Project Roadmap & Implementation Phases

Phase 1: Knowledge Acquisition & Environment Setup

- Concepts to Learn:
  - CANbus framing and arbitration.
  - CiA301 Protocol: Understand Network Management (NMT) , Service Data Objects (SDO) for configuration , Process Data Objects (PDO) for real-time data , and Heartbeats.
    +4

  - CiA402 Protocol: Understand the standard motion control state machine. * Implementation Steps:

1. Set up a Linux environment (or WSL on Windows). Linux natively supports CAN routing via SocketCAN.

2. Set up a virtual CAN interface (vcan0) to act as the simulated bus.
3. Initialize your Git repository.

Phase 2: Core CANopen Communication (Backend)

- Focus: Build the infrastructure to listen to and transmit CAN frames according to the ZLAC8015D's expected COB-IDs (Communication Object Identifiers).

- Implementation Steps:
  1. Node-ID Setup: Default the simulator to Node-ID 1 (COB-IDs offset by 1).

  2. Heartbeat Simulation: Implement a timer to broadcast a heartbeat message (COB-ID 0x700 + Node-ID) indicating the NMT state.
     +2

  3. NMT Handling: Listen for COB-ID 0x000 to transition the simulator between Pre-Operational, Operational, and Stopped states.
     +2

  4. SDO Server: Implement logic to read/write to the Object Dictionary when receiving SDO requests (COB-ID 0x600 + Node-ID) and send responses (COB-ID 0x580 + Node-ID).
     +1

Phase 3: Object Dictionary & CiA402 State Machine (Backend)

- Focus: The ZLAC8015D acts as a dual-motor controller (Left and Right wheels mapped to Sub-index 01 and 02). You need to simulate its memory and states.
  +3

- Implementation Steps:
  1. Object Dictionary (OD) Data Structure: Create a class or dictionary to store all values (e.g., 0x6040 Control Word, 0x6060 Mode of Operation, 0x607A Target Position).
     +2

  2. State Machine Logic: Implement the CiA402 state transitions (Switch On Disabled -> Ready to Switch On -> Switched On -> Operation Enable).

  3. Control/Status Mapping: Write logic so that when a user writes to the Control Word (0x6040), the internal state updates, and the Status Word (0x6041) reflects this change.
     +2

Phase 4: Motion Control Physics Simulation (Backend)

- Focus: Emulate the actual physical response of the motors based on the selected mode.

- Implementation Steps:
  1. Mode Switching: Read 0x6060 to switch between modes (1: Profile Position, 3: Profile Velocity, 4: Profile Torque).

  2. Profile Position Mode: Simulate movement to a Target_position (0x607A) using Profile_velocity (0x6081), Profile_acceleration (0x6083), and Profile_deceleration (0x6084). 3. Profile Velocity Mode: Simulate accelerating to a Target_velocity (0x60FF). Update internal position values over time using a simple physics loop ($Position += Velocity \times \Delta time$).
     +2

  3. Profile Torque Mode: Emulate achieving a Target_torque (0x6071) using a Torque_slope (0x6087).

Phase 5: Graphical User Interface (Frontend)

- Focus: Build an interactive dashboard to visualize what is happening inside the simulated controller.
  +1

- Implementation Steps:
  1. Status Panel: Display current NMT state, CiA402 state, Control Word, and Status Word in real-time.

  2. Telemetry Charts: Plot actual speed, position, and torque against target values to prove the physics simulation is working.

  3. CAN Log: Create a scrolling text area showing raw CAN frames sent and received.

  4. Control Panel: Add buttons for developers to manually inject commands (e.g., "Start", "Enable Operation", "Fault Reset") without needing an external script.

Phase 6: Integration, Testing & Documentation

- Focus: Prove the simulator can trick a master controller into thinking it's the real hardware.
- Implementation Steps:
  1. PDO Mapping: Ensure that TPDOs (Transmit Process Data Objects) automatically push updates like actual speed/position to the bus when in the "Operational" state. +1

  2. Validation: Use standard Linux tools like cansend and candump to interact with your simulator.

  3. Demonstration: Write a simple Python master script that commands the simulated robot to drive forward, turn, and stop, showing the UI reacting accurately.

Recommended File System Architecture

To keep your code organized, separate the CAN communication, the physical simulation, and the UI logic.

```
zlac8015d_simulator/
│
├── README.md                # Project overview and setup instructions
├── requirements.txt         # Python dependencies (python-can, PyQt5, etc.)
├── setup_vcan.sh            # Bash script to easily initialize vcan0
│
├── src/
│   ├── main.py              # Entry point: launches backend threads and UI
```

```
│   │
│   ├── backend/              # Core Simulator Logic
│   │   ├── __init__.py
│   │   ├── can_node.py       # SocketCAN interface, reads/writes raw frames
│   │   ├── sdo_pdo_handler.py # Parses/constructs CANopen SDOs and PDOs
│   │   ├── object_dict.py    # Data class holding registers (0x1000 - 0x60FF)
│   │   ├── state_machine.py  # CiA402 state transition logic (Control/Status words)
│   │   └── physics_sim.py    # Math loops calculating position/velocity/torque over time
│   │
│   └── frontend/             # User Interface
│       ├── __init__.py
│       ├── app_window.py     # Main PyQt/Web window assembly
│       ├── widgets/          # Modular UI components
│       │   ├── state_panel.py # Visualizes NMT and CiA402 states
│       │   ├── telemetry.py  # Graphs for Speed/Torque/Position
│       │   └── can_logger.py  # Scrolling log of CAN messages
│
├── tests/                    # Unit testing
│   ├── test_state_machine.py  # Ensures 0x6040 commands yield correct 0x6041 status
│   └── test_physics.py       # Validates S-curve/acceleration math
│
└── docs/                     # Documentation
    ├── manual/               # Store the provided PDF manuals here
    └── architecture.md       # Documentation on your OD usage and simulation architecture
```