Shell Eco-marathon Autonomous Urban Concept System Plan

1. Introduction

This document outlines the system architecture and implementation plan for our team's autonomous Urban Concept vehicle participating in the Shell Eco-marathon. The goal is to design, build, and program a vehicle that can navigate a predefined track autonomously, demonstrating energy efficiency and adherence to competition rules, specifically navigating through distinct sections: empty track (lane following), obstacle avoidance, and a final parking area, with required stops at designated stop signs between sections. Due to constraints, the system will operate without GPS and wheel encoders, relying solely on onboard sensors for localization and environmental understanding.

2. System Architecture Overview

The autonomous system is designed using a modular approach based on the ROS2 framework. This allows for independent development and testing of components, promoting robustness and maintainability. The core modules are:

- **Sensors:** Collecting raw data from the environment and vehicle.
- **Perception:** Processing sensor data to understand the environment (lanes, obstacles, signs).
- **Localization:** Estimating the vehicle's position and orientation based on sensor data.
- **Planning and Decision Making:** Deciding the vehicle's actions based on its state, environment, and goals, managed by a state machine.
- **Control:** Executing the planned actions by commanding the vehicle's actuators.
- **Actuation Interfaces:** Communicating with the vehicle's hardware (motor, steering).

The system operates within defined coordinate frames managed by ROS2's `tf2` library.

3. Hardware Components

- **Computing Platform:** Jetson Orin AGX 16GB - Provides the processing power for sensor data processing, localization, planning, and control.
- **Camera:** For lane detection, stop sign detection, and potentially parking area identification.
- **2D LiDAR:** For obstacle detection and providing geometric features for localization (LiDAR Odometry).
- **IMU (Inertial Measurement Unit):** Provides angular velocity and linear acceleration data for localization (Inertial Odometry).
- **Motor and Motor Controller:** For controlling the vehicle's propulsion. Communicates via serial.
- **Steering Motor and Driver:** For controlling the steering angle.
- **Steering Rotary Encoder:** Provides feedback on the current steering angle.
- **Safety System:** Emergency stop button(s) and associated hardware/software cutoff.

4. Software Architecture (ROS2 Nodes and Topics)

The system is structured as a graph of ROS2 nodes communicating via topics.

- **Sensor Nodes:**
    - `camera_node` : Publishes `/camera/image_raw` (sensor_msgs/Image)
    - `lidar_node` : Publishes `/scan` (sensor_msgs/LaserScan)
    - `imu_node` : Publishes `/imu/data` (sensor_msgs/Imu)
    - `motor_controller_interface` : Subscribes to `/control/motor_commands` , publishes (optional) feedback.
    - `steering_interface` : Subscribes to `/control/steering_commands` , publishes `/sensor/steering_angle` .
- **Perception Nodes:**
    - `lane_detection_node` : Subscribes to `/camera/image_raw` , publishes `/perception/lane_detections` (Custom Message).
    - `obstacle_detection_node` : Subscribes to `/scan` , publishes `/perception/obstacles` (Custom Message/MarkerArray). (Includes LiDAR clustering).
    - `stop_sign_detection_node` : Subscribes to `/camera/image_raw` , publishes `/perception/stop_sign_detected` (std_msgs/Bool or Custom Message).
    - `parking_area_detection_node` : Subscribes to `/camera/image_raw` , `/scan` , publishes `/perception/parking_area_detected` (std_msgs/Bool or Custom Message).
- **Localization Nodes (Odometry Fusion):**
    - `lio_node` : Subscribes to `/scan` , `/imu/data` , publishes `/lio/odometry` (nav_msgs/Odometry). (2D LiDAR-Inertial Odometry).
    - `vins_node` (Optional): Subscribes to `/camera/image_raw` , `/imu/data` , publishes `/vins/odometry` (nav_msgs/Odometry). (Visual-Inertial Odometry).
    - `odometry_fusion_node` : (Using `robot_localization`) Subscribes to `/lio/odometry` , `/vins/odometry` (if available), `/imu/data` , publishes `/odometry/filtered` (nav_msgs/Odometry) and the `/tf` transform from `odom` to `base_link` .
- **Planning and Decision Making Nodes:**
    - `state_machine_node` : Subscribes to `/perception/stop_sign_detected` , `/perception/parking_area_detected` , `/odometry/filtered` , `/control/event` . Publishes `/state_machine/current_state` (Int8/enum) and potentially commands to the active planner. Manages state transitions based on events and progress.
        - **States:** Idle, Section1_LaneFollowing, ApproachingStopSign_Section1, Stopped_Section1, Section2_ObstacleAvoidance_LaneKeeping, ApproachingStopSign_Section2, Stopped_Section2, Section3_Parking, Parked, EmergencyStop.
    - `section1_planner_node` : Subscribes to `/odometry/filtered` , `/perception/lane_detections` , `/state_machine/current_state` . Publishes `/planning/cmd_vel` or `/planning/target_pose` when active. Focuses on lane following.
    - `section2_planner_node` : Subscribes to `/odometry/filtered` , `/perception/obstacles` , `/perception/lane_detections` , `/state_machine/current_state` . Publishes `/planning/cmd_vel` or `/planning/target_pose` when active. Focuses on constrained obstacle avoidance within lanes.
    - `section3_planner_node` : Subscribes to `/odometry/filtered` , `/perception/parking_area_detected` , `/perception/obstacles` (potentially), `/state_machine/current_state` . Publishes `/planning/cmd_vel` or `/planning/target_pose` when active. Focuses on the parking maneuver.
    - `stop_sign_handling_node` : (Can be integrated or separate) Subscribes to `/perception/stop_sign_detected` , `/odometry/filtered` , `/control/event` . Publishes stop commands and signals completion to the state machine.
- **Control Nodes:**
    - `longitudinal_controller_node` : Subscribes to `/planning/cmd_vel` (or speed target), `/odometry/filtered` (for speed), publishes `/control/motor_commands` , `/control/event` (stop signal).

    - `lateral_controller_node`: Subscribes to `/planning/cmd_vel` (or steering target), `/sensor/steering_angle`, `/odometry/filtered` (for heading/position), publishes `/control/steering_commands`.

5. Plan and Implementation Steps

This plan outlines the key development stages. It is recommended to work on modules in parallel where possible, but ensure clear interfaces are defined early on.

- **Phase 1: Hardware Interfaces and Basic Sensor Data (Weeks 1-X)**
    - Mount and connect all sensors (Camera, LiDAR, IMU) and actuators (Motor Controller, Steering Motor/Encoder) to the Jetson.
    - Install ROS2 on the Jetson.
    - Develop/find ROS2 nodes for each sensor to publish raw data as ROS2 topics. Test each sensor node independently.
    - Develop/find ROS2 nodes to interface with the motor controller (serial communication) and steering motor/encoder. Implement basic manual control (e.g., using ROS2 topics to command speed and steering) for testing.
    - Accurately calibrate the static transforms (`tf`) between `base_link` and each sensor (`lidar_frame`, `camera_frame`, `imu_frame`).
- **Phase 2: Core Perception (Weeks X-Y)**
    - Develop the `lane_detection_node`. Start with simpler methods (e.g., color filtering, line fitting) and progressively add robustness. Test with recorded camera data.
    - Develop the `obstacle_detection_node`. Implement LiDAR clustering. Test with recorded LiDAR data containing obstacles.
    - Develop the `stop_sign_detection_node`. Start with basic image processing or explore pre-trained models. Test with images of stop signs.
    - Start thinking about the `parking_area_detection_node` and how the parking area will be defined and detected.
- **Phase 3: Localization (Odometry Fusion) (Weeks Y-Z)**
    - Select and integrate a ROS2-compatible 2D LIO library (`lio_node`). Configure it for your 2D LiDAR and IMU.
    - Set up the `odometry_fusion_node` using `robot_localization` (or a similar package). Configure an EKF to fuse IMU and `/lio/odometry`.
    - (Optional but Recommended) Explore and integrate a ROS2 VIO library (`vins_node`). Add its output to the `odometry_fusion_node`.
    - Test localization accuracy by driving the vehicle manually on a representative surface and comparing the `/odometry/filtered` output in RViz to the actual path. Analyze drift.
    - Investigate techniques for re-initializing or correcting odometry at section transitions (stop signs).
- **Phase 4: Planning and Decision Making (Weeks Z-A)**
    - Develop the `state_machine_node`. Define the states and transitions. Implement the logic for switching between states based on perceived events and odometry.
    - Develop the `section1_planner_node` (Lane Following). Use the output of `lane_detection_node` to generate target steering/velocity commands.
    - Develop the `section2_planner_node` (Obstacle Avoidance and Lane Keeping). Implement a local planning algorithm that uses both obstacle detections from `obstacle_detection_node` and lane detections from `lane_detection_node`.
    - Develop the `section3_planner_node` (Parking). Implement the logic for detecting the parking spot and executing the maneuver.
    - Develop the `stop_sign_handling_node` logic for stopping and pausing.
- **Phase 5: Control (Weeks A-B)**
    - Develop the `longitudinal_controller_node` (e.g., PID controller for speed).
    - Develop the `lateral_controller_node` (e.g., PID or more advanced controller for steering angle, using encoder feedback).
    - Tune the controllers using manual control data or simulations.
- **Phase 6: Integration and Testing (Weeks B-C)**
    - Integrate all the developed nodes within the ROS2 framework.
    - Perform incremental testing:
        - Test perception nodes with live sensor data.
        - Test localization with all sensors running.
        - Test planners with simulated or recorded sensor/localization data.
        - Test controllers with the vehicle on blocks (if possible) or in a safe open area with manual planner inputs.
    - Begin testing the integrated system in a safe, open area, gradually introducing elements of the track (e.g., a straight lane, simple turns).
    - Test transitions between simplified sections (e.g., a lane following segment followed by a stop).
    - Test obstacle avoidance with static obstacles.
    - Test the parking maneuver.
    - Move testing to a track that closely resembles the competition environment.
    - Perform full end-to-end autonomous runs.
- **Phase 7: Refinement and Optimization (Weeks C-Competition)**
    - Identify and fix bugs found during testing.
    - Tune planning and control parameters for energy efficiency and robustness.
    - Refine perception algorithms for better accuracy.
    - Improve localization robustness to drift.
    - Practice runs on a simulated competition track.
    - Address any specific rules or requirements of the competition.

6. Key Challenges

- **Localization Accuracy without GPS and Wheel Encoders:** Minimizing drift through robust LIO (and VIO if implemented) and clever handling of section transitions.
- **Reliable Perception in Varying Conditions:** Ensuring lane, obstacle, and stop sign detection works in different lighting and track conditions.
- **Robust Transition Handling:** Accurately detecting stop signs and executing precise stops and restarts.
- **Constrained Obstacle Avoidance:** Safely navigating around obstacles while staying within lane boundaries in Section 2.
- **Accurate Parking:** Precisely maneuvering into the parking area.
- **Energy Efficiency:** Balancing autonomous operation with minimizing energy consumption.

7. Team Roles and Responsibilities

Define specific roles within your team corresponding to these modules (e.g., Perception Lead, Localization Lead, Planning Lead, Control Lead, Hardware Integration, Software Integration).

8. Communication and Collaboration

Establish clear communication channels (e.g., daily stand-ups, version control system like Git, project management tools) to ensure the team is coordinated and on track.

9. Testing and Validation

Develop a comprehensive testing plan, including unit tests, integration tests, and on-vehicle testing. Document test results and iterate based on findings. Utilize ROS2 tools like RViz and rqt for visualization and debugging.