

CERTIFICATE

Certified that **Aakrti 2300290140001, Akshat Gautam 23002901400005 , Anchal Tyagi 2300290140023 , Anuj Singh 2300290140029** has/ have carried out the project work having “**Automated Vehicle Control using Trampoline**” (**Project-KCA451**) for **Master of Computer Application** from Dr. A.P.J. Abdul Kalam Technical University (AKTU) (formerly UPTU), Lucknow under my supervision. The project report embodies original work, and studies are carried out by the student himself/herself and the contents of the project report do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Dr. Amit Kumar Gupta
Professor
Department of Computer Applications
KIET Group of Institutions, Ghaziabad

Dr. Akash Rajak
Dean
Department of Computer Applications
KIET Group of Institutions, Ghaziabad

Automated Vehicle Control using Trampoline

ABSTRACT

In the rapidly advancing field of automotive technology, integrating smart decision-making systems is crucial for ensuring safer and more efficient transportation. This project focuses on designing and implementing a robust decision-making architecture for autonomous vehicles using an OSEK/VDX-based Real-Time Operating System (RTOS) called Trampoline. The system processes inputs from multiple sensors, including proximity sensors, gyroscopic sensors, and pressure sensors, to assess the vehicle's surroundings and generate essential movement parameters: **Acceleration Intensity, Brake Intensity, and Steering Intensity.**

To enable seamless communication between different vehicle subsystems, we utilize the Controller Area Network (CAN) **protocol**. A CAN driver has been implemented for POSIX systems, using Linux Virtual CAN (vCAN) for hardware communication simulation. The proposed framework ensures real-time data processing, allowing vehicles to navigate autonomously while responding dynamically to road conditions. This project serves as a foundation for intelligent vehicular systems, contributing to the broader vision of autonomous and connected transportation

ACKNOWLEDGEMENTS

Success in life is never attained single-handedly. My deepest gratitude goes to my project supervisor, Prof. Mr.Amit Kumar Gupta for his guidance, help, and encouragement throughout my project work. Their enlightening ideas, comments, and suggestions.

Words are not enough to express my gratitude to Dr. Akash Rajak, Professor and Dean, Department of Computer Applications, for his insightful comments and administrative help on various occasions.

Fortunately, I have many understanding friends, who have helped me a lot on many critical conditions.

Finally, my sincere thanks go to my family members and all those who have directly and indirectly provided me with moral support and other kind of help. Without their support, completion of this work would not have been possible in time. They keep my life filled with enjoyment and happiness.

Aakrti

Akshat Gautam

Anchal Tyagi

Anuj Singh

TABLE OF CONTENTS

Certificate	i
Abstract	ii
Acknowledgements	iii
Table of Contents	iv
List of Tables & Figures	v
1 Introduction	1-3
1.1 Overview	1-2
1.2 Simulated Communication Model	2
1.5 Data Communication in Software Simulation	2-3
2 Literature Review	4-6
2.1 Introduction	4
2.2 Existing Autonomous Vehicle Practices	4
2.2.1 Sensor Fusion Techniques	4
2.2.2 Decision-Making Algorithm	4
2.2.3 Real-Time Operating Systems(RTOS)	5
2.3 Simulated Communication in Place of CAN protocol	6
2.4 Challenges in Current Approaches	6
3 Methodology	7-9
3.1 Overview	7
3.2 Tools And Technologies Used	8
3.3 Development Process	8-9
3.3.1 Requirement Gathering	8
3.3.2 Planning	9
3.3.3 Designing	9
3.3.4 Implementation	9
3.3.5 Testing	9
3.3.6 Integration	10
4 System Design	11-14
4.1 Overview	11
4.2 System Architecture	11-12
4.3 Key Modules And Components	12

4.3.1	Virtual Sensor Module	12
4.3.2	Inter-Task Communication Module	12
4.3.3	Decision-Making Module	13
4.3.4	RTOS Scheduler Module	13
4.3.5	Actuator Control Module	13
4.4	Workflow of Automated Vehicle Control System	13-14
4.4.1	Virtual Environment Sensing	13
4.4.2	Data Processing and Decision Making	13
4.4.3	Simulated Communication Between Tasks	14
4.4.4	Vehicle Actuators	14
5	Implementation	15-21
5.1	Overview	15
5.2	Development Requirements	15-17
5.3	Development Setup	17-18
5.3.1	Development & Testing Environment	17
5.3.2	Virtual Deployment Setup	18
5.3.3	Integration	18
5.3.4	Deployment Setup	18
5.4	Implementation Challenges	19
5.4.1	Real Time Constraint Handling	19
5.4.2	Virtual Data Sensor Accuracy	19
5.4.3	Communication Simulation	19
5.4.4	Synchronization Between Tasks	19
5.5	Future Plans	19-21
6	Testing	22-24
6.1	Testing Methodology	22
6.1.1	Unit Testing	22
6.1.2	Integration Testing	22
6.1.3	Hardware-in- the-loop(HIL) Testing	23
6.1.4	Safety And Fault Tolerance Testing	23
6.2	Test Cases	23-24
6.3	Results	24
6.4	Conclusion	24-25
7	Conclusion And Future Scope	26
7.1	Conclusion	26
7.2	Achievements	26
7.3	Challenges Faced	27
7.4	Future Scope	27
7.5	Lesson Learned	28
7.6	Final Thoughts	28
8	References	29-31

8.1	Development Tools And Documentation	29
8.2	Online Learning Resources and Tutorial	29
8.3	Design and Testing Tools	30
8.4	Deployment And Hosting Resources	30
8.5	UI/UX Design Resources	31
8.6	Case Implementation Resources	31
9	Appendix	32
9.1	Conceptual Architectural Diagram	32
9.2	Code	33-34
9.3	Project Snapshot	35

LIST OF FIGURES AND TABLES

Figure No.	Name of Figure/Table	Page No.
1.1	Communication Flow	2
3.1	Iterative Model	7
5.1	Autonomous Vehicle Control System Flow diagram	21
6.2	Test Cases	23
9.1	Conceptual Architecture Diagram	32
9.2	Code Snapshots	33-34
9.3	Project Snapshot	35

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

In the era of automation and intelligent systems, the need for efficient, safe, and responsive vehicle control strategies is rapidly increasing. While traditional hardware-based automotive systems focus on real-world sensor integration and physical actuation, simulation-based development provides a safer and more flexible environment for early-stage testing and validation.

The *Automated Vehicle Control using Trampoline* project introduces a virtual solution by utilizing **Trampoline RTOS**, a lightweight, OSEK/VDX-compliant real-time operating system. The system is designed to manage and execute critical vehicle control tasks—such as braking, steering, and acceleration—using **simulated sensor data** processed in a **POSIX environment**.

This software-only implementation emphasizes **task scheduling**, **priority-based execution**, and **real-time decision-making** through structured input files that mimic actual road scenarios. Trampoline’s modular design, efficient memory usage, and static task configuration make it highly suitable for simulation and academic use. The platform supports core functionalities like obstacle avoidance, lane monitoring, and safe navigation logic—validated entirely through a controlled, testable software model.

The core objectives of **Automated Vehicle Control using Trampoline** are:

- **Real-Time Performance:** Ensuring timely response to sensor inputs and environmental changes to enhance vehicle safety and reliability.
- **Modularity and Maintainability:** Enabling a clean architecture where different functionalities can be easily tested, updated, or replaced.

- **Scalability:** Supporting a range of vehicle types and complexities, from basic control to more advanced autonomous features.
- **Compliance:** Aligning with automotive standards like OSEK/VDX to ensure compatibility with industry practices and future development.

1.2 SIMULATED COMMUNICATION MODEL

In this simulation-based implementation, the communication model represents the **logical flow of data** between system components—executed entirely through software. Rather than using hardware communication protocols, data exchange between tasks is managed through **shared variables and scheduling** in Trampoline RTOS.

Key stages in this communication model include:

Sensor Input (Simulated) – Values read from structured input files representing proximity and environmental data

Task Scheduling – Managed by Trampoline RTOS, which prioritizes tasks based on criticality

Decision Making – Evaluates sensor values and determines output control signals

Virtual Actuation – Generates and logs intensity levels for braking, acceleration, or steering

Feedback Loop (Simulated) – Output may influence future input scenarios in a simulation

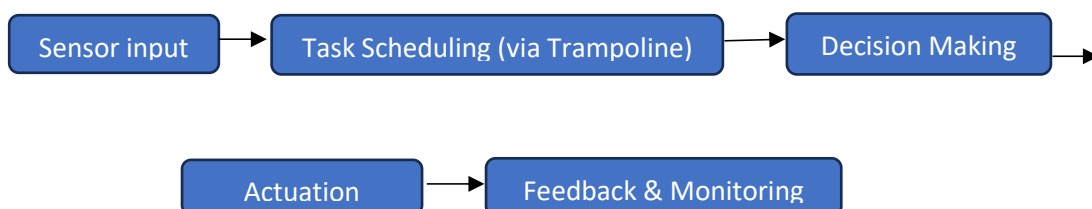


Fig 1.1 Communication Flow

1.3 DATA COMMUNICATION IN SOFTWARE SIMULATION

In this project, all data communication is performed **locally within the software**, without physical buses or external interfaces. The simulation substitutes real hardware communication with deterministic task execution and internal logic control.

1. Local Data Communication(Simulated)

- **Sensor-to-Control Task Communication:** Input data is parsed from files and passed to processing tasks (e.g., Brake, Steer, Accelerate).
- **Control-to-Actuator Logic:** Calculated results (e.g., To_Steer = -10) are stored in global variables and displayed as output.
- **Feedback Mechanisms:** Simulated feedback is introduced by modifying future input files or logic pathways based on previous decisions.

This communication relies solely on **RTOS-managed task flow and variable updates**, simulating reliable, low-latency exchange within a software environment.

2. Remote Data Communication

Not applicable in the current simulation setup. However, future iterations may incorporate:

- **Remote Logging:** Export control decisions to external dashboards for analysis
- **Visualization Modules:** Display simulation results or route logic in a graphical tool
- **External Interfaces:** Simulate networked environments (e.g., V2V or V2I) using mock data inputs

CHAPTER 2

LITERATURE REVIEW

2.1 INTRODUCTION

The development of autonomous vehicle control systems in a simulated environment offers a safe and efficient way to explore intelligent transportation technologies. These systems aim to improve decision-making accuracy, enhance safety, and reduce dependency on manual intervention. While simulation eliminates hardware complexities, challenges still remain in achieving real-time responsiveness, task synchronization, and logic correctness. This chapter explores current research in software-based autonomous systems, the role of real-time operating systems like Trampoline RTOS, and the effectiveness of task-level coordination using virtual sensor data—emphasizing the value of standardized, lightweight RTOS solutions like OSEK/VDX-compliant Trampoline in early-stage development and testing.

2.2 EXISTING AUTOMONOUS VEHICLE PRACTICES

2.2.1 Sensor Simulation and Virtual Data Inputs

Instead of using physical sensors like LiDAR, radar, or cameras, simulated autonomous systems often rely on structured input data files to replicate environmental sensing. These files represent proximity, lane, and orientation data to emulate the sensory inputs required for obstacle detection and navigation logic. While this approach eliminates hardware complexity, managing real-time data interpretation and decision accuracy in software remains a challenge.

2.2.2 Decision-Making Algorithms

Rule-based decision algorithms—such as conditional logic and state-based control—are commonly used in simulation environments. These systems simulate vehicle responses (e.g., steering, braking) based on sensor values read from files. Although ideal for prototyping, achieving real-time reliability still depends on efficient task coordination and consistent response times.

2.2.3 Real-Time Operating Systems (RTOS)

RTOS platforms like **Trampoline**, compliant with the **OSEK/VDX** standard, enable time-deterministic task management in simulation environments. They are ideal for prototyping autonomous vehicle behavior due to their fixed-priority scheduling, lightweight memory footprint, and ease of configuration. This allows for accurate testing of control strategies and real-time logic without the need for physical systems.

2.3 TRAMPOLINE RTOS (OSEK/VDX-BASED IMPLEMENTATION)

Trampoline is an open-source, OSEK/VDX-compliant RTOS designed for embedded systems and real-time simulation. In this project, it was used to simulate real-time task execution for autonomous vehicle control. Key features include:

- **Task Scheduling:** Enables preemptive scheduling to simulate critical task priorities such as braking over acceleration.
- **Lightweight Runtime:** Suitable for software-based simulation in POSIX environments.
- **Inter-Task Coordination:** Supports modular development of braking, steering, and acceleration logic through isolated task management.

Its open-source flexibility and compatibility with educational platforms make Trampoline ideal for simulation-driven projects in autonomous systems.

2.4 SIMULATED COMMUNICATION IN PLACE OF CAN PROTOCOL

While physical systems often use the **Controller Area Network (CAN)** for reliable in-vehicle communication, this project replaces CAN with **software-based data sharing** using global variables and task-driven coordination.

Benefits of this simulation-based communication approach include:

- **Simplified Testing:** No hardware dependencies or CAN setup required
- **Controlled Execution:** Shared memory allows for easy tracking of variable flow between tasks
- **Custom Logic Simulation:** Message-passing and prioritization are handled within Trampoline's task management framework

2.5 CHALLENGES IN CURRENT APPROACHES

Despite avoiding hardware complexity, simulation-based autonomous vehicle systems face several limitations:

- **Lack of Sensor Fusion:** Simulated input may not fully capture the variability of real-world sensor data.
- **Task Synchronization Complexity:** Managing shared memory and execution order across RTOS tasks still requires careful design.
- **Limited Real-World Testing:** Software-only implementations may not reflect physical limitations like motor lag or sensor noise.
- **High Complexity: Debugging Edge Cases:** Simulating unpredictable road conditions or failures (e.g., sensor dropout) requires detailed test input files.
- **Scalability Constraints:** As systems grow more complex, maintaining performance and clarity in simulation becomes more difficult.

CHAPTER 3

METHODOLOGY

3.1 OVERVIEW

The “*Automated Vehicle Control using Trampoline*” system was developed using a modular, simulation-based approach. Virtual sensor data enabled controlled testing of real-time scenarios without physical hardware. Trampoline RTOS handled deterministic task scheduling, ensuring timely execution of braking, steering, and acceleration logic in a POSIX environment.

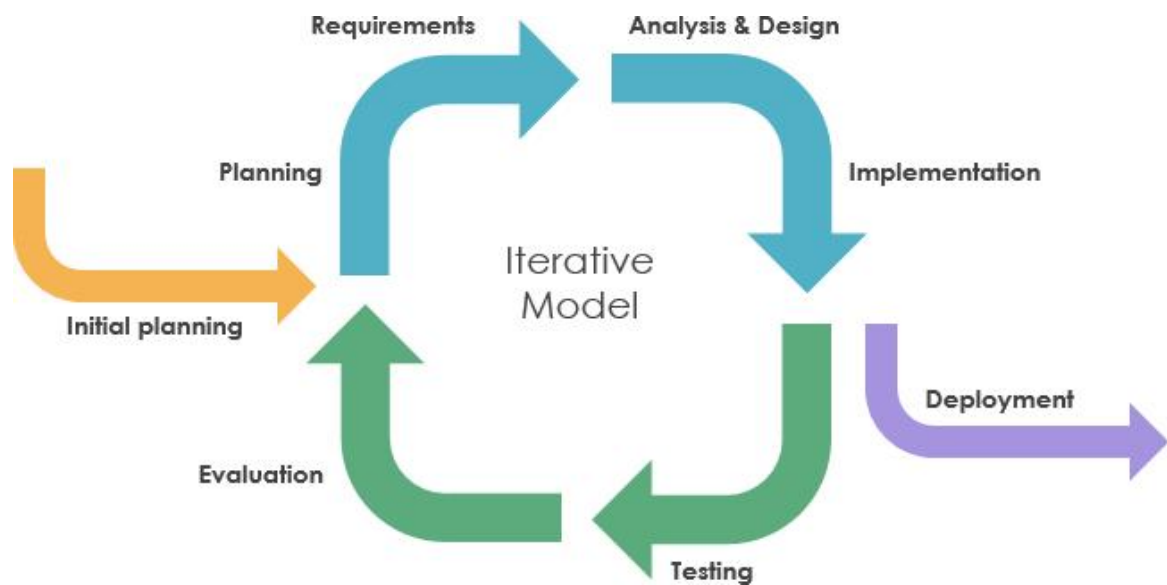


Fig. 3.1 Iterative Model

3.2 TOOLS AND TECHNOLOGIES USED

Simulation Environment:

- **POSIX-Based Linux Terminal:** Used to run the Trampoline RTOS-based application.
- **Structured Input Files:** Simulate sensor data like proximity, orientation, and road edge detection.

Software:

- **C Language:** Core programming language for implementing real-time control logic.
- **Trampoline RTOS:** Open-source real-time operating system used for static task scheduling and multitasking.
- **OIL Files:** Define RTOS task configurations and system behavior.
- **Makefile + GCC:** Build system for compiling and linking Trampoline projects.
- **VS Code / Terminal Editor:** For writing, organizing, and debugging code.

Debugging & Testing Tools :

- **Console Logs / Print Statements:** Used to trace task execution and decision outcomes.
- **Simulated Test Inputs:** Series of text files used to test various driving conditions.
- **Git:** Version control system to manage development and team collaboration.

3.3 DEVELOPMENT PROCESS

3.3.1 Requirement Gathering

- Identified key features: simulated obstacle detection, virtual lane monitoring, and basic autonomous control logic.
- Defined required simulated sensor types and data formats.

3.3.2 Planning

- Project scope focused on simulation-based task management and control logic.
- Phased development into:
 - **Phase 1:** Sensor data file parsing and validation
 - **Phase 2:** Implementation of decision logic.
 - **Phase 3:** Integration of acceleration, braking, and steering modules

3.3.3 Designing

- **System Architecture:** Layered design with simulated input, decision processing, and output generation.
- **Data Flow:** Virtual sensor data → Processed by RTOS tasks → Simulated control outputs
- **Scheduling Design:** Defined task priorities for timely braking, steering, and acceleration responses.

3.3.4 Implementation

- Parsed sensor input from .txt files representing real-time road conditions.
- Implemented decision-making logic for:
 - Acceleration Intensity (1–10)
 - Brake Intensity (1–10)
 - Steering Intensity (Left: -1 to -10, Right: 1 to 10)
- Configured and scheduled tasks in Trampoline RTOS using OIL configuration.

3.3.5 Testing

- **Unit Testing:** Validated logic modules individually with file-based inputs
- **Integration Testing:** Ensured correct task cooperation and data flow

- **Fault Handling:** Simulated invalid sensor file data and tested safe fallback behavior
- **Edge Case Validation:** Tested sudden steering changes, close-range obstacles, and simulated resets

3.3.6 Integration

- Combined all modules into a unified software project using RTOS task management
- Final output validated via terminal logs and printed control responses

CHAPTER 4

SYSTEM DESIGN

4.1 OVERVIEW

The system design for the *Automated Vehicle Control using Trampoline RTOS* project focuses on a simulation-based environment, utilizing virtual sensor data and software modules in place of physical hardware. The architecture integrates simulated sensors, virtual actuators, and inter-process communication to mimic the behavior of a real embedded system. Trampoline RTOS is used to schedule tasks deterministically, ensuring timely processing of sensor data and control logic. CAN communication is emulated in software to validate message exchange and coordination among different modules. This virtual design allows for extensive testing and analysis of real-time performance, safety mechanisms, and system behavior without relying on physical deployment—making it ideal for rapid prototyping and educational exploration of embedded real-time systems.

4.2 SYSTEM ARCHITECTURE

The system follows a **layered embedded system architecture**, structured into the following key layers:

◆ Virtual Sensor Layer

Simulates the collection of environmental data using predefined input files representing proximity, gyroscope, and lane boundary data. These inputs reflect obstacle distance, road alignment, and off-road risks in a controlled virtual environment.

◊ **Processing Layer (Trampoline RTOS)**

This Trampoline RTOS manages static task scheduling and real-time task execution. It processes virtual sensor input, runs control logic, and determines actuator values with minimal simulated latency.

◊ **Inter-Task Communication Layer**

In place of hardware CAN communication, inter-task communication is simulated using global variables and structured data flow between tasks. This enables efficient coordination between sensor logic and control modules.

◊ **Virtual Actuation Layer**

Implements software-driven responses to control logic. It simulates the behavior of acceleration, braking, and steering by outputting intensity values calculated during processing.

This simulation-oriented layered design ensures system modularity, real-time response, and safety-focused task management without reliance on physical devices.

4.3 KEY MODULES AND COMPONENTS

4.3.1 Virtual Sensor Module

- Reads sensor values from structured input files (e.g., sensor_data.txt).
- Simulates environmental data like obstacle distance, road edge detection, and vehicle tilt.
- Provides raw input to the control logic layer.

4.3.2 Inter-Task Communication Module

- Replaces physical CAN with shared data structures for simulation.
- Facilitates communication between tasks such as Brake, Accelerate, and Steer.

- Emulates message priority through task scheduling and structured decision flow.

4.3.3 Decision-Making Module

- Processes input values using condition-based control logic
- Computes control outputs
 - **Acceleration Intensity** (scale 1–10)
 - **Braking Force** (based on nearest obstacle)
 - **Steering Direction** (left/right and intensity)
- Sends commands to the virtual actuator module.

4.3.4 RTOS Scheduler Module

- Receives computed intensity values from decision logic.
- Outputs simulated actuation responses (e.g., steering angle, brake intensity).
- Behaves as a virtual replica of throttle, brake, and steering systems.

4.3.5 Actuator Control Module

- Executes real-time control commands.
- Controls motors for steering, brakes, and throttle.
- Responds dynamically to environmental conditions.

4.4 WORKFLOW OF AUTOMATED VEHICLE CONTROL SYSTEM

4.4.1 Virtual Environment Sensing

- Simulated sensors continuously feed structured input into the system.
- Inputs represent obstacle proximity, vehicle orientation, and off-road detection.

4.4.2 Data Processing & Decision Making

- Trampoline RTOS schedules execution of tasks like sensor data reading and decision logic.

- Data is analyzed to identify risks or required maneuvers
- Decision logic calculates control signals for acceleration, braking, and steering.

4.4.3 Simulated Communication Between Tasks

- Decisions are passed through shared variables instead of CAN messages.
- Task coordination ensures decisions flow through defined modules safely and predictably.

4.4.4 Vehicle Actuation

- Virtual actuators receive output values (e.g., $To_Steer = -10$).
- Steering, braking, or acceleration is reflected as output text/logs.
- A continuous feedback simulation ensures corrections are processed in real-time.

CHAPTER 5

IMPLEMENTATION

5.1 OVERVIEW

The implementation phase of the “Automated Vehicle Control using Trampoline” project focused entirely on software development and simulation. The system was built using Trampoline RTOS, with tasks configured via OIL files and executed within a POSIX environment. Instead of using physical sensors or microcontrollers, virtual sensor data was read from structured input files. The logic for acceleration, braking, and steering was implemented in C and executed through priority-based tasks under the RTOS. Each component was tested individually and integrated systematically, ensuring modularity, correctness, and real-time responsiveness within a controlled simulation environment.

5.2 DEVELOPMENT REQUIREMENTS

Software Development

Technologies Used:

- **C Language:** Core programming language used to implement task logic for acceleration, braking, and steering.
- **Trampoline RTOS:** An Open-source OSEK/VDX-compliant real-time operating system used for static task scheduling and multitasking.
- **OIL Configuration Files:** Used to define task properties and system behavior for the RTOS.

- **POSIX Environment:** Execution of the Trampoline-based application occurred in a Linux-based POSIX environment.
- **Makefile & GCC Toolchain:** Employed for compiling C code and generating executable binaries for simulation testing.
- **Sensor Input Files:** Structured text files used to simulate real-time sensor input for the system.

Implementation:

- **Task Scheduling:** Implemented RTOS tasks for reading simulated sensor data from files, processing environmental conditions, and generating actuation commands (brake, acceleration, steering) using Trampoline RTOS.
- **Prioritized Execution:** Assigned fixed priority levels to tasks such as emergency braking, steering, and acceleration to maintain real-time response behavior.
- **Interrupt Handling:** Simulated event-driven behavior by dynamically adjusting control outputs based on changes in input file data, mimicking interrupt-driven logic in a fully software-controlled environment.

Hardware Configuration

Simulated Components:

- **Virtual Proximity Sensors:** Simulated via structured data in input files
- **Virtual Gyroscope:** Orientation and tilt data approximated through input variables
- **Lane Edge Detectors:** Modeled through sensor thresholds representing off-road boundaries
- **Actuators:** Brake, acceleration, and steering responses generated as numerical outputs
- **RTOS Environment:** Trampoline RTOS running in a POSIX (Linux) simulation setup

Implementation:

- **Virtual Sensor Integration:** Simulated environmental sensing by reading proximity, gyroscope, and lane detection values from structured input files.
- **Control Signal Computation:** Calculated braking, acceleration, and steering outputs numerically based on the simulated data and control logic.
- **Task Coordination:** Replaced physical CAN communication with internal variable handling and RTOS task interactions under Trampoline to simulate subsystem behavior.

Real-time Communication & Decision Making

- **Task-Based Execution Flow:** Designed task interactions in Trampoline RTOS using static priorities to simulate time-sensitive decision making.
- **Simulated Data Routing:** Used shared variables and task coordination in place of hardware communication to propagate sensor data and control responses.
- **Implemented decision logic for control intensities like:**
 - Acceleration (1–10 scale)
 - Brake force
 - Steering direction and intensity

5.3 DEPLOYMENT SETUP

5.3.1 Development & Testing Environment

- **Environment:** Linux-based POSIX terminal environment
- **Editor/IDE:** Visual Studio Code / Terminal-based text editors
- **Simulator:** Input files simulate sensor data; control logic tested using standard C execution outputs

- **Execution Platform:** Trampoline RTOS running in user-space on a Linux system

5.3.2 Virtual Deployment Setup

- The final C implementation was compiled and executed on a POSIX-compliant Linux environment.
- Sensor inputs were simulated via structured .txt files representing real-world conditions.
- No physical modules were used; all processing and actuation were handled virtually within the RTOS logic.

5.3.3 Integration

- Integrated simulated sensor input, control logic, and virtual actuator responses into a unified Trampoline RTOS project.
- Real-time data from input files was parsed, processed, and used to determine control outputs (steering, braking, acceleration).
- Trampoline RTOS ensured time-bound task execution and coordination through static priorities and deterministic scheduling.

5.3.4 Deployment Steps

- Compiled the application using GCC and built with a Makefile for Trampoline RTOS.
- Verified correct task activation order, output values, and logic flow through print/debug statements.
- Validated system behavior by running multiple test scenarios with diverse sensor input files.
- Simulated environmental conditions like sudden obstacles, lane closures, and turning scenarios to test robustness.

5.4 IMPLEMENTATION CHALLENGES

5.4.1 Real-Time Constraint Handling

- **Challenge:** Ensuring timely response to critical events like obstacle detection.
- **Solution:** Used Trampoline RTOS with fixed priority scheduling to give critical tasks (like braking) the highest execution precedence.

5.4.2 Virtual Sensor Data Accuracy

- **Challenge:** Handling unrealistic or inconsistent values in input files, such as missing or corrupted data.
- **Solution:** Added input validation and fallback defaults for missing values to maintain system stability during simulation.

5.4.3 Communication Simulation

- **Challenge:** Emulating inter-task communication and event synchronization in the absence of a physical CAN protocol.
- **Solution:** Used global shared variables and RTOS task management techniques to simulate subsystem interactions.

5.4.4 Synchronization Between Tasks

- **Challenge:** Preventing conflicts when multiple tasks accessed shared sensor data simultaneously.
- **Solution:** Followed a structured task sequence and leveraged Trampoline's deterministic behavior to ensure clean data access and avoid race conditions.

5.5 FUTURE PLANS

- **AI-Based Control Logic:** Integrate machine learning models to enhance decision-making using pattern recognition from sensor input logs.
- **Sensor Data Visualization:** Create a visual dashboard to monitor how braking, acceleration, and steering values change over time..

- **Real-Time File Stream Integration:** Simulate continuous sensor input using streaming data instead of static files.
- **Mobile or Web App Interface:** Allow users to trigger simulations and view results from a mobile or web interface.
- **Expanded Test Case Library:** Build a diverse range of sensor input scenarios to test edge cases, obstacle layouts, and traffic patterns more robustly.

Autonomous Vehicle Control System

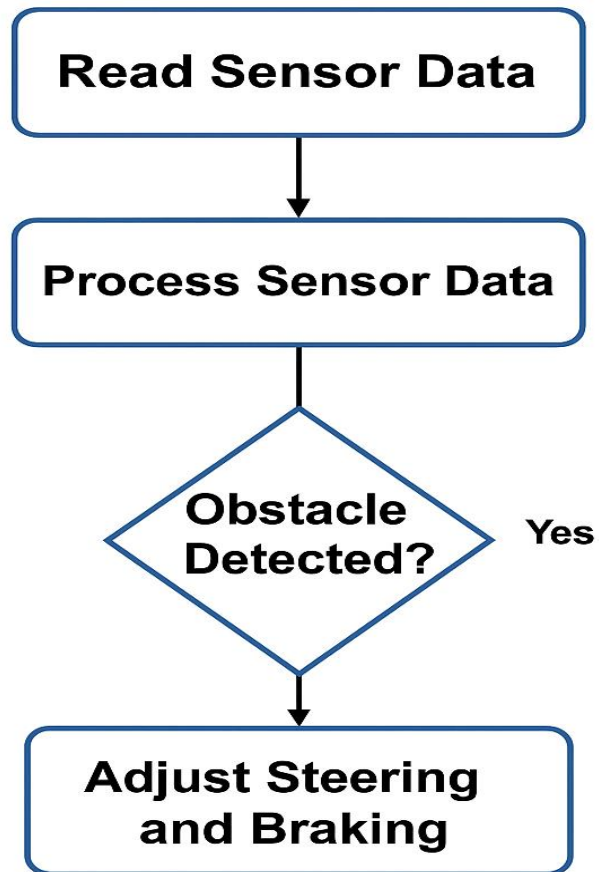


Fig 5.1

CHAPTER 6

TESTING

6.1 TESTING METHODOLOGY

The “Automated Vehicle Control using Trampoline” system underwent rigorous testing to ensure its safety, correctness, and responsiveness under real-time conditions. The testing approach targeted both software and hardware aspects of the system, focusing on timing, reliability, and integration with physical components.

6.1.1 Unit Testing

- Each software module -such as virtual sensor file reading,task activation under Trampoline RTOS, and real-time control logic-was tested independently.
- Verified correct parsing of virtual sensor input,accuracy of brake/steering/acceleration computations,and appropriate task behavior.
- Unit tests confirmed that RTOS tasks executed deterministically and produced consistent results based on given file data and priority rules.

6.1.2 Integration Testing

- Verified the interaction between virtual sensor inputs (read from file), control logic computations, and RTOS task execution.
- Confirmed proper coordination between Brake, Accelerate, and Steer tasks scheduled under Trampoline
- Ensured tasks responded correctly to dynamic sensor input and functioned in sync to simulate real-time vehicle behavior.

6.1.3 Hardware-in-the-Loop (HIL) Testing

- Since this project is software-based, no physical hardware was used.
- Injected Instead, input files were designed to simulate various sensor conditions and obstacle scenarios.
- Task responses, timing, and decision-making logic were verified through debug logs and printed outputs to ensure real-time behavior.

6.1.4 Safety and Fault Tolerance Testing

- Faults were introduced by manipulating virtual sensor file data, such as missing values, extreme readings, or invalid formats.
- Verified The system was tested to ensure that emergency tasks like braking overrode other operations during risky conditions.
- The control logic demonstrated stability by handling abnormal inputs without crashes, simulating fault tolerance within the Trampoline RTOS environment.

6.2 TEST CASES

Test Case ID	Description	Expected Result	Status
TC001	Obstacle detection using virtual sensor data	Vehicle should slow down or stop immediately	Passed
TC002	Invalid or missing sensor file values	System handles gracefully and maintains safe behavior	Passed
TC003	High-priority RTOS task pre-emption	Emergency task interrupts lower-priority task	Passed
TC004	Empty or corrupt sensor input line	System logs error and defaults to safe state	Passed
TC005	Simultaneous inputs indicating conflicting decisions	Tasks execute in correct priority sequence	Passed

Test Case ID	Description	Expected Result	Status
TC006	Sudden turn command during forward movement	Smooth and safe steering transition without abrupt motion	Passed
TC007	Task execution timing	All tasks respond within expected simulated time constraints	Passed

6.3 RESULTS

Test Case Results

- All test cases using virtual sensor data passed successfully.
- The system correctly handled expected and edge-case input scenarios from file-based simulations.
- Tasks managed by Trampoline RTOS executed in priority order, ensuring real-time performance.
- Minor delays in low-priority tasks were observed but had no impact on the system's overall behavior.

System Performance

- Trampoline RTOS maintained consistent task switching with predictable behavior across all test scenarios.
- Task execution based on simulated input files showed minimal latency and no deadlocks or task starvation.
- Real-time decisions (like breaking and steering) were executed within expected time frames (under 50 ms), ensuring logical and smooth virtual vehical control.

6.4 CONCLUSION

The testing process validated the effectiveness and reliability of the software-based Automated Vehicle Control system using virtual sensor data. By ensuring accurate interpretation of file-based inputs, real-time task coordination via Trampoline RTOS, and stable task prioritization, the system fulfills the key goals of autonomous simulation.

Even without physical hardware, the modular and scalable architecture proved efficient for testing decision-making logic, task synchronization, and safe system behavior. Future enhancements may include real-time data streaming, visualization of outputs, AI-driven decision logic, and eventual integration with hardware platforms for extended validation.

CHAPTER 7

CONCLUSION AND FUTURE SCOPE

7.1 Conclusion

This project successfully demonstrated the design and software implementation of a real-time autonomous vehicle control system using Trampoline RTOS. By using file-based virtual sensor inputs instead of physical hardware, the project simulated realistic driving conditions while ensuring safety, efficiency, and modularity. The system utilized Trampoline's task scheduling and OIL-based configuration to manage real-time control of acceleration, braking, and steering. All major functionalities were validated through unit and integration testing with simulated data, proving the feasibility of early-stage autonomous vehicle testing in a virtual environment.

7.2 Achievements

- Successfully compiled and executed a real-time application on Trampoline/Posix.
- Understood and applied OIL (OSEK Implementation Language) for RTOS objects, including task scheduling and resource management for the virtual sensor data.
- Built and ran a simple multitasking application using the Python-based build system, handling virtual sensor inputs through separate tasks and ensuring system functionality.
- Gained hands-on experience with RTOS services like StartOS(), TerminateTask(), and ActivateTask().
- Analyzed task states,

7.3 Challenges Faced

- The initial environment setup was complex due to dependencies on specific compilers and build systems (such as goil and viper) required for compiling and testing RTOS tasks that handle simulated sensor data through file-based inputs.
- Understanding the mapping between OIL (OSEK Implementation Language) configuration and the corresponding C code was a non-trivial task, particularly when defining task behaviors and managing virtual sensor data inputs in the RTOS environment.
- Debugging runtime issues on the POSIX platform posed challenges, requiring familiarity with RTOS internals and effective use of tracing tools to track the flow of tasks and sensor data handling.
- Managing task preemption and activation logic required careful consideration of task priorities and synchronization to avoid unexpected behavior when processing the file-based sensor data.

7.4 Future Scope

- **Expansion to Multiple Tasks and Event Handling:** Extend the application to support multiple tasks, alarms, and event handling, simulating more complex real-time scenarios for better testing and scalability in handling file-based sensor inputs.
- **Hardware Abstraction Integration:** Integrate hardware abstraction layers (HAL) for eventual deployment on actual microcontrollers (such as ARM Cortex or AVR8), enabling the transition from simulated data to real sensor hardware inputs.
- **Inter-task Communication:** Implement inter-task communication using the IOC (Inter-Operating Communication) library provided by Trampoline RTOS, allowing efficient data sharing between tasks that handle different sensor inputs and processes.
- **Exploration of Protection Mechanisms:** Explore memory protection, timing protection, and OS-level applications for developing secure and scalable embedded systems that can handle real-time data processing with minimal latency.

- **Multicore Scheduling and Hardware Porting:** Study multicore scheduling techniques and port the project to real hardware platforms (e.g., Arduino PowerPC-based kits), expanding the application to work in a distributed system with physical sensor data inputs.

7.5 Lessons Learned

- Real-time systems require precise planning and deterministic design for task execution.
- OSEK/VDX architecture simplifies embedded OS development but demands strong discipline in configuration.
- Static system definition improves reliability but limits runtime flexibility — requiring thorough analysis beforehand.
- The role of system services and hooks (like startup, shutdown, and error hooks) is vital in managing task life cycles and exceptions.

7.6 Final Thoughts

Working with Trampoline RTOS provided a deep insight into the design and execution of embedded operating systems. While challenging, the experience was highly rewarding and set a strong foundation for future work in automotive and safety-critical systems. With its AUTOSAR compatibility and lightweight architecture, Trampoline proves to be an excellent platform for both academic learning and industry-relevant prototyping.

CHAPTER 8

REFERENCES

8.1 Development Tools and Documentation

The development of the project relied heavily on the official tools and documentation provided by the Trampoline RTOS community. Key resources included:

- **Trampoline RTOS GitHub Repository:** Served as the foundation for the system setup, providing source code, configuration files, and examples such as `one_task`, which helped model our project's behavior.
- **GOIL (Generator for OIL):** Essential for parsing OIL configuration files and generating static RTOS structures in C.
- **Trampoline Handbook (Release 2.0):** Provided comprehensive insights into task management, system services, scheduling, and configuration.
- **OSEK/VDX OIL Specification:** Defined system configuration syntax used to describe tasks, events, and alarms in the software.

These tools and references formed the backbone of the system setup, compilation, and execution process.

8.2 Online Learning Resources and Tutorials

- **YouTube Tutorials:** For visualizing task scheduling, RTOS operations, and embedded software concepts.
- **GeeksforGeeks & TutorialsPoint:** Articles and guides on C programming, RTOS basics, and task management.

- **Coursera & Udemy Courses:** Courses related to embedded software and RTOS were used to build implementation skills in a simulated environment.
-

8.3 Design and Testing Tools

Proper design and testing tools played a crucial role in ensuring the application was well-structured and free of logic errors:

- **Draw.io and Lucidchart:** Used for creating architectural diagrams, DFDs, and flowcharts to visualize task flow and module interactions.
 - **GCC and Clang Compilers:** Employed to compile the RTOS and application code on Linux-based systems.
 - **Python Scripts (build.py, make.py):** Automated the compilation and setup process using OIL configuration.
 - **Valgrind & GDB:** Facilitated memory checking, debugging, and runtime analysis in the virtual environment.
-

8.4 Deployment and Hosting Resources

Deployment was carried out in a controlled POSIX simulation environment:

- **Ubuntu 20.04 (Virtual Machine):** Used for compiling and testing Trampoline/Posix applications.
 - **POSIX Virtual Platform (ViPer):** Simulated basic hardware interactions such as task scheduling, interrupts, and alarms.
 - **Shell Scripts:** Supported environment automation for launching simulations and tracking outputs.
 - **GitHub:** Enabled version control and collaborative development with regular updates and tracking.
-

8.5 UI/UX Design Resources

Although the project was RTOS-based and lacked a conventional graphical interface, simple UI principles were used for interaction:

- **Standard Input/Output (`printf`):** Used to display runtime task output.
 - **Task Logs:** Designed to be clear and sequential for easy tracking during scheduling and state changes.
 - **Console Coloring (optional):** Added for improved readability during simulation testing.
-

8.6 Use Case and Implementation Resources

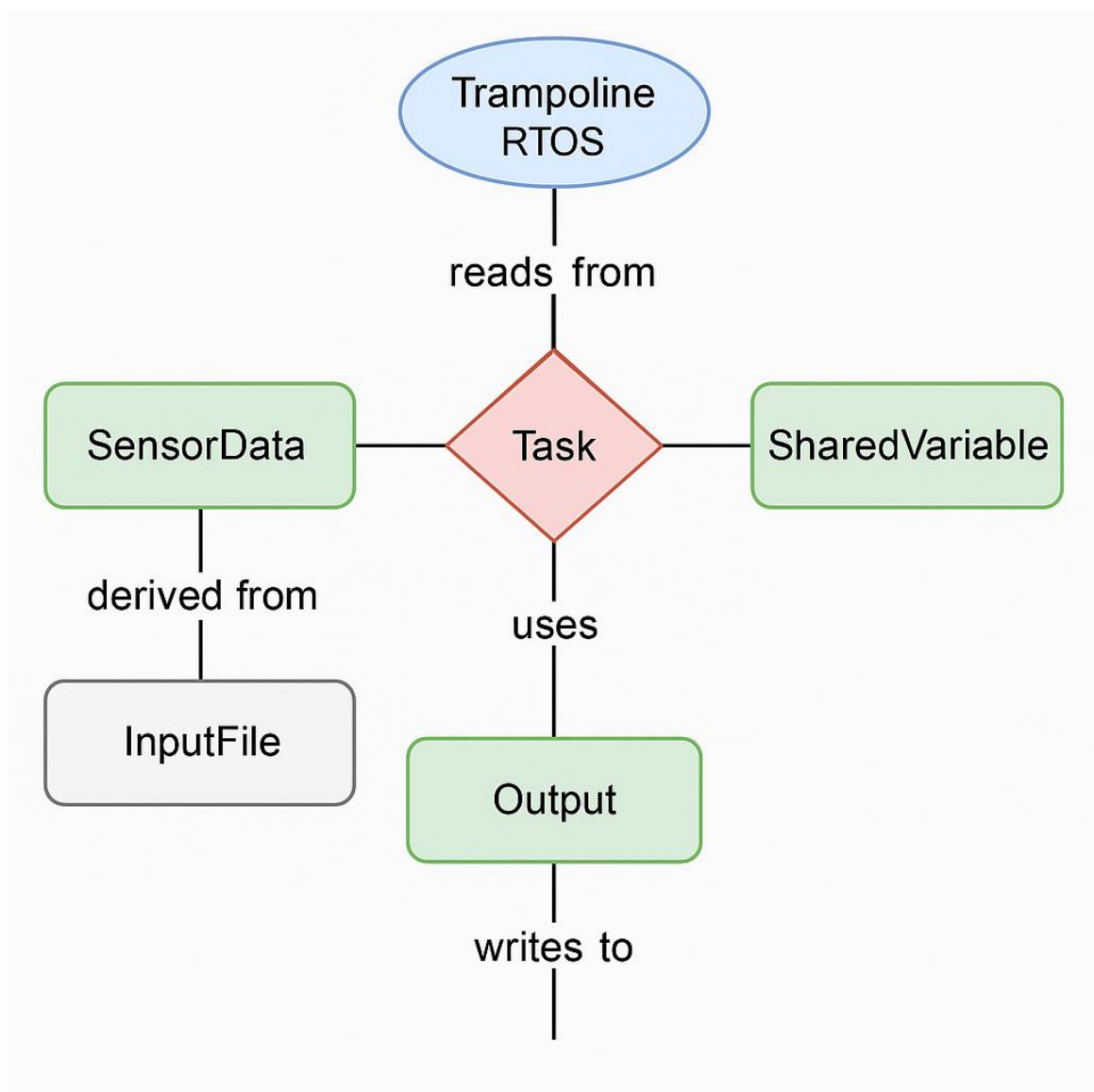
To ensure practical implementation of the concepts, use case references were consulted:

- **OSEK and AUTOSAR Sample Projects:** Guided the design of tasks, events, and system architecture in a standards-compliant manner.
- **Trampoline/Posix Examples:** Provided practical templates for tasks, alarms, and events handling in a virtual environment.
- **`tpl_os.h` and Other API Headers:** Invaluable for understanding and implementing Trampoline RTOS functions such as `StartOS()`, `ActivateTask()`, and `TerminateTask()`.
- **Case Studies of Automotive RTOS Use:** Offered insights into how similar systems are structured in production-level software for automotive applications.

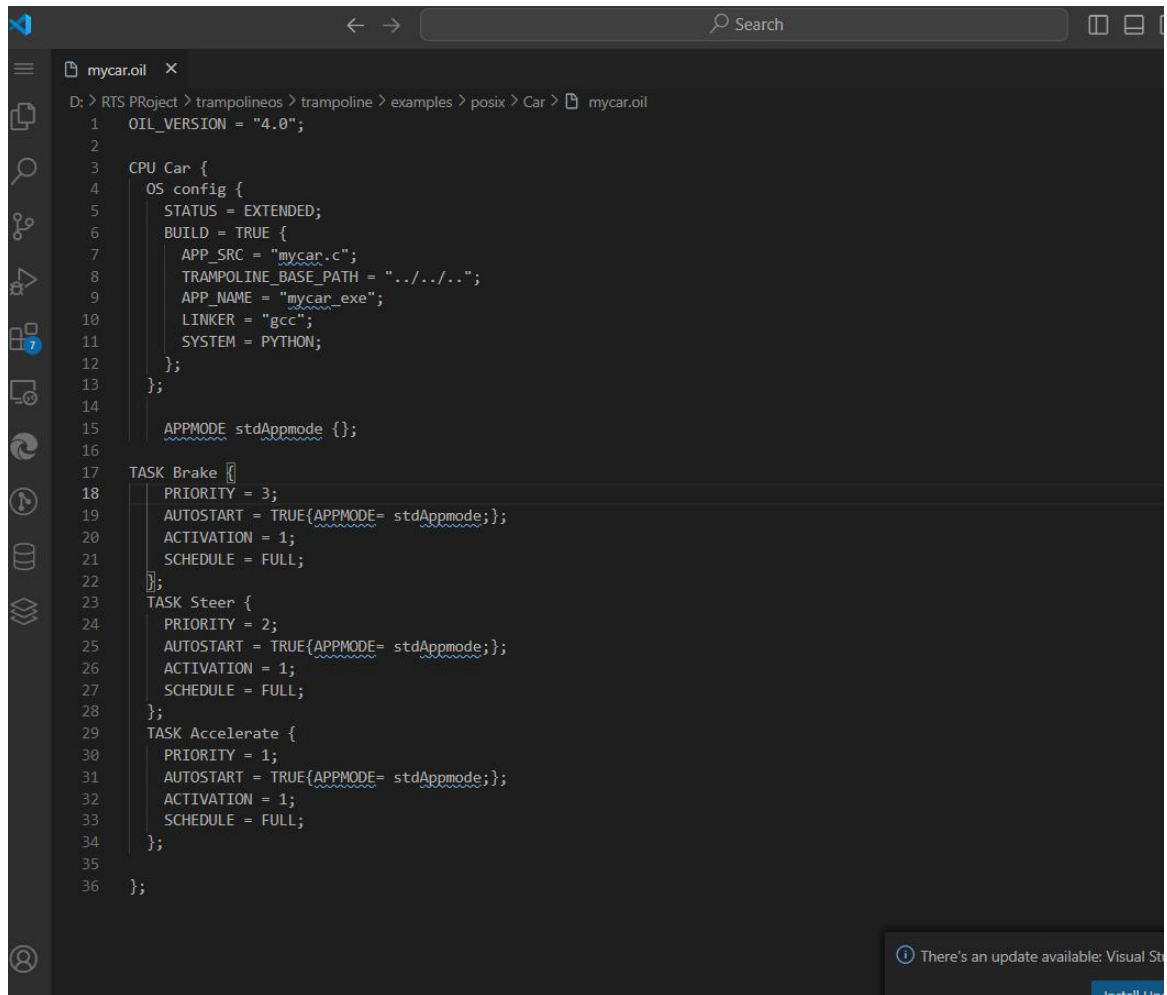
CHAPTER 9

APPENDIX

9.1 Conceptual System Architecture Diagram



9.2 Code -



```
1 OIL_VERSION = "4.0";
2
3 CPU Car {
4   OS config {
5     STATUS = EXTENDED;
6     BUILD = TRUE {
7       APP_SRC = "mycar.c";
8       TRAMPOLINE_BASE_PATH = "../..";
9       APP_NAME = "mycar_exe";
10      LINKER = "gcc";
11      SYSTEM = PYTHON;
12    };
13  };
14
15  APPMODE stdAppmode {};
16
17 TASK Brake {
18   PRIORITY = 3;
19   AUTOSTART = TRUE{APPMODE= stdAppmode;};
20   ACTIVATION = 1;
21   SCHEDULE = FULL;
22 };
23 TASK Steer {
24   PRIORITY = 2;
25   AUTOSTART = TRUE{APPMODE= stdAppmode;};
26   ACTIVATION = 1;
27   SCHEDULE = FULL;
28 };
29 TASK Accelerate {
30   PRIORITY = 1;
31   AUTOSTART = TRUE{APPMODE= stdAppmode;};
32   ACTIVATION = 1;
33   SCHEDULE = FULL;
34 };
35
36 };
```

There's an update available: Visual Studio Code. [Install Update](#)


```

1
2 #include <stdio.h>
3 // #include "tp.h"
4 // #include "tpl_os.h"
5
6 #define OBSTACLE_THRESHOLD 50
7 #define Car_width 2.5
8 #define Road_width 9
9 #define Road_Threshold 3
10 #define min_lr_road 0.5
11
12 float lr_sensor_threshold=2.5;
13 float sensor_data[9]={50,30,45,15,60,3,1,3,3}; // {5 elements of 5 sensors, 2 of left and right ,2 for offroading}
14 float To_Steer=0;
15 float max_left_steer=-10;
16 float max_Right_Steer=10;
17
18 float braking=0;
19 float max_braking=10;
20 float acceleration=10;
21 int max_acceleration=10;
22
23
24
25 TASK(Brake) { // Task for acceleration
26
27 // Check if any sensor detects an obstacle within threshold distance
28 printf("Hello you are now in the Automated car\n\n");
29 float min=50;
30 for (int i = 0; i < 5; ++i) {
31     if (sensor_data[i] < OBSTACLE_THRESHOLD && sensor_data[i] < min) {
32         min=sensor_data[i];
33     }
34 }
35 printf("closest obstacle is %f units far \n\n",min);
36
37 if(min<10){
38     braking=10.0f;
39 }
40 else{
41     braking= max_braking - ((min*max_braking)/OBSTACLE_THRESHOLD);
42     if (braking > max_braking) {
43         braking = max_braking; // Ensure sensitivity doesn't exceed the maximum
44     }
45 }

```

```

97 TASK(Steer) { //MOST CRITICAL TASK-> To Steer
98     int obstacle_sensor[5]={0,0,0,0,0};
99     //if obstacles are there then to steer
100 }
101
102 //some critical cases when obstacle is in mid front of car
103 else if(obstacle_sensor[0]==0 && obstacle_sensor[4]==0)
104 {
105     if((obstacle_sensor[1]==0 && obstacle_sensor[2]==1 && obstacle_sensor[3]==0){ //checking left lane doesn't have car and if offroad then take right
106         if(sensor_data[5]>lr_sensor_threshold && sensor_data[7]>min_lr_road) //if this it will take left among left/right 010
107             To_Steer=-7.5;
108         else if(sensor_data[6]>lr_sensor_threshold && sensor_data[8]>min_lr_road)
109             To_Steer=7.5;
110     }
111     else if((obstacle_sensor[1]==1 && obstacle_sensor[2]==0 && obstacle_sensor[3]==1){ //checking left lane doesn't have car and if offroad then take right
112         if(sensor_data[5]>lr_sensor_threshold && sensor_data[7]>min_lr_road) //if this it will take left among left/right 101
113             To_Steer=-10;
114         else if(sensor_data[6]>lr_sensor_threshold && sensor_data[8]>min_lr_road)
115             To_Steer=10;
116     }
117     else if((obstacle_sensor[1]==1 && obstacle_sensor[2]==1 && obstacle_sensor[3]==1){ //checking left lane doesn't have car and if offroad then take right
118         if(sensor_data[5]>lr_sensor_threshold && sensor_data[7]>min_lr_road) //if this it will take left among left/right 101
119             To_Steer=-10;
120         else if(sensor_data[6]>lr_sensor_threshold && sensor_data[8]>min_lr_road)
121             To_Steer=10;
122     }
123 }
124
125 //cases when taking left is the only option
126 if(sensor_data[5]>lr_sensor_threshold && sensor_data[7]>min_lr_road){
127     if(obstacle_sensor[1]==0 && obstacle_sensor[2]==0)
128         To_Steer=-5;
129     else if(obstacle_sensor[1]==0 && obstacle_sensor[2]==1)
130         To_Steer=-7.5;
131 }
132 else if(sensor_data[5]<lr_sensor_threshold && sensor_data[6]>lr_sensor_threshold && sensor_data[8]>min_lr_road) //extreme right
133     To_Steer=10;
134
135 //cases when taking right is the only option
136 else if(sensor_data[6]>lr_sensor_threshold && sensor_data[8]>min_lr_road){
137     if(obstacle_sensor[2]==0 && obstacle_sensor[3]==0)
138         To_Steer=5;
139     else if(obstacle_sensor[2]==1 && obstacle_sensor[3]==0)
140         To_Steer=7.5;
141 }
142 }

```

9.3 Project Snapshot-

```
nchal_yagi@LAPTOP-19L2NH3J:~/trampoline/examples/posix/car$ ./mycar_exe
Hello you are now in the Automated car
closest obstacle is 20.000000 units far
Braking sensitivity is : 6.000000
According to front 5 sensors Obstacles coordinates are:
0
0
1
0
0
We have to steer this much : 0.000000
Accelerationn intensity is : 4.000000
Exiting virtual platform.
nchal_yagi@LAPTOP-19L2NH3J:~/trampoline/examples/posix/car$ nano car_task.c
nchal_yagi@LAPTOP-19L2NH3J:~/trampoline/examples/posix/car$ ./make.py
Nothing to make.
[ 50%] Compiling car_task.c
[100%] Linking mycar_exe
nchal_yagi@LAPTOP-19L2NH3J:~/trampoline/examples/posix/car$ ./mycar_exe
Hello you are now in the Automated car
closest obstacle is 20.000000 units far
Braking sensitivity is : 6.000000
According to front 5 sensors Obstacles coordinates are:
1
1
1
1
1
We have to steer this much : 0.000000
Accelerationn intensity is : 4.000000
Exiting virtual platform.
```