Checkpoint #3 Report

[EECN30169] Mobile Robot 2024

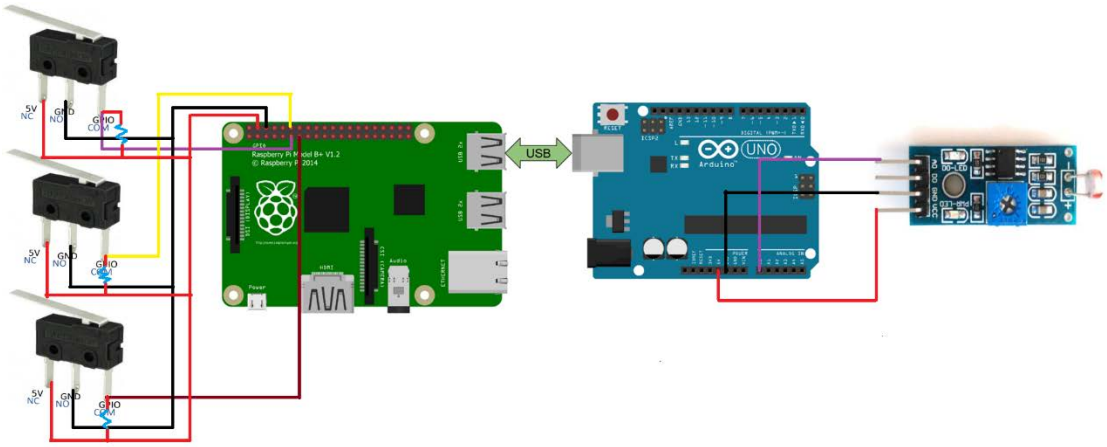
Student ID: 313605019 **Name:** 方敏 **Date:** **2024/11/1**

1. Purpose:

The purpose of this checkpoint is to ensure that we can control the robot to move within the arena. The mobile robot must be able to detect obstacle in front of it and take actions to avoid it, allowing it to continue moving until it finds the light ball.

2. Description of Design:

Wiring:

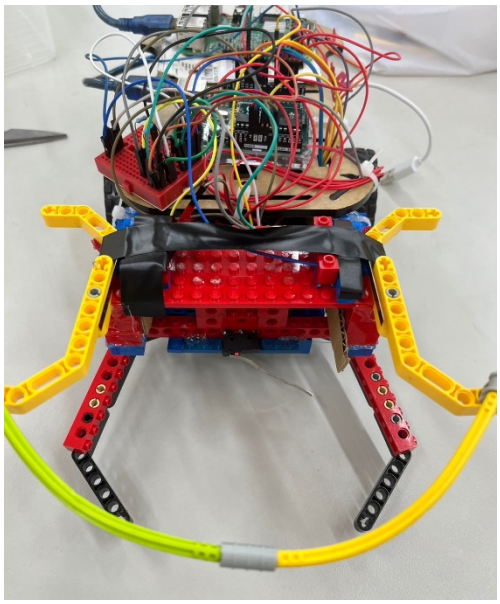


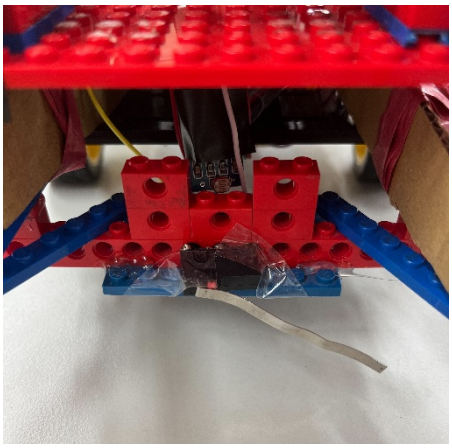
This implementation uses a total of four sensors: three touch sensors and one light sensor. We connected the touch sensors to GPIO pins 0~2 of the Raspberry Pi and the analog output of the light sensor to the A0 pin of the Arduino. The reason for this setup is that there are few remaining pin slots on the Arduino, while many unused pin slots are available on the Raspberry Pi. However, since the GPIO on the Raspberry Pi can only perform digital read and write operations, this greatly limits the accuracy of the light sensor's measurements. Therefore, we decided to connect the light sensor to the Arduino, which uses an ADC (generates a continuous range of voltage and converts into a digital value ranging from 0 to 1023) for easier processing in subsequent programs.

When the touch sensor is not in contact with any object, the GPIO pin remains at a high voltage (1). Once the touch sensor touches an object, the GPIO pin switches from high to low voltage (0), allowing us to determine if the touch sensor is activated. The light sensor outputs a voltage ranging from 0V to 5V based on the

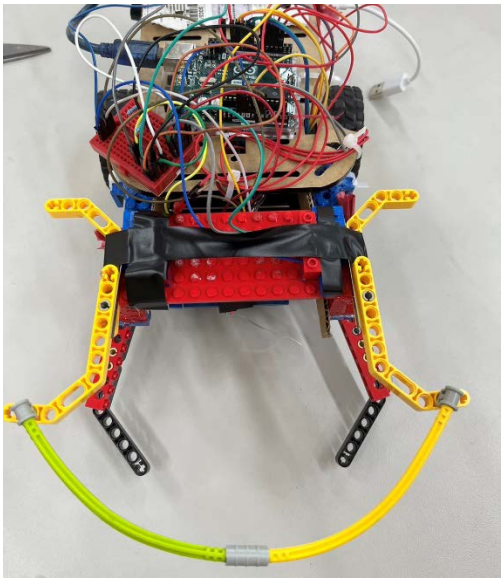
brightness of the environment, which is then converted to a value between 0 and 1023 using the Arduino's ADC. We use this value to determine the brightness of the environment: the smaller the value, the brighter the environment, while larger values indicate a darker environment. As shown in the picture above.

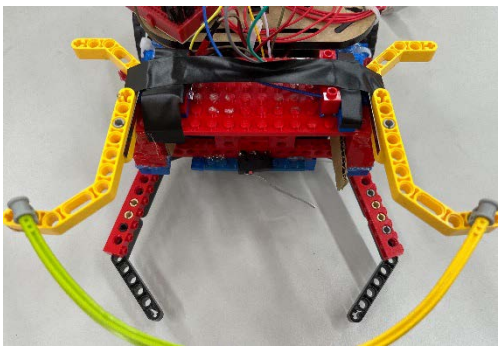
Assembling:



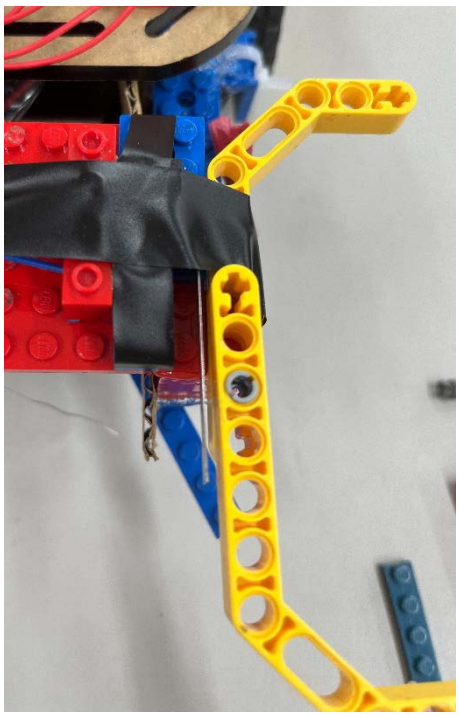
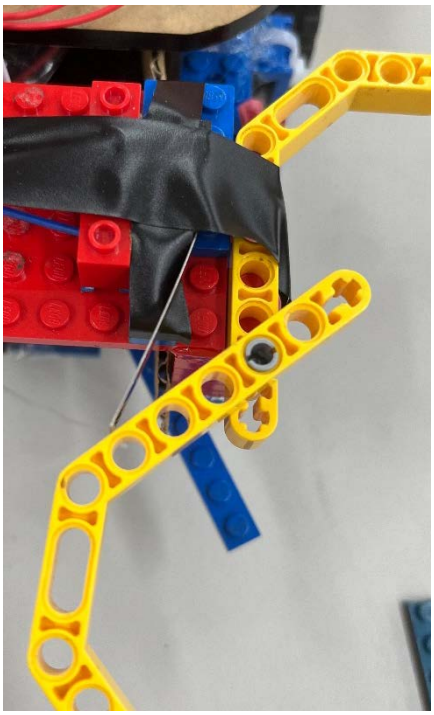
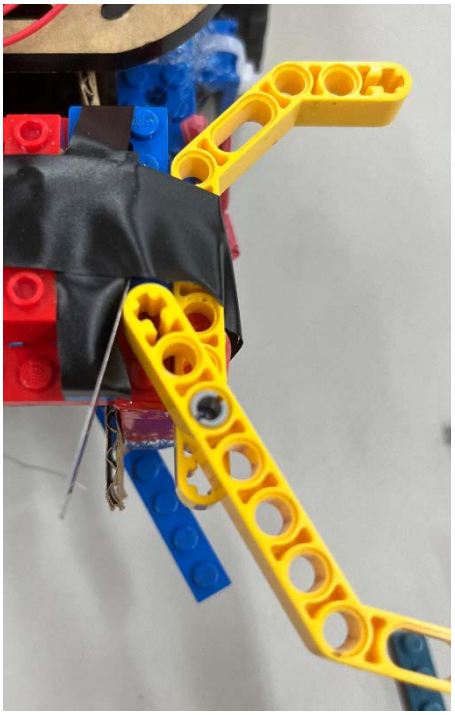


Having discussed the wiring layout, let's move on to the structure of the vehicle. From the front view, it appears that there's only one touch sensor at the lower part of the vehicle. However, upon closer inspection, you'll notice a light sensor cleverly concealed. This design choice was made because environmental light sources can be quite complex. We try our best to block any non-target light sources, and since the light sensor itself has an LED that emits light, we covered the LED with electrical tape. The touch sensor at the lower part of the vehicle serves a straightforward purpose: to detect if the light ball has been captured.





Viewed from above, two touch sensors are visible, used for detecting obstacles in front. The area that makes contact with obstacles is a flexible tube, which is connected at both ends to L-shaped Lego pieces. The advantage of using this setup is that it increases the contact area and ensures that both touch sensors are triggered simultaneously. A gray Lego pin is used to secure the axle, allowing it to rotate freely.

We are proud of this design because, whether contact is made from directly in front or at an angle, it can accurately transmit the information to the touch sensor, allowing the sensor to provide correct feedback, as shown in the picture above.

Coding:

Arduino:

On the Arduino side, the program hasn't changed much compared to cp2. The only difference is that the publisher outputs one additional value from the light sensor, and the rest is consistent with cp2.

```
// Output system info
if (output_counter == OUTPUT_DELAY) {
  output_counter = 0;
  output_msg.data[0] = analogRead(light);
  output_msg.data[1] = encoderA_accumulation;
  output_msg.data[2] = encoderB_accumulation;
  output_pub.publish(&output_msg);
}
```

Raspberry Pi:

We decided to use Python to deal with the complex programming environment. The program receives three sets of data from the Arduino approximately every 50 milliseconds via ROS. The first set of data comes from the light sensor, and the second and third sets come from the sum of the left and right encoder values. In order to maintain efficiency, the subscriber callback function merely stores data.

```
// Output system info
if (output_counter == OUTPUT_DELAY) {
  output_counter = 0;
  output_msg.data[0] = analogRead(light);
  output_msg.data[1] = encoderA_accumulation;
  output_msg.data[2] = encoderB_accumulation;
  output_pub.publish(&output_msg);
}
```

```
####### ROS function #######
# Subscriber callback function
def callback(dataset):
    global subscribe_data
    subscribe_data_queue.append([int(dataset.data[0]), int(dataset.data[1]), int(dataset.data[2])])
```

Due to compatibility issues or lack of support for wiringpi, and the fact that we are using a newer version of Raspberry Pi OS, wiringpi may not work properly. Therefore, we decided to use RPi.GPIO instead.

```
import RPi.GPIO as GPIO
```

The main part of the program consists of two loops: publisher_func, read_gpio. We will introduce these in order.

```
# Start the publisher function as a thread
threading.Thread(target=publisher_func).start()
threading.Thread(target=read_gpio).start()
```

The publisher_func is responsible for publishing messages with the topic of "control" to the Arduino for motor control. Whenever there are values in the global variable "publish_queue", the publisher_func will publish messages.

```
# Publisher loop function
def publisher_func():
    pub = rospy.Publisher('control', Int16MultiArray, queue_size=10)
    rate = rospy.Rate(1)  # 1Hz

    while not rospy.is_shutdown():
        if publish_queue:
            msg = Int16MultiArray()
            data = publish_queue.pop(0)
            msg.data.append(data['left'])
            msg.data.append(data['right'])
            pub.publish(msg)
            rate.sleep()
        else:
            time.sleep(0.01)
```

The read_gpio is responsible for handling the touch sensors connected to the Raspberry Pi's GPIO. It can update the status of the touch sensors in real-time and store it in the "sensor[ ]" variable.

```
####### Loop #######
# Read GPIO
def read_gpio():
    print("=start read_gpio")
    global state, sensor
    # Init GPIO
    GPIO.setmode(GPIO.BCM)
    # Set GPIO pin mode
    pins = [0, 1, 2]  # GPIO0 到 GPIO2
    for pin in pins:
        GPIO.setup(pin, GPIO.IN)

    # Save GPIO0 ~ GPIO2 to sensor[]
    while True:
        for idx, pin in enumerate(pins):
            sensor[idx] = GPIO.input(pin)
        time.sleep(0.01)
```

Our code is relatively straightforward because we know the initial position of the robot in advance. We first let the robot move forward until it hits a wall, then it moves backward and enters the **right_and_go** function. The robot then turns right and proceeds towards the brightest area. If sensor[0] == 0, it means the light ball has been detected, and the program ends. If sensor[1] == 0, it means the light ball has

not been captured and the robot has touched the wall, so it jumps to the **right_and_go_again** function. The robot turns right again and moves towards the brightest area. If sensor[0] == 0, the program ends. If sensor[1] == 0, it enters the **right_and_go_again** function once more until the function ends.

```
def right_and_go():
    print("=start right turn and go towards brightest")
    global state, publish_queue, max_light, subscribe_data, subscribe_data_queue
    state = "right_and_go"

    # Perform a right turn
    publish_queue = [{'left': rotate_speed, 'right': -rotate_speed}]
    time.sleep(2.6)  # Perform right turn for a while to change direction

    # Go straight after turning right
    publish_queue = [{'left': speed, 'right': speed}]

    # Continue moving straight towards brightest light
    while state == "right_and_go":
        if sensor[0] == 0:
            finish()
        elif sensor[1] == 0:
            back(0.1)
            right_and_go_again()
            return
        elif subscribe_data_queue:
            subscribe_data = min(subscribe_data_queue, key=lambda x: x[0])
            subscribe_data_queue = []
            # Adjust direction based on light sensor reading
            if subscribe_data[0] < max_light:
                publish_queue = [{'left': speed, 'right': speed}]
            else:
                publish_queue = [{'left': rotate_speed, 'right': -rotate_speed}]
        time.sleep(0.01)
```

```
def right_and_go_again():
    print("=start right turn and go again towards brightest!")
    global state, publish_queue, max_light, subscribe_data, subscribe_data_queue
    state = "right_and_go_again"

    # Perform a right turn
    publish_queue = [{'left': rotate_speed, 'right': -rotate_speed}]
    time.sleep(2.4)  # Perform right turn for a while to change direction

    # Go straight after turning right
    publish_queue = [{'left': speed, 'right': speed}]

    # Continue moving straight towards brightest light
    while state == "right_and_go_again":
        if sensor[0] == 0:
            finish()
        elif sensor[1] == 0:
            back(0.1)
            right_and_go_again()
            return
        elif subscribe_data_queue:
            subscribe_data = min(subscribe_data_queue, key=lambda x: x[0])
            subscribe_data_queue = []
            # Adjust direction based on light sensor reading
            if subscribe_data[0] < max_light:
                publish_queue = [{'left': speed, 'right': speed}]
            else:
                publish_queue = [{'left': rotate_speed, 'right': -rotate_speed}]
        time.sleep(0.01)
```

3. Result

In our first demo, the robot missed the light ball and had to go another round, taking 49 seconds. In the second attempt, the light ball was captured in the first round, so it only took 13 seconds.

Discussion

Problem 1:

Why did our code not use sensor[2]?

Explanation 1 :

After multiple tests, we found that one of the upper touch sensors was broken, and none of the remaining touch sensors in the classroom allowed us to connect the wires properly, so we used only sensor[1] to determine whether an obstacle or wall was hit. This is also why we used the Lego flexible tube.

Problem 2:

Why did we tape over the light sensor?

Explanation 2:

Initially, we found that the robot kept spinning in circles. Later, we discovered that this was because the light sensor's built-in LED was too bright, so we covered it with electrical tape to prevent interference.

In this checkpoint, it was much more challenging compared to the previous one, and I spent a lot of time debugging. I learned a great deal about hardware control and software integration for robots. I am grateful for the guidance from the professor and teaching assistants. I hope that the next checkpoint will proceed more smoothly.