Turtlebot3 Burger - Autonomous Driving

Mauricio Álvarez
Department of Computer Science
UTEC
mauricio.alvarez@utec.edu.pe

Jose Chachi
Department of Computer Science
UTEC
jose.chachi@utec.edu.pe

Leonardo Candio
Department of Computer Science
UTEC
leonardo.candio@utec.edu.pe

Abstract

This report explores the implementation of a TurtleBot3 Burger robot using ROS 2, detailing its hardware and software specifications, manual and autonomous control methods and reinforcement learning-based training. The TurtleBot3 Burger features a Raspberry Pi 4, OpenCR1.0 controller, and LDS-02 LiDAR, enabling precise navigation and real-time obstacle detection. ROS 2 provides a modular framework for communication and control. Manual teleoperation is achieved using a keyboard, laying the groundwork for autonomous navigation. Reinforcement learning is then employed to train the robot in simulated environments, enabling it to learn optimal navigation strategies for real-world deployment. This work demonstrates the integration of robotic hardware and AI techniques for autonomous mobile robotics.

1 Introduction

In this report, we present the development of an autonomous driving project using the TurtleBot3 Burger. Our goal is to implement both manual and autonomous driving functionalities leveraging Python and ROS 2, while utilizing the full capabilities of the TurtleBot3 Burger, such as the LDS-02 LiDAR and the OpenCR1.0 controller. The report aims to provide a detailed explanation of the implementation process and its underlying methodology.

The structure of the report is as follows: we begin by discussing the motivation behind this project and reviewing related works that have successfully achieved autonomous driving. The second section outlines the hardware and software specifications of the TurtleBot3 Burger. In the third section, we provide a step-by-step guide for configuring the robot, following the official ROBOTIS documentation [3]. Sections four and five detail the implementation of manual driving and autonomous driving, respectively. Finally, section six concludes the report with a summary of findings and potential directions for future work.

Report for the CS5364-2024.2 - Deep Learning Class at UTEC. Lima, Peru.

1.1 Motivation

The primary motivation for this project is to develop a TurtleBot3 Burger capable of real-time obstacle detection and navigation, a critical feature for safe and efficient autonomous driving. Beyond this technical goal, we aim to prepare the robot for participation in a TurtleBot3 Burger race competition, where it must interact with other robots in a dynamic environment. This scenario closely mirrors real-world applications, such as autonomous vehicles sharing roads with human drivers and other self-driving systems, highlighting the importance of adaptability and precise navigation in complex settings.

Extending this concept to real-world challenges, autonomous driving in crowded and chaotic cities like Lima presents significant difficulties. The streets are often filled with street vendors occupying sidewalks and roads, while citizens frequently disregard traffic signals, adding unpredictability to the environment. Public transportation further complicates the scenario, with buses stopping arbitrarily—even in the middle of the road—to pick up or drop off passengers. These factors create highly dynamic and unstructured conditions, making navigation a formidable task for autonomous systems.

1.2 Related work

Autonomous driving has been widely explored using TurtleBot platforms due to their versatility and integration with ROS. In the study [1] focused on the TurtleBot3 Waffle Pi, aiming to enable autonomous navigation on a custom-built racing rig designed with traffic lights, signs, tunnels, and obstacles. Using tools like TensorFlow and ROS, reinforcement learning was applied to improve obstacle avoidance and autonomous driving adherence to traffic rules. The research emphasized SLAM and navigation tuning for optimal performance in both simulation and real-world tests.

Another study [2] was implemented an intelligent navigation system on a TurtleBot equipped with a rotating Kinect sensor, enabling it to navigate U-shaped paths. The research focused on enhancing mobility through real-time path planning and SLAM, utilizing a Kalman filter to address localization and mapping challenges. This work provided insights into navigation in forward and reverse directions, revealing a 5%-6% increase in deviation when navigating in reverse, and laid the foundation for future advancements in wheelchair navigation systems.

Together, these works demonstrate the potential of TurtleBot platforms for autonomous navigation, highlighting advancements in real-time obstacle detection, path planning, and reinforcement learning.

2 Technical Specifications

The TurtleBot3 Burger have the following hardware specifications:

- **Performance:**
  - Maximum translational velocity: 0.22 m/s
  - Maximum rotational velocity: 2.84 rad/s (162.72 deg/s)
  - Maximum payload: 15 kg
- **Physical Dimensions:**
  - Size (L x W x H): 138 mm x 178 mm x 192 mm
  - Weight (with SBC + Battery + Sensors): 1 kg
  - Threshold of climbing: 10 mm or lower
- **Battery and Power:**
  - Battery: Lithium polymer 11.1V 1800mAh / 19.98Wh 5C
  - Expected operating time: 2h 30m
  - Expected charging time: 2h 30m
  - Power connectors:
    - 3.3V / 800mA
    - 5V / 4A
    - 12V / 1.3A

- **Core Components:**
  - SBC (Single Board Computer): Raspberry Pi
  - MCU: 32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
  - Actuator: XL430-W250
  - LDS (Laser Distance Sensor): 360° LDS-01 or LDS-02
  - IMU: Gyroscope (3-axis) and Accelerometer (3-axis)
- **Connectivity:**
  - UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
  - DYNAMIXEL ports: RS485 x3, TTL x3
  - Expansion pins:
    - GPIO: 18 pins
    - Arduino: 32 pins
- **Additional Features:**
  - Audio: Several programmable beep sequences
  - Programmable LEDs: User LED x4
  - Status LEDs:
    - Board status LED x1
    - Arduino LED x1
    - Power LED x1
  - Buttons and switches: Push buttons x2, Reset button x1, Dip switch x2
  - Firmware upgrade: via USB / via JTAG
- **Power Adapter (SMPS):**
  - Input: 100-240V AC 50/60Hz, 1.5A @max
  - Output: 12V DC, 5A

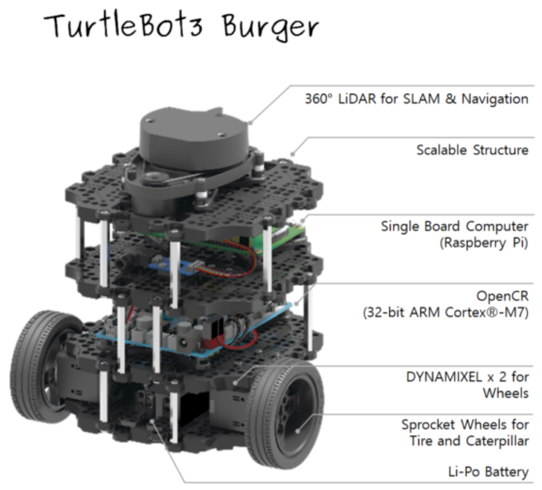

TurtleBot3 Burger
360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (32-bit ARM Cortex®-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery

Figure 1: Quick view of Turtlebot3 Burger components

3 Preliminary configuration

3.1 ROS2 Image Humble

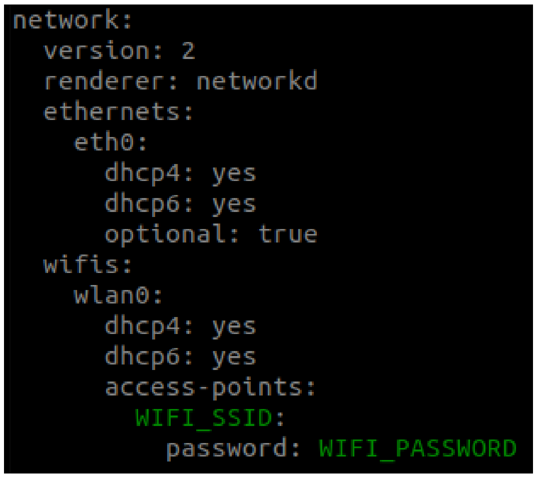
The provided Raspberry Pi is initially empty, so we will install the ROS 2 Humble image on it. To begin, ensure access to an application called *Raspberry Pi Imager* on a compatible device. This application is essential for installing Ubuntu Server 22.04 on the Raspberry Pi. You'll also need a microSD card and a card reader to connect the Raspberry Pi to the device. Once the *Raspberry Pi Imager* is launched, select the operating system as "Other general-purpose OS" and download Ubuntu 22.04 onto the microSD card as the storage medium. After the installation is complete, insert the microSD card into the Raspberry Pi on the robot. This setup allows us to access the Raspberry Pi's terminal, enabling the subsequent configuration of the robot. To gain access and visualize the terminal, ensure to connect the HDMI port of the Raspberry Pi to a monitor.

3.2 Internet Connection

One important configuration is enabling access to internet to the Raspberry Pi. To do so, we will use the terminal and we will open the network configuration as follows:

```
$ sudo nano /writable/etc/netplan/50-cloud-init.yaml
```

When the editor is opened, edit the content with your username and password.



```
network:
  version: 2
  renderer: networkd
  ethernets:
    eth0:
      dhcp4: yes
      dhcp6: yes
      optional: true
  wifis:
    wlan0:
      dhcp4: yes
      dhcp6: yes
      access-points:
        WIFI_SSID:
          password: WIFI_PASSWORD
```

Figure 2: Network configuration

You can now save the file with Ctrl+S and exit with Ctrl+X

At the implementation, we reboot the raspberry Pi and changed the password to *turtlebot*

3.3 Connection through SSH

To make the connections from one device to the TurtleBot3, we followed these lines:

```
$ ssh {HOSTNAME}@{BURGER_IP}
```

SSH may be deactivated by default. You may use the following commands:

```
$ sudo service ssh start
$ sudo ufw allow ssh
```

After this, a message will appear as security. We enter 'yes' and enter. Then we enter the password for our TurtleBot (default is turtlebot).

Steps shown in this section came from the oficial guide. For more information, some graphical guides or install another version of ROS2, we hardly suggest to view the following link [3]: https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/#pc-setup

4 Manual Driving

4.1 Bringup

Bringup is essential to run at the beginning of the project. If done, we can omit this step, instead we should open a new terminal and connect with our Turtlebot3 Burger. Then we should use the following lines:

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

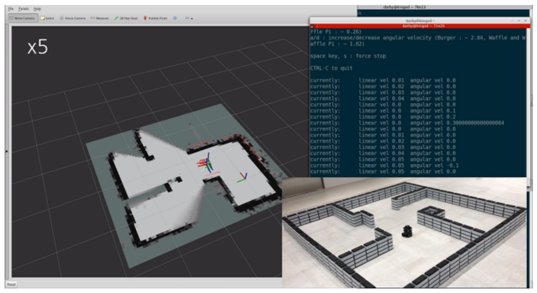
If done correctly and using the Turtlebot3 model Burger, we should see a log as a summary.

4.2 Run SLAM Node

Once Bringup is done correctly and we are connected to our TurtleBot3 Burger via terminal, we will run the following lines:

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_cartographer cartographer.launch.py
```

After that, we should see the following screens:



```
ffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and W
affle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit

currently:      linear vel 0.01  angular vel 0.0
currently:      linear vel 0.02  angular vel 0.0
currently:      linear vel 0.03  angular vel 0.0
currently:      linear vel 0.04  angular vel 0.0
currently:      linear vel 0.0   angular vel 0.0
currently:      linear vel 0.0   angular vel 0.1
currently:      linear vel 0.0   angular vel 0.2
currently:      linear vel 0.0   angular vel 0.30000000000000004
currently:      linear vel 0.0   angular vel 0.0
currently:      linear vel 0.01  angular vel 0.0
currently:      linear vel 0.02  angular vel 0.0
currently:      linear vel 0.03  angular vel 0.0
currently:      linear vel 0.04  angular vel 0.0
currently:      linear vel 0.05  angular vel 0.0
currently:      linear vel 0.05  angular vel -0.1
currently:      linear vel 0.05  angular vel 0.0
```

Figure 3: SLAM screen, Movement such as linear and angular speed screen, Real world image (does not pop up)

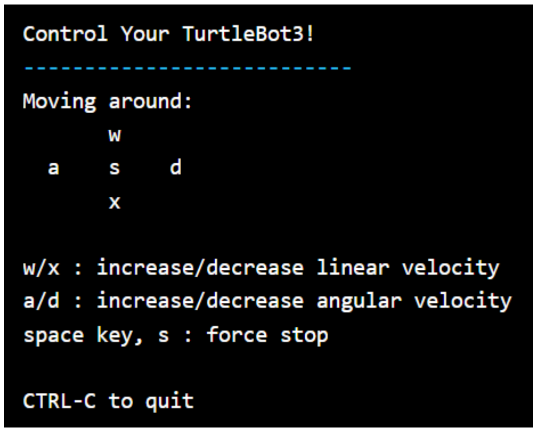
4.3 Run Teleoperation Node

To update the SLAM map and move the Turtlebot3, we will be exploring unknown area of the map using teleoperation movement.

We are going to run the following scripts:

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 run turtlebot3_teleop teleop_keyboard
```

We should see the following lines in our terminal:



```
Control Your TurtleBot3!
---------------------------
Moving around:
       w
  a    s    d
       x

w/x : increase/decrease linear velocity
a/d : increase/decrease angular velocity
space key, s : force stop

CTRL-C to quit
```

Figure 4: Terminal showing the teleoperation Node

Keybinds ’w’ and ’x’ may be used as to move forward at high speed or slow speed. Keybinds ’a’ and ’d’ may be used as to move left or move right by turning to the left or right the wheels, respectively. Keybinds ’s’ or space key is to stop any movement from the Turtlebot3 Burger.

5 Autonomous Driving

Reinforcement learning was initially considered for the project, as it offers a powerful framework for training agents to navigate complex environments. However, due to the lack of access to GPU resources, the training process became prohibitively time-consuming. Without the computational acceleration provided by GPUs, the time required to converge to an optimal policy exceeded practical limits. Consequently, we opted for a more traditional algorithmic approach, leveraging domain knowledge and deterministic strategies such as PID control and obstacle-avoidance logic. This approach, while less adaptive than reinforcement learning, allowed us to achieve reliable and efficient navigation within the available computational constraints.

This code can be seen in apendix A and is a ROS2-based robot controller for autonomous navigation. It uses LiDAR data to sense the environment and employs a PID controller to adjust the robot’s linear and angular velocities for tasks such as wall-following, obstacle avoidance, and navigating towards specific markers. The RobotController class subscribes to LiDAR sensor data, processes it to identify obstacles or safe zones, and uses a PID-based feedback loop to generate velocity commands. The system considers multiple sectors of the robot’s surroundings to dynamically adjust its path, ensuring safe and efficient navigation. Advanced preprocessing of LiDAR data and configurable PID parameters enable robust and adaptive behavior in varied environments.

6 Conclusions

In conclusion, this project presented significant challenges, particularly in implementation and configuration. Initially, setting up the TurtleBot was complex due to the absence of pre-installed packages, requiring considerable effort to configure essential functionalities. Establishing communication between the TurtleBot and the remote PC also posed obstacles, as we explored multiple methods before achieving a stable connection.

The autonomous navigation segment was the most demanding aspect of the project. Testing and fine-tuning various algorithms often resulted in unexpected behavior, with the robot sometimes failing to respond correctly to commands. Despite these setbacks, we successfully resolved the issues, enabling the robot to move both via teleoperation and autonomously. Leveraging its LiDAR sensor, the TurtleBot was able to detect its surroundings and navigate autonomously, showcasing the practical integration of deep learning and robotics for real-time decision-making and path planning. This experience demonstrated the potential of autonomous systems and provided valuable insights into their practical implementation.

6.1 Future work

For future work, we aim to enhance the capabilities of the TurtleBot by integrating computer vision techniques to detect traffic signals and other road elements. By utilizing object detection models such as YOLO (You Only Look Once), the robot could identify and classify objects in real time, improving its decision-making process in complex environments. Additionally, incorporating advanced path-planning algorithms and multi-sensor fusion could enable more efficient navigation in dynamic scenarios. Another area of exploration includes optimizing reinforcement learning models to better handle diverse and unpredictable conditions, such as those found in crowded or obstacle-rich environments. These improvements could extend the robot's applications to tasks like navigating urban-like settings, participating in competitive racing tracks, and serving as a platform for more advanced autonomous vehicle research.

References

[1] Sindre Haugseter. Autonomous driving and machine learning with turtlebot3 waffle pi mobile rover. Master's thesis, University of South-Eastern Norway, 2023.

[2] Ankit Kumar, Kamred Udham Singh, Pankaj Dadheech, Aditi Sharma, Ahmed I Alutaibi, Ahed Abugabah, and Arwa Mohsen Alawajy. Enhanced route navigation control system for turtlebot using human-assisted mobility and 3-d slam optimization. *Heliyon*, 10(5), 2024.

[3] RoboTiS. Turtlebot3 overview. `https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`, 2024. Accessed: 05-Dec-2024.

A Appendix

```
# ROS2 module imports
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
from rclpy.qos import QoSProfile, QoSReliabilityPolicy, QoSHistoryPolicy
from rclpy.qos import qos_profile_sensor_data
from rclpy.duration import Duration

# Python module imports
import numpy as np
import queue
import time


class PIDController:
    def __init__(self, kP, kI, kD, kS):
        self.kP = kP  # Proportional gain
        self.kI = kI  # Integral gain
        self.kD = kD  # Derivative gain
        self.kS = kS  # Saturation constant (error history buffer size)
        self.err_int = 0  # Error integral
        self.err_dif = 0  # Error difference
        self.err_prev = 0  # Previous error
        self.err_hist = queue.Queue(self.kS)
# Limited buffer of error history
        self.t_prev = 0  # Previous time

    def control(self, err, t):
        dt = t - self.t_prev  # Timestep
        if dt > 0.0:
            self.err_hist.put(err)  # Update error history
            self.err_int += err  # Integrate error
            if self.err_hist.full():  # Jacketing logic to prevent integral windup
                self.err_int -= self.err_hist.get()
# Rolling FIFO buffer
            self.err_dif = err - self.err_prev  # Error difference
            u = (
                (self.kP * err)
                + (self.kI * self.err_int * dt)
                + (self.kD * self.err_dif / dt)
            )  # PID control law
            self.err_prev = err  # Update previous error term
            self.t_prev = t  # Update timestamp
            return u  # Control signal
        return 0


class RobotController(Node):
    def __init__(self):
        # Information and debugging
        info = "\nMake␣the␣robot␣follow␣wall,␣avoid␣obstacles,␣follow␣line,␣detect␣s
        print(info)

        # ROS2 infrastructure
        super().__init__("robot_controller")

        qos_profile = QoSProfile(
            reliability=QoSReliabilityPolicy.RMW_QOS_POLICY_RELIABILITY_RELIABLE,
            history=QoSHistoryPolicy.RMW_QOS_POLICY_HISTORY_KEEP_LAST,
            depth=10,
        )

        self.robot_lidar_sub = self.create_subscription(
            LaserScan, "/scan", self.robot_lidar_callback, qos_profile_sensor_data
        )

        self.robot_ctrl_pub = self.create_publisher(Twist, "/cmd_vel", qos_profile)

        timer_period = 0.001  # Node execution time period (seconds)
        self.timer = self.create_timer(timer_period, self.robot_controller_callback)

        # PID Controllers
        self.pid_1_lat = PIDController(0.3, 0.01, 0.1, 10)
        self.pid_1_lon = PIDController(0.1, 0.001, 0.005, 10)

        # State variables
        self.lidar_available = False
        self.laserscan = None
        self.start_mode = "outside"
        self.start_time = self.get_clock().now()
        self.ctrl_msg = Twist()
        self.prefer_left_turns = True

    def robot_lidar_callback(self, msg):
        # Robust LIDAR data preprocessing
        ranges = np.array(msg.ranges)

        # Replace NaN and inf with maximum sensor range
        ranges = np.nan_to_num(ranges, nan=3.5, posinf=3.5, neginf=3.5)

        # Filter out extreme or invalid ranges
        ranges = np.clip(ranges, 0.1, 3.5)

        self.laserscan = ranges
        self.lidar_available = True

    def robot_controller_callback(self):
        DELAY = 4.0  # Time delay (s)
        if self.get_clock().now() - self.start_time > Duration(seconds=DELAY):
            if self.lidar_available and self.laserscan is not None:
                # Dynamically calculate scan parameters
                scan_length = len(self.laserscan)
                step_size = 360.0 / scan_length

                # Safe sector calculation
                front_sector = max(1, int(30 / step_size))
# Wider front sector
                side_sector = max(1, int(45 / step_size))
# Wider side sector

                # Robust distance calculations
                def safe_mean(arr):
                    valid_ranges = arr[np.isfinite(arr)]
                    return np.mean(valid_ranges) if len(valid_ranges) > 0 else 3.5

                # More granular obstacle detection
                front_left = safe_mean(self.laserscan[: int(scan_length / 4)])
                front_right = safe_mean(self.laserscan[-int(scan_length / 4) :])
                front_center = safe_mean(
                    np.concatenate(
                        [self.laserscan[:front_sector], self.laserscan[-front_sector
                    )
                )

                # Additional side and diagonal sectors
                left_side = safe_mean(
                    self.laserscan[int(scan_length / 4) : int(scan_length / 2)]
                )
                right_side = safe_mean(
                    self.laserscan[int(scan_length / 2) : int(3 * scan_length / 4)]
                )

                # Timestamp for PID
                tstamp = time.time()

                # More aggressive obstacle detection thresholds
                FRONT_OBSTACLE_THRESHOLD = 0.7
# Larger threshold for front
                SIDE_OBSTACLE_THRESHOLD = 0.5
# Smaller threshold for sides

                # Debugging print statements
                print(
                    f"Distances␣-␣Front:␣{front_center:.2f},␣Left:␣{front_left:.2f},␣
                )

                # Advanced Obstacle Avoidance Logic
                if (
                    front_center < FRONT_OBSTACLE_THRESHOLD
                    or front_left < FRONT_OBSTACLE_THRESHOLD
                    or front_right < FRONT_OBSTACLE_THRESHOLD
                ):
                    # Obstacle directly in front or on sides
                    if (front_left > front_right) == self.prefer_left_turns:
                        # Turn left if more space on left (and preferring left) or m
                        LIN_VEL = 0.05  # Very slow forward movement
                        ANG_VEL = 1.2  # Strong left turn
                        print("OBSTACLE␣AHEAD:␣Turning␣Left")
                    else:
                        # Turn right if more space on right (and preferring left) or
                        LIN_VEL = 0.05  # Very slow forward movement
                        ANG_VEL = -1.2  # Strong right turn
                        print("OBSTACLE␣AHEAD:␣Turning␣Right")

                elif (
                    left_side < SIDE_OBSTACLE_THRESHOLD
                    or right_side < SIDE_OBSTACLE_THRESHOLD
                ):
                    # Obstacles on sides
                    if left_side < SIDE_OBSTACLE_THRESHOLD:
                        # Obstacle on left, turn right
                        LIN_VEL = 0.1
                        ANG_VEL = -0.8
                        print("SIDE␣OBSTACLE:␣Turning␣Right")
                    else:
                        # Obstacle on right, turn left
                        LIN_VEL = 0.1
                        ANG_VEL = 0.8
                        print("SIDE␣OBSTACLE:␣Turning␣Left")

                else:
                    # Normal wall following with more aggressive correction
                    LIN_VEL = 0.2
                    ANG_VEL = self.pid_1_lat.control(
                        (left_side - right_side) * 2.0,
# Increased sensitivity
                        tstamp,
                    )
                    print("Wall␣Following")

                # Velocity Limits
                self.ctrl_msg.linear.x = min(0.22, float(LIN_VEL))
                self.ctrl_msg.angular.z = min(2.84, float(ANG_VEL))

                # Publish control message
                self.robot_ctrl_pub.publish(self.ctrl_msg)
        else:
            print("Initializing...")


def main(args=None):
    rclpy.init(args=args)
    node = RobotController()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()


if __name__ == "__main__":
    main()
```