger was placed between the gripper's jaws. At a low current limit, the gripper held the finger without causing discomfort, demonstrating its ability to apply minimal force when needed. However, as the current limit

was slightly increased, the force became painful, emphasizing the importance of current limits in protecting fragile or sensitive objects during operation.

These tests validated the end effector's ability to operate safely and effectively across a wide range of scenarios. The combination of hall sensor calibration, load cell measurements, and safety features like the trigger switch and actuator feedback ensured the system was not only precise and reliable but also adaptable to diverse tasks. Through extensive testing, the system proved to meet both performance and safety requirements, offering dynamic control and protection in real-world applications.

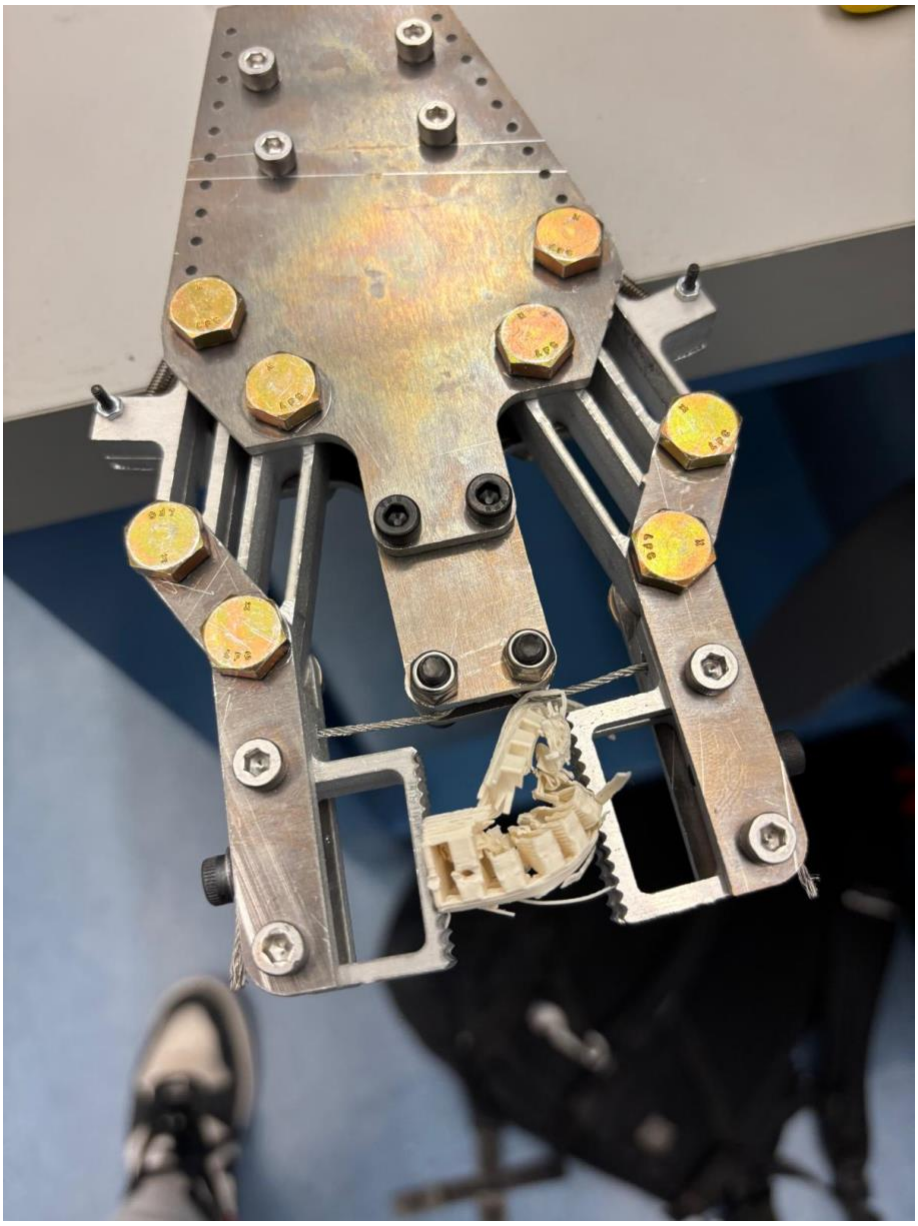


Figure 10: Crush testing objects by controlling the current limit on the gripper

6. Reflections

In this project, we hoped to explore the control of a more niche transmission system for robotics in a cable-driven linkage. Cable-driven robotics have some inherent advantages such as low transmission routing volume, low complexity, and ability to place actuators virtually anywhere. However, cable-drives come with some disadvantages like its inability to be double-acting (push and pull) as well as eventual cable creep and fatigue. Recently, cable-drive robotics have seen a resurgence with the advent of humanoid robotics, and

many consider cable-drive to be a viable transmission mechanism for implementing high-DOF humanoid robotic hands.

While our force-control via closed-loop current control efforts worked, it predictably only worked in the gripper closing action and the transmission mechanisms lack of back-drivability led to its inability to sense when there were forces above the force setpoint and for it to reduce actuator pressure. If we were to attempt this again, we would try for a more backdrivable and low friction setup. Some possibilities include a rotary actuator and pulley mechanism that would allow "pushing" and "pulling" via two cables on the joint. This way we should be able to detect when external forces push or pull our actuator away from the force setpoint.

Among the three most useful things we learned in MECH 423 for our project was:

1. The real-life implementation of a firmware PID controller. It is the foundation for many robotic controllers and is easy to implement and quick to manually tune for most applications. The difference between an only P controller and a PI controller is sometimes drastic in its performance when implemented appropriately.
2. Writing custom firmware to a microprocessor. It really opened our eyes to see how much you can really specify exactly what your microprocessor will do and to be able to implement a "real time" environment for your code that can respond in microseconds or even nanoseconds is going to be extremely useful for time-sensitive applications
3. The whole package of writing firmware to send UART to a PC and having that PC pick up UART packets and display onto a UI is going to be very useful for implementing any kind of testing or process that requires a user-interface much easier than on a PLC/HMI. Likewise, UART has helped us in understanding how communication protocols work and are designed and can be useful in working with other prominent CANBUS, MODBUS, etc.

Some limits of our knowledge as mechatronics engineers:

1. Lack of motor/actuator design experience. The only motor course we take is ELEC 343 and, in my opinion, it was not taught very well and was very much in the weeds of theory and mathematical derivation while having labs that were not very effective in conveying principles of motors and actuators. Having more in-depth motor/actuator/geartrain design experience would be very useful to not only be able to design powertrain systems from the ground-up but also to be able to purchase appropriate off-the-shelf motor/actuator options for the exact application in mind.

2. Lack of real time communication protocol knowledge. Whilst UART was great in understanding communication protocols. Synchronous programming and RTOS is very prominent in industry. Likewise understanding bootloaders, and how to write one, how to implement an operating system on a MCU is crucial. Maybe this should be covered in future versions of MECH 423.
3. Limited knowledge of ROS and C++. Now, to get a job in the field of robotics, C++ is crucial, and we do not get much chance to program robots in our mechatronics degree.
4. Scalable and Distributed Software Systems. We indeed made a simple C# application. How do we scale it and incorporate a bigger traffic? Things like scalability, containerization using Docker and deployment of applications, practical skills like infrastructure as a code for hosting on a Kubernetes environment/ cloud should be covered in a lab/ project setting as these skills are crucial for industry.

Appendix: Firmware code

```
#include "msp430fr5739.h"
#include <stdlib.h>

// Define the data packet start byte
#define START_BYTE 255

// ADC channels for GPIO analog inputs (P1.2 -> A2, P1.3 -> A3)
#define ADC_PIN1_CHANNEL ADC10INCH_4
#define ADC_PIN2_CHANNEL ADC10INCH_3
#define ADC_PIN3_CHANNEL ADC10INCH_2
#define ADC_PIN4_CHANNEL ADC10INCH_5

// GPIO binary input (P3.0)
#define BINARY_INPUT_PORT P3IN
#define BINARY_INPUT_PIN BIT0

unsigned int adc_value1, adc_value2;       // Store raw 10-bit ADC results
unsigned char forcevalue, currentsensorvalue = 0; // Store 8-bit ADC results
volatile unsigned char transmit = 0;        // Flag to trigger data transmission
volatile unsigned char current_pin = 0;     // Track which pin is currently

volatile unsigned int force_input_voltage;
volatile unsigned int input_current;
volatile unsigned int curr_digital;
volatile unsigned int amplified_curr;
```

```
// PI control variables
volatile signed int error;                  // Proportional error
volatile signed int integral_error = 0;     // Accumulated integral error
const float Kp = 10;                        // Proportional gain
const float Ki = 1.0;                       // Integral gain
const unsigned int max_integral = 1000;     // Maximum integral value to prevent
windup

// Define Saturation as a scaling factor to map from 0-255 to 0-2000
const float Saturation = 2000.0f / 255.0f;

#define VoltagetoCurrentGain 1
#define MaxCurr 1

// Function Prototypes
void configure_ADC10();
void configure_analog_pins();
void configure_binary_input();
void configure_UART();
void configure_timer_b0();
void configure_timer_b();
void configure_clocks();
void configure_gpio_3_6_and_3_7();
unsigned char sampleADC(unsigned int channel);
void control_dc_motor(unsigned int duty_cycle, unsigned char direction);
void transmit_uart(unsigned int data);

// Function to configure binary input pin (P3.0 as input)
void configure_binary_input() {
    P3DIR &= ~BINARY_INPUT_PIN;  // Set P3.0 as input
    P3REN |= BINARY_INPUT_PIN;   // Enable pull-up/down resistor
    P3OUT |= BINARY_INPUT_PIN;   // Configure pull-up resistor
}

// Function to configure ADC for sampling GPIO analog inputs
void configure_ADC10() {
    ADC10CTL0 = ADC10SHT_2 | ADC10ON;  // Sample and hold time = 16 ADC10CLK cycles,
ADC on
    ADC10CTL1 = ADC10SHP | ADC10SSEL_3; // Use sampling timer, SMCLK source
    ADC10CTL2 = ADC10RES;              // 10-bit resolution
}

// Function to configure GPIO pins as analog inputs
void configure_analog_pins()
{ P1SEL1 |= BIT2 | BIT3 | BIT4 | BIT5; // Configure P1.4 and P1.3, P1.5 as analog
inputs
P1SEL0 |= BIT2 | BIT3 | BIT4 | BIT5; }

// Function to sample ADC for a given channel
unsigned char sampleADC(unsigned int channel) {
    ADC10CTL0 &= ~ADC10ENC;            // Disable ADC before changing channels
```

```
    ADC10MCTL0 = channel;              // Select the channel (e.g., A2 for P1.2, A3
for P1.3)
    ADC10CTL0 |= ADC10ENC | ADC10SC;   // Enable and start conversion
    while (ADC10CTL1 & ADC10BUSY);     // Wait until conversion completes
    return ADC10MEM0 >> 2;             // Return 8-bit result (shifted 10-bit)
}

// UART Configuration
void configure_UART() {
    // Select SMCLK for UART and configure UART pins
    P2SEL0 &= ~(BIT5 | BIT6);
    P2SEL1 |= (BIT5 | BIT6);

    UCA1CTLW0 |= UCSWRST;              // Put UART in reset mode
    UCA1CTLW0 |= UCSSEL__SMCLK;        // Use SMCLK (1 MHz after division)

    UCA1BRW = 104;                     // Set baud rate for 9600 (SMCLK 1 MHz)
    UCA1MCTLW = 0xD600;                // Set modulation UCBRSx=0xD6, UCOS16=1

    UCA1CTLW0 &= ~UCSWRST;             // Release UART from reset
}

// Configure Timer B0 to periodically trigger the encoder count transmission
void configure_timer_b0() {
    TB0CCTL0 = CCIE;                   // Enable interrupt for CCR0
    TB0CCR0 = 40000 - 1;                // Set interval (e.g., 8000 for 10 ms with
SMCLK = 1 MHz)
    TB0CTL = TBSSEL_2 | MC_1 | TBCLR;  // SMCLK, up mode, clear timer
}

// Timer B Configuration for DC Motor PWM (TB2.1 -> P2.1)
void configure_timer_b() {
    P2DIR |= BIT1;          // Set P2.1 as output for Timer B (TB2.1)
    P2SEL1 &= ~BIT1;        // Clear P2SEL1 to select Timer B functionality
    P2SEL0 |= BIT1;         // Set P2SEL0 to select TB2.1 (PWM output)

    // Stop Timer B2 during setup
    TB2CTL = TBSSEL_2 | MC_0 | TBCLR;  // SMCLK as clock source, stop mode, clear
timer

    // Configure Timer B2 for PWM (500 Hz)
    TB2CCR0 = 2000 - 1;     // Set the PWM period for 500 Hz (SMCLK 1 MHz / 2000)
    TB2CCTL1 = OUTMOD_7;    // Set/reset output mode (PWM mode)
    TB2CCR1 = 1000;         // Set initial duty cycle to 50% (1000/2000)

    // Start Timer B2 in up mode
    TB2CTL = TBSSEL_2 | MC_1 | TBCLR;  // SMCLK as clock source, up mode
}

// Clock Configuration (8 MHz DCO, SMCLK 1 MHz)
```

```
void configure_clocks() {
    CSCTL0 = CSKEY;
    CSCTL1 |= DCOFSEL_3;               // DCO frequency select: 8 MHz
    CSCTL2 |= SELS__DCOCLK;            // DCO as the source for SMCLK
    CSCTL3 |= DIVS__2;                 // SMCLK divider 1
}

// GPIO Configuration for H-Bridge Direction Control (P3.6 and P3.7)
void configure_gpio_3_6_and_3_7() {
    P3DIR |= BIT6 | BIT7;  // Set P3.6 and P3.7 as outputs
    P3OUT &= ~BIT6;        // Set P3.6 high
    P3OUT |= BIT7;         // Set P3.7 low
}

// UART Transmission Function
void transmit_uart(unsigned int data) {
    while (!(UCTXIFG));          // Wait until the transmit buffer is ready
    UCA1TXBUF = data;            // Transmit data
}

// Control DC Motor using received duty cycle and direction
void control_dc_motor(unsigned int duty_cycle, unsigned char direction) {
    TB2CCR1 = duty_cycle; // Set PWM duty cycle for DC motor speed
    if (direction == 1) {
        P3OUT &= ~BIT6;  // Set P3.6 high
        P3OUT |= BIT7;   // Set P3.7 low (forward direction)
    } else {
        P3OUT |= BIT6;   // Set P3.6 low
        P3OUT &= ~BIT7;  // Set P3.7 high (reverse direction)
    }
}

// Timer B0 Interrupt ISR for periodic UART transmission
#pragma vector = TIMER0_B0_VECTOR
__interrupt void TIMER0_B0_ISR(void) {
    forcevalue = sampleADC(ADC_PIN1_CHANNEL);  // Sample P1.2 (A2)
    currentsensorvalue = sampleADC(ADC_PIN2_CHANNEL);  // Sample P1.3 (A3)
    TB0CCTL0 &= ~CCIFG;    // Clear interrupt flag
}

// Main program
int main(void) {
    WDTCTL = WDTPW | WDTHOLD;  // Stop watchdog timer

    configure_clocks();         // Configure clock settings
    configure_UART();           // Configure UART
    configure_timer_b0();       // Configure Timer B for periodic transmission
    configure_timer_b();        // Configure Timer B for PWM (DC motor control)
    configure_gpio_3_6_and_3_7(); // Configure GPIO for H-Bridge (motor direction
control)
    configure_analog_pins();    // Configure P1.2 and P1.3 as analog inputs
    configure_ADC10();          // Set up ADC for GPIO inputs
```

```
    configure_binary_input();   // Configure P3.0 as binary input

    __bis_SR_register(GIE);     // Enable global interrupts

    while (1) {
        if (BINARY_INPUT_PORT & BINARY_INPUT_PIN) {
            // Binary input HIGH
            force_input_voltage = forcevalue * 3.22 / 256;
            input_current = force_input_voltage * VoltagetoCurrentGain;

            error = input_current - currentsensorvalue * VoltagetoCurrentGain;

            integral_error += error;
            if (integral_error > max_integral) {
                integral_error = max_integral;
            } else if (integral_error < -max_integral) {
                integral_error = -max_integral;
            }

            // Calculate control output (PI controller)
            amplified_curr = (error * Kp) + (integral_error * Ki);

            // Scale amplified_curr to duty cycle
            duty_cycle = (unsigned int)abs(amplified_curr * Saturation);

            // Cap the duty cycle to 2000 if it exceeds the limit
            if (duty_cycle > 1999) {
                duty_cycle = 1999;
            }

            // Set escape_byte based on the sign of the error
            escape_byte = (error > 0) ? 1 : 0;

            // Control motor speed and direction
            control_dc_motor(duty_cycle, escape_byte);
        } else {
            // Binary input LOW: Reset force value and integral term
            forcevalue = 0;
            integral_error = 0;  // Reset integral error
        }
    }
}
```