Vanderbilt University
School of Engineering

ECE4375 Embedded Systems

Marco Polo Car
Technical Manual

Submitted by: Joseph Quatela, Lisa Liu, and Patrick Chen

Date of Submission: April 28, 2024

Table of Contents

Principle of Operation

The Marco Polo Car Project aimed to have two predator cars chase one prey car using only echolocation to locate the prey while avoiding obstacles. The prey car reveals its location by emitting a sound through a buzzer, which is picked up by microphones connected to the predator cars. The predators use this sound to turn towards the prey and continue moving forward. At that point, the predators will continue moving forward unless an obstacle is encountered. In this case, an object avoidance algorithm is employed to dodge the obstacle and continue chasing the prey. The basic logic for the game is outlined in Figure 1.

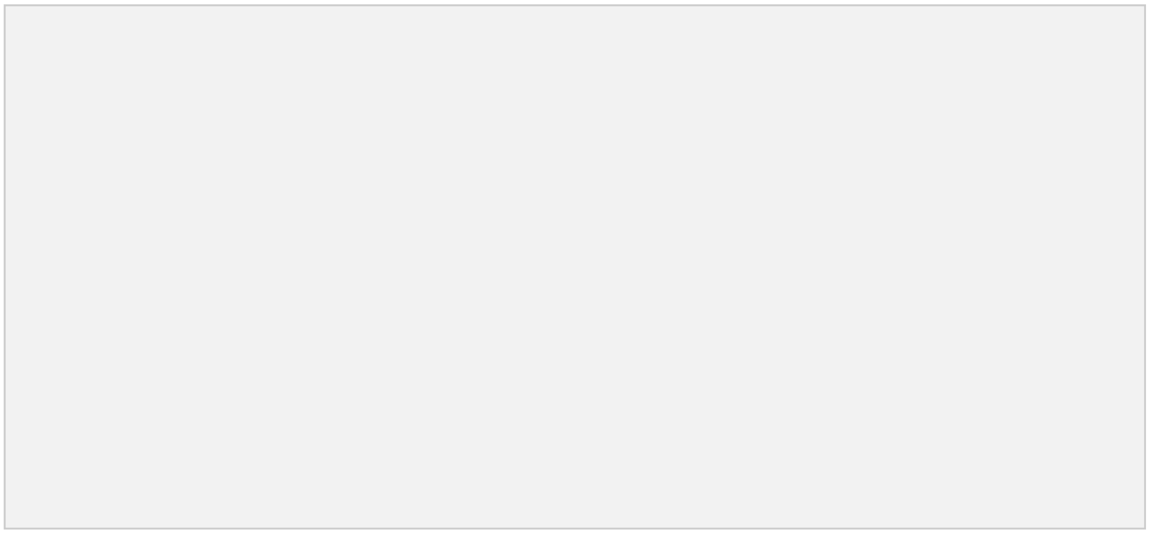

Figure 1. Marco Polo abstract car logic.

1. Predator: Object Avoidance

The onboard ultrasonic sensor sweeper was utilized to detect obstacles in front of the car and make decisions.

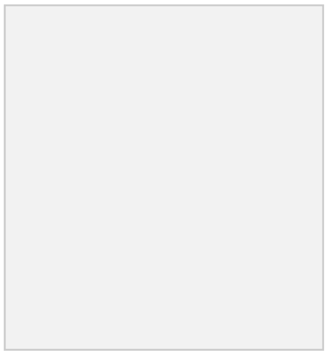

Figure 1.1 Ultrasonic Sensor Sweeper consisting of Ultrasonic Sensor and Servo Motors

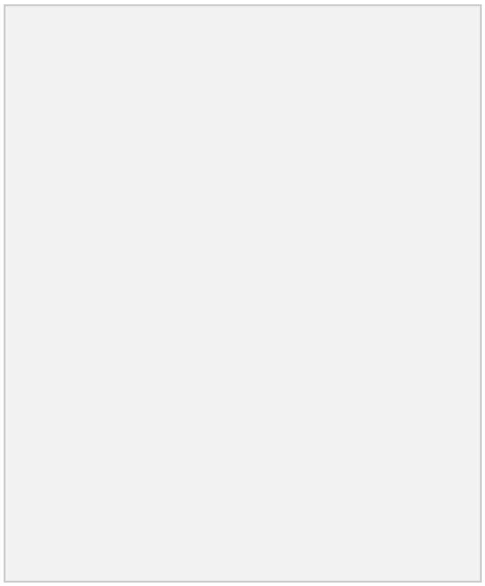

Figure 1.2 Ultrasonic Sensor Sampling Angle Groups

In the design, the ultrasonic sensor was moved by the servo motors so it could sweep the region in front of the car. To sample the distance most conservatively and accurately, a strategy of using a minimum of a set of distances was employed.

More specifically, the ultrasonic sensor was designed to provide distance information at seven different angles during each sweep. These angles are grouped into three categories: left, middle, and right. The sampling angles of 23, 45, 68, 90, 113, 135, and 158 degrees are then grouped into L, M, and R categories as displayed in the table below:

Figure 1.3 Angle Grouping

In each sweeping iteration, the smallest value in each of the L, M, and R sets is chosen to represent the distance between the car and the obstacle in three areas: the left of the car, the middle of the car, and the right of the car. This allows for an effective assessment of the surroundings during navigation and enhances safety by using the most conservative values.

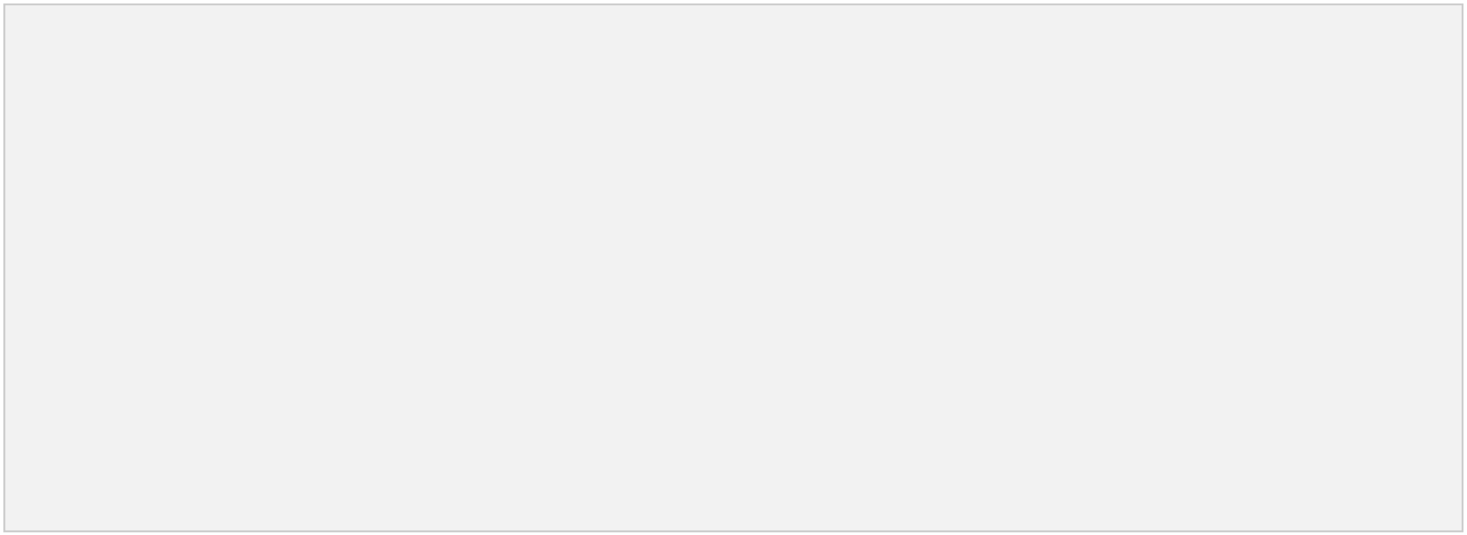

Figure 1.4 State Machine Diagram for Object Avoidance

The above state diagram shows the decision process in object avoidance. The car first locks the mutex and sets the conditional variable to ensure exclusive access to the motors. Then, it completes a scan to obtain the L, M, and R values and uses them to determine the action the car needs to take next. It then sets the motor rpm values, unsets the conditional variable, notifies all threads looking to gain access to motors, and unlocks the mutex.

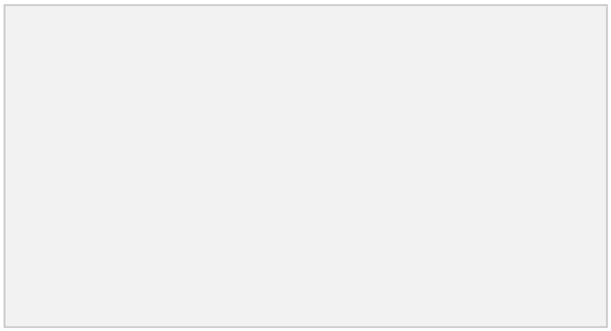

Figure 1.5 Mapping of L, M, R to Object Avoidance Case

The control uses rule-based logic to determine which state the car should enter. Intuitively, L, M, and R's threshold should be set lower for the "Close Up" case. Through intensive testing, the threshold set in Figure 1.5 performed the best, as when an object in all three directions is detected, there has to be a wall ahead that requires backing up.

2. Predator: Echolocation

An external microphone, the SeeedStudio ReSpeaker Mic Array V2.0, was used to receive the sound emitted by the prey car.

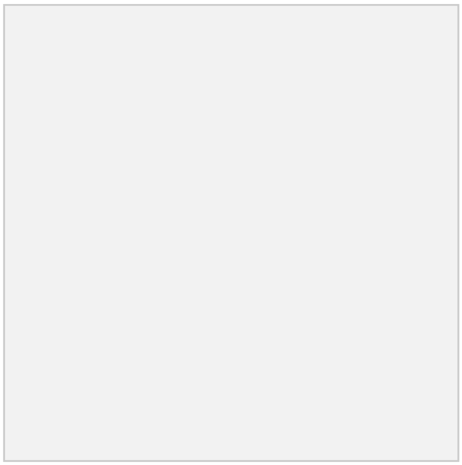

Figure 2.1. The SeeedStudio ReSpeaker Mic Array V2.0.

The microphone comes out of the box with the ability to detect the angle of arrival of a sound and visually show that angle using the LEDs on the microphone. This can easily be made into a script that outputs the angle as an integer. Thus, with an integer value of the angle available, it was only necessary to determine how to turn the car to that angle. This happens in 2 steps: calibration and turning.

Calibration
When the car initially turns on, the user will be prompted to calibrate it. On a key press, the car will begin turning in place in one direction. The user then stops the car once it has turned 360 degrees. This will then be repeated in the other direction to account for differences in turning time. Internally, the car starts and stops a timer while it is turning to generate a *time / 360* conversion value.

Calculation and Turning
With the calibrated conversion factor, the car can determine how long it will take to turn to any given angle using *Equation 1*.

$$time\ turning\ =\ conversion\ factor\ *\ target\ angle \tag{1}$$

With the angle available from the microphone and a way to turn to that angle, the two parts then had to be integrated to form a functioning unit test. The overall control flow is as follows: on each loop, the microphone would listen for a sound. If a sound was detected, the angle was saved from the microphone and then used to calculate the time needed to turn towards that sound. Then, the motors were activated to turn for that amount of time. The code for this logic is summarized in Figure 2.2.

```
def Turn_To_Angle():
calibrate_angles()
        init_mic()
        While (True):
                listen_for_sound()
                if (sound):
                        save_angle()
                        time = calculate_time()
                        turn()
                        sleep(time)
                        stop_motors()
```

Figure 2.2. Code segment demonstrating turn-to-angle logic.

Mic Initialization

The microphone array chosen for this project came out of the box with many features and no required drivers, which made it an appealing option. One of the adjustable features is the dB threshold at which the mic will register a sound, which was useful for this project. During mic initialization, the threshold must be calibrated to an appropriate level. Unfortunately, the frequency detection of the microphone is not easily adjustable and severely limits the performance of this project, as the microphone is tuned for voice detection, but a buzzer was used in this case.

3. Predator: Threading

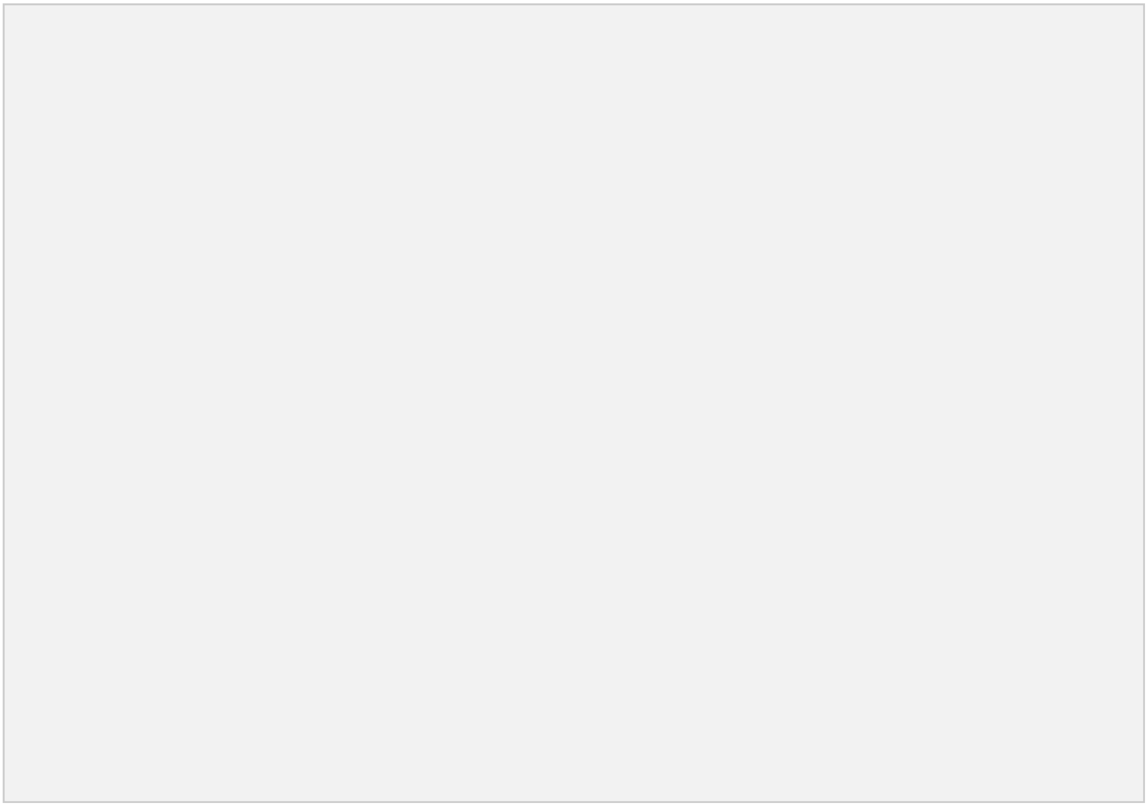

Figure 3.1 Use Case Diagram Showing Using Geared Motor is the Critical Section

The design requires concurrent operation of Object Avoidance and Echolocation. Threading provides the most timely response when Echolocation wants to take over. To ensure exclusive access to the Geared Motors that run the wheels, a conditional variable-based approach was proposed to implement this idea.

Generally, the two threads work by focusing on the state of two conditional variables: “object_avoidance_running” and “sound_detected”. Here, the simplified code for two threads will be shown to explain the task-switching design.

```
def object_avoidance_thread():
    # Setting up...
    while True:
        # Scan Left to Right
        with mutex:
            while sound_detected:
                wait_sound.wait() # Block
            object_avoidance_running = True
            # Scan for Obstacle
            # Move motors accordingly
            object_avoidance_running = False
            wait_OV.notify()

        # Repeat but scan Right to Left
```

Figure 3.2 Object Avoidance Thread

Before each iteration of the object avoidance code, the thread first checks the conditional variable “sound_detected” with a mutex. “sound_detected” is a variable that is manipulated in the echolocation thread. If the echolocation thread wants to take over the control of the motors, it will set “sound_detected” to be true. Then, the object avoidance thread will be blocked at the “wait()” statement until being notified by the Echolocation thread. When the echolocation thread is finished working with motors, it will set “sound_detected” to false and notify the object avoidance thread. Here, the while loop will be broken and the object avoidance will take control of the motor. At this step, it will change the state of the conditional variable “object_avoidance_running” to ensure exclusive access to the motor. This thread will have the control of motos for most of the time.

```
def echolocation_thread():
    # Set up Echolocation
    while True:
        # Waiting for Publisher's Message
        sound_detected = True
        with mutex:
            while object_avoidance_running:
                wait_OV.wait()
            # Stop Motors from Object Avoidance
            time.sleep(1)
            # Listening for Sound (While Loop)
            # Turn to the sound's direction
            # Stop the motor
            sound_detected = False
            sound_waiting = False
            wait_sound.notify_all()
```

Figure 3.3 Echolocation Thread

Similar to the Object Avoidance Thread, Echolocation Thread’s operation also relies on a conditional variable called “object_avoidance_running”. Essentially, when it receives a message from the prey, it would change the conditional variable “sound_detected” in an attempt to take over the control immediately after the object avoidance thread finishes its current iteration. Once it exists the blocking state, it will run the appropriate echolocation methods and reset all conditional variables associated with itself. Then, it notifies all threads, which allows object avoidance to continue its operation.

It is noted that from the classroom discussion, it was said that threading priority is not controllable in Python due to how they are implemented using a global interpreter lock (GIL). The closest thing that could be done here is to use conditional variables to ensure the higher priority task would be guaranteed to be executed immediately after the current thread's iteration is over.

4. Prey: Remote Control and Buzzer

The prey car uses the buzzer and motor objects provided by the Freenove smart car in order to control the wheels and buzzer of the car based on the user input. The program runs an infinite loop waiting for the user to input commands into the terminal to control the car with Python's `input()` function. The characters 'w,' 'a,' 's,' 'd,' when entered into the command line, will configure the car to move forward, right, stop, and left, respectively. The character 'b,' when entered into the command line, will cause the buzzer on the prey car to buzz. Due to the microphone's limitations, as described in the following section, this command will publish a message to the subscribers listening to the IP address and port of the publisher. The subscribers, the predator cars, will then know to listen to the buzzing sound. After the message is broadcast, the buzzer will emit the sound for half a second.

5. Prey - Predator: Networking

The microphone's functionality is compromised by its inability to filter out the operational noise of the motors and servos. Consequently, if the microphone was configured to continually monitor the buzzer, it would consistently capture the operating sound of motors and servos, triggering the echolocation process and impeding the car's mobility.

To address this issue, the following procedure is proposed:

1. Predator: Object Avoidance is Running
    - Ensure that the Predator module responsible for object avoidance is actively functioning.
2. Prey: Initiate Buzzer Sound
    - The Prey module intends to emit a buzzer sound, but due to the operational noise, the Predator may not perceive it clearly.
3. Prey: Send Network Message to Predator
    - Send a network message to the Predator module, indicating the intention to emit a sound.

4. Predator: Receive Network Message
   - Upon receiving the network message, the Predator module acknowledges it and initiates the echolocation process by acquiring the mutex, halting all movements.
5. Prey: Emit Sound
   - The Prey emits the intended sound signal.
6. Predator: Orient Towards Prey
   - Utilizing the echolocation data, the Predator module adjusts its orientation towards the source of the emitted sound, i.e., the Prey.
7. Predator: Resume Object Avoidance
   - After completing the orientation adjustment, the Predator module reverts to its Object Avoidance mode to continue navigating its environment safely.

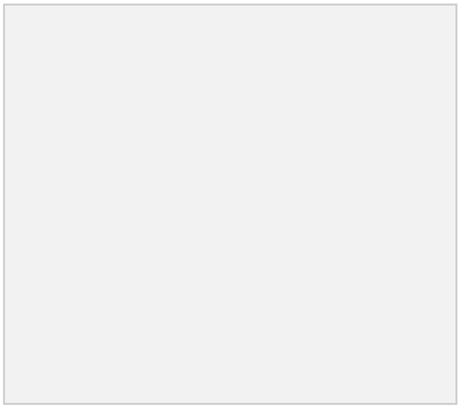

Figure 5.1 Pub-Sub Structure

To allow multiple predators in the game, a Pub-Sub mode of the ZMQ socket was used. Pub-Sub allows a server to send one-way messages to multiple subscribers. In this design, the prey serves as the server and the predator as the subscribers.

Results and Discussion

The networking solution, however, was not fail-proof. The main issue, that the microphone is not sensitive to the frequency of the sound of the buzzer, was unable to be resolved. Even with the predator car stopping to listen for the buzzer sound, it would, at times, fail to register the sound and turn to the correct angle. The threshold of sound on the microphone had to be configured to be extremely low to attempt to pick up the buzzer sound. This made it hard to test, as the microphone was incredibly sensitive to other noise coming from the environment. In the end, the networking solution worked well during one run with one of the predator cars as seen in this video. The echolocation capability is exhibited in this video, where the car is turning to a clapping sound to which it is more sensitive.

In conclusion, the success of the original vision for this project was hindered by the microphone's low sensitivity to the buzzer sound. Further work can be done to investigate different microphones that would be more sensitive to the specific frequency that the smart car buzzer emits.

Appendix

Code Repository:
https://github.com/lisaliuu/rpi-marco-polo

Open Source Library Used:
For controlling the car's peripherals:
https://github.com/Freenove/Freenove_4WD_Smart_Car_Kit_for_Raspberry_Pi

For interfacing with the microphone:
https://wiki.seeedstudio.com/ReSpeaker_Mic_Array_v2.0/

Personnel

Joseph Quatela: Developed the Calibration code, Turn To Angle code, and Echolocation Thread. Researched and planned the microphone interface for the project. Designed, 3D printed, and iterated on the microphone mounts for the cars.

Lisa Liu: Developed the code controlling the prey car to emit a buzzer sound and move. Helped debug threading issues and optimal microphone threshold limit.

Patrick Chen: Developed & Optimized Object Avoidance Algorithm, Echolocation & Object Avoidance Threading (Merge all tasks), and Networking Communication (Pub-Sub). Wrote the part of the report and manual associated with these parts.