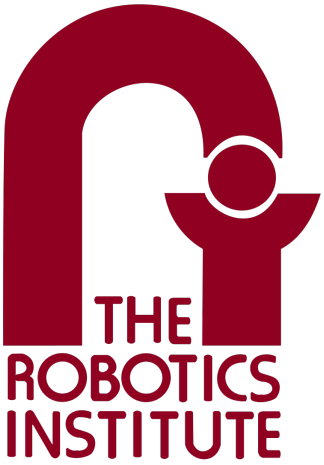
Individual Lab Report - 1



Lunar ROADSTER

Team I

Author: **Simson D'Souza**
Andrew ID: sjdsouza
E-mail: *sjdsouza@andrew.cmu.edu*

Teammate: **Ankit Aggarwal**
ID: ankitagg
E-mail: *ankitagg@andrew.cmu.edu*

Teammate: **Deepam Ameria**
ID: dameria
E-mail: *dameria@andrew.cmu.edu*

Teammate: **Bhaswanth Ayapilla**
ID: bayapill
E-mail: *bayapill@andrew.cmu.edu*

Teammate: **Boxiang (William) Fu**
ID: boxiangf
E-mail: *boxiangf@andrew.cmu.edu*

Supervisor: **Dr. William "Red" Whittaker**
Department: Field Robotics Center
E-mail: *red@cmu.edu*

February 7, 2025

Carnegie Mellon University
Robotics Institute

Contents

1 Individual Progress

1.1 Sensors and Motor Lab

My responsibilities included implementing DC motor control using an encoder with position feedback and PID control, enabling users to control the motor through a GUI. This allowed the motor to rotate by a specified number of degrees from an initial position or operate at a user-defined velocity in either direction. The velocity could be set within a range of 0 to 118 RPM, with positive and negative signs indicating direction, while position control allowed inputs between 0 and 360 degrees, also with directional control through sign convention. Additionally, I integrated a TMP36 temperature sensor to convert raw sensor readings into temperature values in degrees Celsius, which were displayed on the GUI. The motor control and temperature sensing operated independently, each providing real-time data and control functionality through the interface.

1.1.1 DC Motor with Encoder

The initially provided DC motor with an encoder and L298 motor driver was not used. Instead, a ServoCity planetary gear motor (SKU: 638324, 118 RPM @12VDC) and a Roboclaw 2x15A motor controller were implemented. The Arduino Due was chosen due to its extensive I/O capabilities, but as it could not supply sufficient power, an external 12V high-voltage DC power supply was used to power the motor through the Roboclaw.

To achieve precise motor control, encoder feedback was utilized, with 3416 encoder ticks per revolution. The encoder values enabled two modes of operation:

1. Position Control: The motor could rotate by a user-specified number of degrees from an initial position. The encoder readings were converted into ticks using the relation:

$$Ticks = \frac{3416}{360} \times Degrees \tag{1}$$

   The RoboClaw's PID position control was configured using Kp = 2.0, Ki = 0.5, and Kd = 1.0 to ensure smooth and accurate movement. The target position was continuously monitored, and the motor stopped within a tolerance of ±15 ticks. The target position is calculated based on the user-specified degrees, converted into encoder ticks. The RoboClaw's position PID controller then drives the motor to the desired position while continuously monitoring encoder readings to stop within a defined tolerance.

2. Velocity Control: The motor speed was set using encoder feedback, allowing motion in either direction. The velocity was scaled to encoder counts per second (Qpps) using.

$$EncoderSpeed = \frac{UserSpeed \times MAXQPPS}{100} \tag{2}$$

   where MAXQPPS = 6718 was the maximum speed in counts per second. The velocity PID parameters (Kp = 1.0, Ki = 0.5, Kd = 0.25) were tuned to provide stable speed control. The user-input speed is mapped to encoder counts per second and sent to the RoboClaw, which regulates motor speed using its built-in velocity PID controller

The RoboClaw library provides built-in functions to simplify motor control by handling encoder data, speed, and position commands. It includes functions like ReadEncM1() to read encoder counts, SpeedM1() to set motor velocity, and SpeedAccelDeccelPositionM1() to control position with acceleration and deceleration parameters. This ensures smooth motion in both directions based on the given commands.

1.1.2 TMP36 Temperature Sensor

The TMP36 temperature sensor was connected to the Arduino Due with its Vcc pin receiving 3.3V, GND connected to ground, and Vout linked to an analog input pin (A0) for reading the output voltage. Since the sensor operates on a low voltage range of 2.7V to 5.5V and provides an analog voltage proportional to the temperature, it was interfaced with an analog input to capture continuous variations in the output. The Arduino's 10-bit ADC (Analog-to-Digital Converter) converted this voltage into a digital value between 0 and 1023. Given the 3.3V reference voltage, the sensor's output voltage was calculated as:

$$Vout = \frac{ADCreading \times 3.3}{1024} \tag{3}$$

To derive the temperature in °C, the sensor's scale factor of 10mV/℃ with a 500mV offset was applied, using the formula:

$$Temperature(^{\circ}C) = (Vout - 0.5) \times 100 \tag{4}$$

This conversion ensured accurate temperature readings based on the sensor's output voltage. The readings were displayed on the GUI.

1.2 MRSD Project

In the MRSD project, I contributed to the prototype dozer development alongside Deepam, conducting preliminary teleoperation and grading tests. I collaborated with Ankit to create the circuit diagram and worked with team members to set up the FARO scanner and scan the Moon Pit.

For navigation, I processed the FARO scan data to generate an occupancy grid map. Since the FARO scanner outputs data in (.fls) format, which is incompatible with ROS, a conversion process was required. Using Recap Pro, I first exported the data to (.pts) format and then converted it to (.pcd) for ROS compatibility. The Figure 1 below show the visualization of the point cloud data of the Moon Pit.

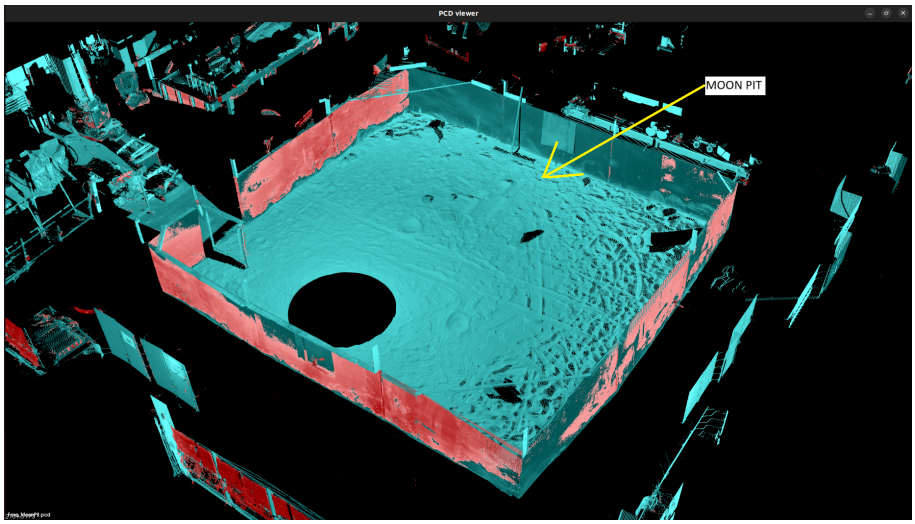


Figure 1: FARO MoonPit Scan

The point cloud data was then downsampled using a Voxel grid to reduce complexity. To create an occupancy grid map, a thresholding approach was applied, marking large craters as occupied and traversable areas as free space based on depth variations. However, the initial occupancy grid lacked accuracy for navigation, requiring further refinement, which I am currently working on.

2 Challenges

2.1 Sensors and Motor Lab

I faced several challenges related to sensor and motor control. One issue with the TMP35 temperature sensor arose when it was supplied with 5V via the breadboard, resulting in inaccurate readings. The shared Vcc and ground connections through the breadboard caused interference, which not only led to noisy sensor data but also made the motors jitter. To resolve this, I connected the sensor to the 3.3V pin of the Arduino Due, which was not being used by other components, and the readings became more accurate. For the motor, I had to fine-tune the PID controller parameters to ensure smooth motion when reaching the target position. Additionally, interfacing all the components together and controlling them via a GUI presented challenges, as each component required different data types. To overcome this, I interfaced and tested each component individually before combining them.

2.2 MRSD Project

One of the main challenges in the MRSD project was making the FARO scan data of the Moon Pit compatible with ROS. The FARO data format could not be directly converted, so I had to use PCL libraries to first convert it into a compatible format before further conversion into a ROS-compatible format. Additionally, certain libraries used for point cloud data processing were not compatible with the ROS2 Humble version, requiring me to process the data step-by-step, including converting it into a grid map, occupancy grid map, and 2D cost map—a tedious process. Another challenge the team faced was the robot's mobility on uneven terrain. During testing, issues with suspension and the dozer's effectiveness in creating a path raised concerns. As sand

accumulated in front, the robot stalled, leading us to explore solutions such as implementing an active dozer mechanism, switching wheels, and using high-torque motors to improve traction and reduce wheel slippage.

3 Teamwork

3.1 Sensors and Motor Lab

Given my contributions outlined in the Individual Progress section, the following are the contributions of my team members.

1. **Ankit**: He worked on controlling the speed and direction of a stepper motor using a potentiometer.
2. **Deepam**: He handled the interfacing of an IMU sensor and a servo motor.
3. **Bhaswanth**: He integrated an ultrasonic range finder, implemented median/-mode filtering and a transfer function.
4. **William**: He contributed by developing the GUI and creating an Arduino template code for seamless integration.

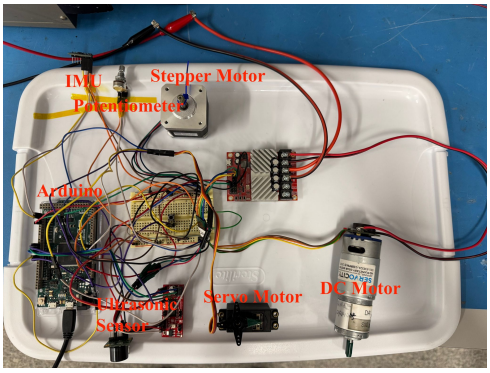


Figure 2: Integrated Circuit with Motors and Sensors

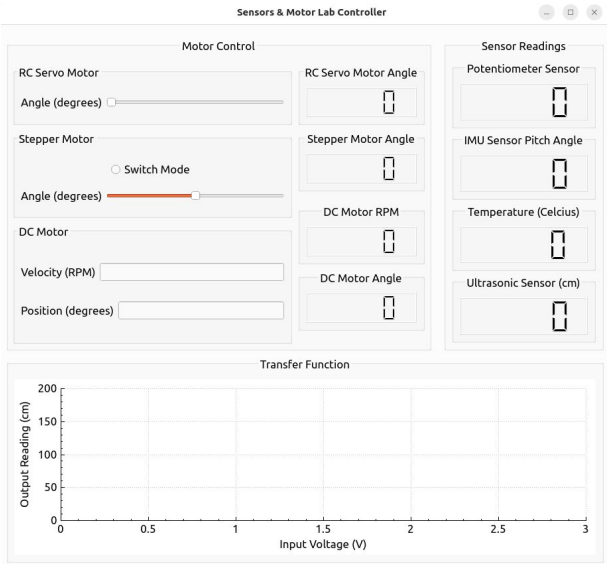


Figure 3: Graphical User Interface (GUI)

3.2 MRSD Project

Building on my contributions detailed in the Individual Progress section, my team members played key roles in various aspects of the MRSD project.

1. **Ankit**: He worked on wheel design and printing, rover hardware setup and maintenance, and VectorNav IMU integration. Additionally, he collaborated with me on the electrical circuit as shown in Figure 4, participated in preliminary testing with the team, and contributed to project management.

2. **Deepam**: He and I worked together on the prototype dozer development and preliminary tests for teleoperation and grading as shown in Figure 5, while he also led the dozer blade and mechanism design and collaborated with the team on the FARO scanner setup and Moon Pit scanning.

3. **Bhaswanth**: He set up the Jetson and encoder drivers, worked with William on teleoperation and ZED camera integration, established the operations terminal, and participated in the FARO scanning process alongside the team.

4. **William**: He contributed to Moon Pit crater distribution analysis, LAN setup, collaborated with Bhaswanth on teleoperation and ZED camera setup, and assisted the team during FARO scanning.

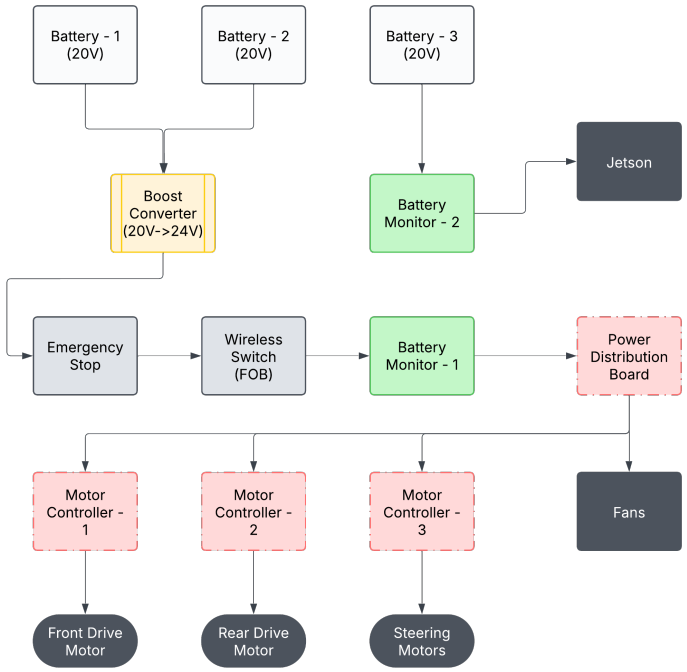


Figure 4: Electrical circuit

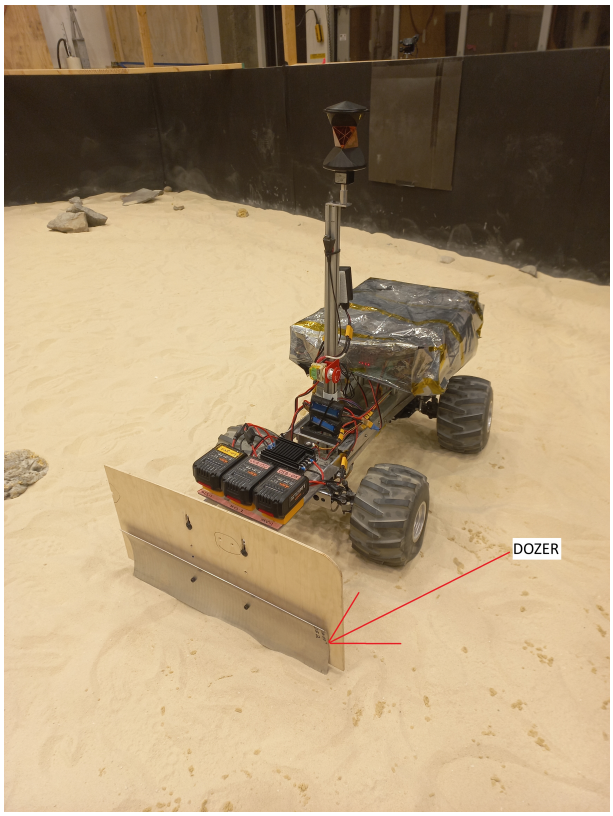


Figure 5: Prototype dozer testing

4 Plans

4.1 Sensors and Motor Lab

The decision to change the DC motor and motor driver was based on the relevance of the ServoCity planetary gear motor and RoboClaw motor controller to our project. We also acquired high-torque motors, and the challenges faced during sensor and motor control will aid in integrating these new components. Additionally, troubleshooting interfacing issues has reinforced a structured approach of testing one subsystem at a time, which we will follow in the MRSD project during full system integration.

4.2 MRSD Project

Until the next lab demo, I plan to use the ZED camera instead of the FARO scanner, as it offers better support and available packages for generating an occupancy grid map for navigation. I will start by mapping the environment to obtain point cloud data, which will then be processed into a 2D costmap. This will aid in localization and navigation, aligning with our project schedule. Additionally, our team aims to complete the preliminary design of the dozer and start working on the dozer mechanism for testing. We will also replace the current motor with a high-torque motor to improve traction and optimize cable management for a cleaner setup.

5 Sensors and Motor Control Quiz

5.1 Reading a Datasheet

1. What is the sensor's range?
   The sensor's measurement range is ±3g.

2. What is the sensor's dynamic range?
   The minimum dynamic range is 6g and maximum dynamic range is 7.2g.

3. What is the purpose of the capacitor CDC on the LHS of the functional block diagram on p. 1? How does it achieve this?
   The CDC capacitor, shown on the left side of the functional block diagram (Figure 1), functions as a decoupling capacitor. It filters out high-frequency noise from the power supply, ensuring stable voltage delivery to the sensor. It achieves this by blocking any sudden voltage spikes or drops, thereby ensuring smooth sensor operation.

4. Write an equation for the sensor's transfer function.
   The transform function is

$$Vout = 1.5V + 0.3V/g * a \tag{5}$$

   where, a is applied acceleration in g

5. What is the largest expected nonlinearity error in g?
   The largest expected nonlinearity error in g is $7.2g * 0.3/100 = 0.0216g$.

6. What is the sensor's bandwidth for the X- and Y-axes?
   The bandwidth for the X and Y axes is 0.5 Hz to 1600 Hz.

7. How much noise do you expect in the X- and Y-axis sensor signals when your measurement bandwidth is 25 Hz? The typical noise of the ADXL335 is determined by (given on pg.11 of the datasheet)

$$rmsNoise = NoiseDensity \times (\sqrt{BW \times 1.6}) \tag{6}$$

   The noise density for X and Y axes is 150 μg $\sqrt{Hz}$ rms. The RMS noise can be calculated as:

$$rmsNoise = 150\mu g\sqrt{Hz} \times (\sqrt{25 \times 1.6}) = 948.683\mu g = 0.949mg \tag{7}$$

8. If you didn't have the datasheet, how would you determine the RMS noise experimentally? State any assumptions and list the steps you would take.
   Assumptions:

   - The sensor is stationary during the measurement.
   - There are no external vibrations.

   Steps:

   - Place the accelerometer on a stable, non-vibrating surface.

- Record the output voltage of the X and Y axes over a period of time using a data acquisition system. The sampling rate should be sufficiently high to capture potential noise.
- Convert the voltage readings to acceleration using the known sensitivity (if known) or an estimated value.
- Subtract the mean (zero-g bias) from the data to isolate the noise component.
- Calculate RMS noise.
- To limit the measurement to a specific bandwidth, apply a low-pass filter to your data before recalculating the RMS noise. This step ensures that only noise within the desired frequency range is considered.

5.2 Signal Conditioning

1. Filtering

   - Name at least two problems you might have in using a moving average filter.
     a. A moving average filter introduces a time delay in the output because it averages multiple data points.
     b. While smoothing out noise, a moving average filter can also soften or blur sudden changes in the signal, potentially hiding sudden movements.

   - Name at least two problems you might have in using a median filter.
     a. Median filters require sorting data within the window to find the median value, which is more computationally intensive than simple averaging.
     b. Less effective at reducing random noise compared to other filters.

2. Opamps
   For the given Opamp gain and offset circuit, the V_{out} is

$$V_{out} = V_{in}(1 + \frac{R_f}{R_i}) - V_{ref}(\frac{R_f}{R_i}) \tag{8}$$

   - Your uncalibrated sensor has a range of -1.5 to 1.0V (-1.5V should give a 0V output and 1.0V should give a 5V output).
     Here, $V_1 = V_{ref}$ and $V_2 = V_{in}$

$$0 = -1.5(1 + \frac{R_f}{R_i}) - V_{ref}(\frac{R_f}{R_i}) \tag{9}$$

$$5 = 1.0(1 + \frac{R_f}{R_i}) + V_{ref}(\frac{R_f}{R_i}) \tag{10}$$

   On solving, we get

$$\frac{R_f}{R_i} = 1 \tag{11}$$

$$V_{ref} = -3V \tag{12}$$

- Your uncalibrated sensor has a range of -2.5 to 2.5V (-2.5V should give a 0V output and 2.5V should give a 5V output).
  Required Gain and Offset: To map -2.5V to 0V and 2.5V to 5V, the circuit needs a gain of -1 and an offset of 2.5V.
  Case 1: If V_1 is the input, solving the equations gives:

$$\frac{R_f}{R_i} = -1$$

  This is not feasible because it cannot have a negative feedback ratio in this configuration.
  Case 2: If V_2 is the input, solving results in:

$$\frac{R_f}{R_i} = 0$$

  This implies no amplification, which is also not practical.
  Hence, calibration isn't possible with this circuit because a single op-amp can't simultaneously achieve the required gain of -1 and the 2.5V offset needed to shift the output range from 0V to 5V.

5.3 Control

1. If you want to control a DC motor to go to a desired position, describe how to form a digital input for each of the PID (Proportional, Integral, Derivative) terms.
   By applying the PID controller equation and feeding the sensor measurement data as input, the controller will generate the necessary output value to adjust the system accordingly.
   Proportional (P): The input is the current error (difference between desired and actual position). This is directly proportional to the error.
   Integral (I): The input is the accumulated error over time, helping eliminate steady-state error.
   Derivative (D): The input is the rate of change of the error.

2. If the system you want to control is sluggish, which PID term(s) will you use and why?
   Use the Proportional (P) term, as it responds to the current error. Increasing the proportional gain can make the system respond faster.

3. After applying the control in the previous question, if the system still has significant steady-state error, which PID term(s) will you use and why?
   Use the Integral (I) term, as it accumulates the error over time and drives the error to zero, eliminating steady-state error.

4. After applying the control in the previous question, if the system still has overshoot, which PID term(s) will you apply and why?
   Apply the Derivative (D) term to dampen the system's response. This term reacts to the rate of change in the error, helping to minimize overshoot.

6 Appendix

6.1 Arduino Code

6.1.1 DC Motor with Encoder

```
#include <RoboClaw.h>

RoboClaw roboclaw(&Serial1, 10000);

#define ADDRESS 0x80

#define ENCODER_TICKS_PER_REV 3416
#define DEGREE_TO_TICKS (ENCODER_TICKS_PER_REV / 360.0)

// PID Tuning Parameters
#define Kp_pos 2.0
#define Ki_pos 0.5
#define Kd_pos 1.0

#define Kp_vel 1.0
#define Ki_vel 0.5
#define Kd_vel 0.25

#define MAX_QPPS 6718

bool isMoving = false;
uint32_t targetPosition = 0;

void setup() {
    Serial.begin(115200);
    Serial1.begin(38400);

    if (!roboclaw.ReadError(ADDRESS)) {
        Serial.println("RoboClaw connected successfully.");
    } else {
        Serial.println("Error detected in RoboClaw!");
    }

    // Set PID values
    roboclaw.SetM1VelocityPID(ADDRESS, Kp_vel, Ki_vel,
    Kd_vel, MAX_QPPS);
    roboclaw.SetM1PositionPID(ADDRESS, Kp_pos, Ki_pos,
    Kd_pos, Kp_vel, Ki_vel, Kd_vel, MAX_QPPS);

    roboclaw.SpeedM1(ADDRESS, 0);
    Serial.println("Motor stopped at startup.");
}

void loop() {
```

```
    if (isMoving) {
        uint32_t currentPos = roboclaw.ReadEncM1(ADDRESS);
        if (abs((int32_t)(currentPos - targetPosition)) <= 15) {
            roboclaw.SpeedM1(ADDRESS, 0);  // Stop motor
            Serial.println("Target position reached.");
            isMoving = false;  // Reset movement flag
        }
    }

    if (Serial.available()) {
        String input = Serial.readStringUntil('\n');
        input.trim(); // Remove leading/trailing spaces

        if (input.startsWith("M ")) {
            int degrees = input.substring(2).toInt();
            moveMotorByDegrees(degrees);
        } else if (input.startsWith("V ")) {
            int speed = input.substring(2).toInt();
            setMotorVelocity(speed);
        }
    }
}

void moveMotorByDegrees(int degrees) {
    uint32_t currentPos = roboclaw.ReadEncM1(ADDRESS);
    targetPosition = currentPos + (degrees * DEGREE_TO_TICKS);

    Serial.print("Moving motor to position: ");
    Serial.println(targetPosition);

    roboclaw.SpeedAccelDeccelPositionM1(ADDRESS, 10000,
    MAX_QPPS, 10000, targetPosition, 0);

    isMoving = true;
}

void setMotorVelocity(int speed) {
    int encoderSpeed = (speed * MAX_QPPS) / 100;
    roboclaw.SpeedM1(ADDRESS, encoderSpeed);

    Serial.print("Setting motor speed to: ");
    Serial.println(encoderSpeed);
}
```

6.1.2 TMP36 Temperature Sensor

```
int sensorPin = A0;
void setup()
{
  Serial.begin(9600);
```

```
}

void loop()
{
 int reading = analogRead(sensorPin);

 float voltage = reading * 5.0;
 voltage /= 1024.0;

 Serial.print(voltage); Serial.println(" volts");

 float temperatureC = (voltage - 0.5) * 100 ;
 Serial.print(temperatureC); Serial.println(" degrees C");
 delay(1000);
}
```

6.1.3 Integrated code for the entire system

```
#include <Adafruit_Sensor.h>
#include <Adafruit_MPU6050.h>
#include <Wire.h>
#include <Servo.h>
#include <SharpIR.h>
#include <RoboClaw.h>
#include <AccelStepper.h>

#define SERIAL_PORT Serial

#define ADDRESS 0x80

// Motor & Encoder parameters
#define ENCODER_TICKS_PER_REV 3416
#define DEGREE_TO_TICKS (ENCODER_TICKS_PER_REV / 360.0)

// Position PID Tuning
#define Kp_pos 2.0
#define Ki_pos 0.5
#define Kd_pos 1.0

// Velocity PID Tuning
#define Kp_vel 1.0
#define Ki_vel 0.5
#define Kd_vel 0.25

// Maximum speed in encoder counts per second
#define MAX_QPPS 6718

// Pins
#define EN_StepperDriver 2
#define Stp_StepperDriver 3
```

```
#define Dir_StepperDriver 4
#define servoPin 6
#define PushButtonPin 7
#define PotentiometerPin A0
#define temperaturePin A1
#define ultrasonicPin A2
#define PushbuttonPin 7

AccelStepper stepper(AccelStepper::DRIVER,
Stp_StepperDriver, Dir_StepperDriver);


// GLOBAL VARIABLES
unsigned long previousMillis = 0;
const unsigned long interval = 100;

Adafruit_MPU6050 mpu;
Servo servo;
RoboClaw roboclaw(&Serial1, 10000);

int servoAngle = 0;
int dc_motor_speed = 0;
int dc_motor_angle = 0;
double cm = 0.0;

int PotControlFlag = 0;
volatile int PotVal = 0;
volatile int globalStepperValue = 0;
volatile int globalStepperAngle = 0;

// Debounce variables
volatile unsigned long lastDebounceTime = 0;
const unsigned long debounceDelay = 100;
const int incrementServo = 30;

bool isMoving = false;
uint32_t targetPosition = 0;

void setup() {
  SERIAL_PORT.begin(9600);
  Serial1.begin(38400);
  while (!SERIAL_PORT) {
    // Wait for the serial port to be ready
  }

  // IMU
  if (!mpu.begin()) {
    Serial.println("Failed to find MPU6050 chip");
    while (1)
      ;
```

```
  }

  // Servo Motor
  servo.attach(servoPin);
  servo.write(0);

  // Push Button
  attachInterrupt(digitalPinToInterrupt(PushbuttonPin),
  pushISR, FALLING);

  // DC Motor
  unsigned long startTime = millis();

  // Stepper
  pinMode(EN_StepperDriver, OUTPUT);
  digitalWrite(EN_StepperDriver, LOW);  // enable stepper(s)
  stepper.setMaxSpeed(2000);
  stepper.setAcceleration(1000);
  stepper.setSpeed(0);

  roboclaw.SetM1VelocityPID(ADDRESS, Kp_vel,
  Ki_vel, Kd_vel, MAX_QPPS);
  roboclaw.SetM1PositionPID(ADDRESS, Kp_pos,
  Ki_pos, Kd_pos, Kp_vel, Ki_vel, Kd_vel, MAX_QPPS);

  roboclaw.SpeedM1(ADDRESS, 0);
  // Serial.println("Motor stopped at startup.");

  SERIAL_PORT.print("Arduino Due Serial is ready!;");
}


void loop() {
  if (isMoving) {
    uint32_t currentPos = roboclaw.ReadEncM1(ADDRESS);
    if (abs((int32_t)(currentPos - targetPosition)) <= 15)
    {  // Position tolerance
      roboclaw.SpeedM1(ADDRESS, 0);
      // Serial.println("Target position reached.");
      isMoving = false;  // Reset movement flag
    }
  }

  // Check if data is available to read from the serial port
  if (SERIAL_PORT.available() > 0) {
    // Read the incoming string
    String receivedString = SERIAL_PORT.readStringUntil(';');

    // // Echo the string back to the serial port
    // SERIAL_PORT.print("Arduino received command: ");
```

```
    // SERIAL_PORT.print(receivedString);
    // SERIAL_PORT.print(";");
    // SERIAL_PORT.flush();

    if (receivedString.length() > 1) {
      char commandType = receivedString.charAt(0);

      String valueString = receivedString.substring(1);
      if (isNumeric(valueString)) {
        int commandValue = valueString.toInt();
        handleCommand(commandType, commandValue);
      }
    }
  }

  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;
    timerCallback();
  }

  stepperCallback();
}

void timerCallback() {
  // SERIAL_PORT.print("Timer callback executed at: ");
  // SERIAL_PORT.print(previousMillis);
  // SERIAL_PORT.print(";");

  // A "$" is used to indicate the serial port return is a command


  String serialReturn;
  serialReturn.concat("$");

  int servoMotorState = servoMotorStateCallback();
  serialReturn.concat(servoMotorState);
  serialReturn.concat(",");

  int stepperMotorState = stepperMotorStateCallback();
  serialReturn.concat(stepperMotorState);
  serialReturn.concat(",");

  int velDCMotorState = velDCMotorStateCallback();
  serialReturn.concat(velDCMotorState);
  serialReturn.concat(",");

  int angleDCMotorState = angleDCMotorStateCallback();
  serialReturn.concat(angleDCMotorState);
  serialReturn.concat(",");
```

```
  int potentiometerSensorState = potentiometerSensorCallback();
  serialReturn.concat(potentiometerSensorState);
  serialReturn.concat(",");

  double imuSensorState = imuSensorCallback();
  serialReturn.concat(imuSensorState);
  serialReturn.concat(",");

  double temperatureSensorState = temperatureSensorCallback();
  serialReturn.concat(temperatureSensorState);
  serialReturn.concat(",");

  int ultrasonicSensorState = ultrasonicSensorCallback();
  serialReturn.concat(ultrasonicSensorState);
  serialReturn.concat(",");

  double electricalInput = analogRead(ultrasonicPin);
  double transferFunctionState =
  transferFunctionCallback(electricalInput);
  double electricalVoltage =
  analogRead(ultrasonicPin) * (5.0 / 1023.0);

  serialReturn.concat(electricalVoltage);
  serialReturn.concat(":");
  serialReturn.concat(transferFunctionState);

  SERIAL_PORT.print(serialReturn);
  SERIAL_PORT.print(";");
}


void handleCommand(char commandType, int value) {
  switch (commandType) {
    case 'R':
      servoMotorController(value);
      break;
    case 'S':
      stepperMotorController(value);
      break;
    case 'V':
      velDCMotorController(value);
      break;
    case 'A':
      angleDCMotorController(value);
      break;
    case 'B':
      buttonStepperMotorController(value);
      break;
    default:
```

```
      break;
    }
  }


bool isNumeric(String str) {
  if (str.length() == 0) return false;

  int startIndex = 0;

  if (str[0] == '-') {
    if (str.length() == 1) return false;
    startIndex = 1;
  }

  for (unsigned int i = startIndex; i < str.length(); i++) {
    if (!isDigit(str[i])) {
      return false;
    }
  }

  return true;
}


// CONTROLLER FUNCTIONS

void servoMotorController(int control) {
  /*
  INPUT: Integer in min/max range of 0 to 180
  corresponding to desired angle
  OUTPUT: Void
  */

  SERIAL_PORT.print("Servo motor controller
  received command: ");
  SERIAL_PORT.print(control);
  SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  servoAngle = control;
  servo.write(servoAngle);
}

void stepperMotorController(int control) {
  /*
  INPUT: Integer in min/max range of -180 to 180
  corresponding to desired angle
  OUTPUT: Void
  */
```

```
  SERIAL_PORT.print("Stepper motor controller
  received command: ");
  SERIAL_PORT.print(control);
  SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  if (PotControlFlag == 0) {
    globalStepperValue = map(control, -180, 180, -1600, 1600);
    globalStepperAngle = control;
  }
}

void velDCMotorController(int control) {
  /*
  INPUT: Integer in min/max range of -118 to 118
  corresponding to desired RPM
  OUTPUT: Void
  */

  SERIAL_PORT.print("Velocity DC motor controller
  received command: ");
  SERIAL_PORT.print(control);
  SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  // Set motor velocity (positive for forward,
  negative for reverse)
  int dc_motor_speed = (control / 118.0) * 100.0;
  int encoderSpeed = (dc_motor_speed * MAX_QPPS) / 100;
  roboclaw.SpeedM1(ADDRESS, encoderSpeed);
}

void angleDCMotorController(int control) {
  /*
  INPUT: Integer in min/max range of -360 to 360
  corresponding to desired angle
  OUTPUT: Void
  */

  // SERIAL_PORT.print("Angle DC motor controller
  received command: ");
  // SERIAL_PORT.print(control);
  // SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  uint32_t currentPos = roboclaw.ReadEncM1(ADDRESS);
  targetPosition = currentPos + (control * DEGREE_TO_TICKS);

  // Serial.print("Moving motor to position: ");
```

```
  // Serial.println(targetPosition);

  roboclaw.SpeedAccelDeccelPositionM1(ADDRESS,
  10000, MAX_QPPS, 10000, targetPosition, 0);

  isMoving = true;  // Set flag for movement tracking
}

void buttonStepperMotorController(int control) {


  SERIAL_PORT.print("Button controller received command: ");
  SERIAL_PORT.print(control);
  SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  PotControlFlag = control;
}


// CALLBACK FUNCTIONS
int servoMotorStateCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer in min/max range of 0 to 180 corresponding
  to servo motor angle
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  return servoAngle;
}

int stepperMotorStateCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer in min/max range of -180 to 180
  corresponding to stepper motor angle
  */

  // TODO: IMPLEMENT FUNCTION BELOW

  return globalStepperAngle;
}

int velDCMotorStateCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer in min/max range of -118 to 118
  corresponding to DC motor RPM
  */
```

```
  // TODO: IMPLEMENT FUNCTION BELOW

  int speed = roboclaw.ReadSpeedM1(ADDRESS);
  speed = speed * 118 / 6718;

  return speed;
}

int angleDCMotorStateCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer in min/max range of -360 to 360
  corresponding to DC motor angle
  */

  // TODO: IMPLEMENT FUNCTION BELOW

  int enc = roboclaw.ReadEncM1(ADDRESS);
  int angle = (enc * 360) / 3416;
  angle = angle % 360;

  return angle;
}

int potentiometerSensorCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer corresponding to potentiometer reading
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  PotVal = analogRead(PotentiometerPin);

  if (PotControlFlag == 1) {
    globalStepperValue = map(PotVal, 0, 1022, -1600, 1600);
    globalStepperAngle = map(globalStepperValue, -1600,
    1600, -180, 180);
  }

  return PotVal;
}

double imuSensorCallback() {
  /*
  INPUT: Void
  OUTPUT: Double corresponding to sensed IMU pitch reading
  */

  // TODO: IMPLEMENT FUNCTION BELOW
```

```
  sensors_event_t a, g, temp;
  mpu.getEvent(&a, &g, &temp);

  // Pitch using accel data
  double pitchAccel = atan2(a.acceleration.x,
  a.acceleration.z) * 180 / PI;

  return pitchAccel;
}

double temperatureSensorCallback() {
  /*
  INPUT: Void
  OUTPUT: Double corresponding to temperature
  reading (degree celsius)
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  int reading = analogRead(temperaturePin);
  double voltage = reading * 3.3;
  voltage /= 1024.0;

  double temperatureC = (voltage - 0.5) * 100;

  return temperatureC;
}

int ultrasonicSensorCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer corresponding to ultrasonic reading (cm)
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  cm = analogRead(ultrasonicPin);
  cm = transferFunctionCallback(cm);

  return int(cm);
}

// OTHER FUNCTIONS
void pushISR() {
  /*
  INPUT: Void
  OUTPUT: Void
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  unsigned long currentTime = millis();
  if ((currentTime - lastDebounceTime) > debounceDelay) {
```

```
    servoAngle += incrementServo;
    if (servoAngle > 180) {
      servoAngle = 0;
    }
    lastDebounceTime = currentTime;
    servo.write(servoAngle);
  }
}

double transferFunctionCallback(double electricalInput) {
  /*
  INPUT: Double corresponding to electrical input voltage
  OUTPUT: Double corresponding to ultrasonic reading (cm)
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  double val = electricalInput * 0.498 * 2.54;

  return val;
}

void stepperCallback() {
  /*
  INPUT: Void
  OUTPUT: Void
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  stepper.moveTo(globalStepperValue);
  stepper.run();
}
```