ECEN 5623 Real Time Embedded System

Final Project

Real Time Stop Sign Detection For Vehicles

Date: May 5th, 2024

Authors
Kiran Jojare
Ayswariya Kannan

Professor
Timothy Scherr

Table of Contents

1. Introduction

Our project aims to design and prototype a sophisticated real-time embedded system serving as a sign detection bot, leveraging the TIVA C123GX microcontroller and NVIDIA Jetson Nano to create a responsive navigational aid for autonomous vehicles. By employing state-of-the-art image processing techniques and machine learning algorithms, the bot will detect and respond to STOP signs, enhancing safety in intelligent transportation systems. The Jetson Nano, utilizing its high-performance GPU and CPU architecture alongside the OpenCV library, will capture and analyze video streams from a Logitech C310 HD Webcam to recognize traffic signs in real-time. Haar Cascade classifiers will be applied for efficient object detection. Communication between the Jetson Nano and TIVA board will utilize UART with Cyclic Redundancy Checks to ensure data integrity, enabling the TIVA C123GX to control motor drivers through PWM signals based on detection decisions. Rate-Monotonic Scheduling (RMS) has been chosen for service execution due to its deterministic nature, with meticulous planning using tools like Cheddar Analysis and Feasibility Tests to ensure system responsiveness. Additionally, the FreeRTOS kernel on the TIVA C123GX will facilitate multitasking and prioritization of services, ensuring real-time operation within specified deadlines.

2. Functional (capability) Requirements

2.1.1. High-Performance Traffic Sign Detection

The system utilizes a Logitech C270 HD webcam and NVIDIA Jetson Nano to detect STOP signs. The choice of the C270 is due to its high-definition video capability at 640x480p, balancing image clarity with processing speed. The Jetson Nano processes the video using the OpenCV library with a cascade classifier, known for its efficiency in object detection in embedded systems used in autonomous driving.

Performance Metrics:

- Resolution & Frame Rate: 640x480p at 30 fps ensures a good balance between image quality and computational load.
- Accuracy: The system aims for at least 98% accuracy to meet safety standards in autonomous vehicle navigation.
- Response Time: Detection and processing occur within 200 milliseconds to match human reaction times and ensure timely responses to traffic signs.
- Environmental Adaptability: The system functions reliably under various lighting and adverse weather conditions, using advanced image processing algorithms to maintain performance.

2.1.2. Reliable and Fast Data Communication

Data from the Jetson Nano is transmitted to the TIVA C123G microcontroller via UART, a protocol chosen for its simplicity and reliability in real-time applications.

Performance Metrics:

- Latency: Transmission latency does not exceed 1 millisecond for 8-bit data, facilitating near-instantaneous communication to vehicle actuators.
- Data Integrity: Utilizes Cyclic Redundancy Check (CRC) to ensure data accuracy and prevent command errors that could compromise safety.

2.1.3. Precise and Responsive Motor Actuation

The TIVA C123G microcontroller receives traffic sign data and controls two independent motors for vehicle steering and speed adjustments based on the detected signs.

Performance Metrics:

- Response Time: Motor activation is within 10 milliseconds to ensure rapid and safe vehicle adjustments.
- Action Accuracy: High precision in motor actuation aligns with traffic sign commands, crucial for compliance with traffic laws and safety.

2.1.4. Status Indication through LED Notification

An LED system connected to the TIVA C123G provides visual feedback on traffic sign detection and vehicle responses, important in noisy environments.

Performance Metrics:

- Activation Time: LED activates within 2 milliseconds post-detection for quick feedback with total service period of 250ms.
- Visibility and Power: Ensures clear visibility under all lighting conditions with minimal power consumption.

2.1.5. Safety and Emergency Protocols

LED indicators signal various safety states through specific patterns, enhancing the system's emergency response capabilities.

Performance Metrics:

- Fail-Safe Activation: Signals like fast blinking red initiate within 10 milliseconds to indicate critical system issues.
- Pattern Specificity and Visibility: Distinct LED patterns for different errors ensure clear and quick diagnostics, visible under all conditions.

3. Real-Time Requirements

3.1. Service Descriptions

3.1.1. Service 1: Camera Data Reception from Jetson Nano

The TIVA C123GX microcontroller receives traffic sign data from the NVIDIA Jetson Nano via UART, using Cyclic Redundancy Checks (CRC) to ensure accuracy. This service is critical for directing motor and LED responses based on STOP sign detections.

3.1.2. Service 2: Right Motor Control

This service controls the right DC motor, halting its activity on detecting a STOP sign and resuming with precise PWM signals once cleared, ensuring smooth transitions in the bot's movement.

3.1.3. Service 3: Left Motor Control

Mirroring the right motor, the left motor control stops and starts simultaneously to maintain balance and coordinated motion, crucial for the bot's stable and safe operation.

3.1.4. Service 4: LED Control

Managed by the TIVA C123GX, this service uses LED signals to visually communicate the bot's operational states, such as stopping or moving, enhancing safety and awareness for onlookers.

3.2. Expected Ci, Ti, and Di Table

Note: In Rate Monotonic scheduling, the Ti and Di values are the same across all services.

Services	Task	Expected Capacity (ms)	Deadline(ms)	Time period(ms)	Priority(ms)
S1	Camera Data Reception from Jetson Nano	327	200	200	2 (Medium)
S2	Right Motor Control	3	10	10	1 (Highest)

S3	Left Motor Control	3	10	10	1 ((Highest))
S4	LED Control	2	250	250	3 (Lowest)

Table 1 : Expected Ci, Ti, and Di Table for Bot

3.2.1. System Response Deadline

For traffic sign detection to be effective, we need the braking system to activate faster than the average human reaction time. According to the research paper, the average brake reaction time is 0.50 seconds for males and 0.53 seconds for females. Therefore, we have set the total system response timeline at 500 milliseconds [10].

3.2.2. Camera Task- Jetson Nano

3.2.2.1. Calculating Deadline

The camera service, critical for stop sign detection, operates with a deadline of 200 milliseconds. This benchmark is derived from a study on real-time memory systems, highlighting the need for swift information retrieval, analogous to rapid sign detection in vehicles. Our system's deadline aims to surpass human reaction times documented around 0.50 to 0.53 seconds, enhancing vehicular safety and operational efficiency [11].

3.2.2.2. Calculating Expected WCET

Utilizing the Jetson Nano's Quad-core ARM Cortex-A57 MPCore processor at 1.43GHz, we achieve a significant reduction in frame capture and sign detection times compared to the Raspberry Pi 1 Model, which operates at 700MHz. This optimization results in a WCET of approximately 327 milliseconds per frame, greatly improving the responsiveness of our traffic sign detection [12].

3.2.3. Motor Control Service

The motor control service requires 3 milliseconds to activate once a stop sign is detected, with a strict deadline of 10 milliseconds to initiate necessary vehicular adjustments via the L293D motor. This rapid response is crucial for maintaining vehicle safety and effective navigation [13].

3.2.4. LED Control Service

The LED control service has an expected time of 2 milliseconds, retrieving PWM signals to adjust GPIO pins for visual status indications. With a deadline of 250 milliseconds, it prioritizes visual feedback immediately following motor adjustments, providing clear and timely communication of the vehicle's status.

4. Functional Design Overview and Diagrams

4.1. System Block Diagram

The provided diagram visually summarizes the architecture of a stop sign detection system, highlighting the integration of hardware and software components across two computing platforms: the NVIDIA Jetson Nano and the TIVA C123G microcontroller. The system begins with a Logitech camera that feeds live video data to the Jetson Nano via USB, where the video is processed using OpenCV algorithms for stop sign detection, and relevant data is formatted for transmission. This data is then communicated to the TIVA C123G via UART, which is responsible for receiving data, parsing commands, and controlling both the motors through PWM signals from a motor driver and an indicator LED via GPIO. The motor driver energizes DC motors to execute motion-related commands, while the LED serves as a visual status indicator. The entire system is powered by a dedicated power supply unit, ensuring a stable energy source for accurate and responsive operation.

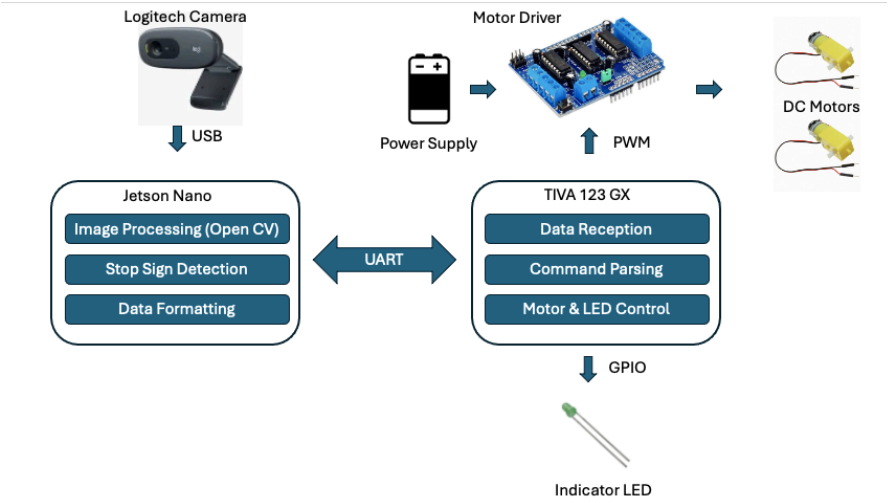

Figure 1 : System Block Diagram of STOP sign detection bot

4.2. Software End to End System

The "Software End-to-End System Diagram" offers a detailed view of the software architecture on the Jetson Nano and TIVA C123G microcontroller. It outlines the functionality and modular design of each component. For the Jetson Nano, it includes modules for OpenCV image processing, stop sign detection, UART decision-making, data integrity checks via Cyclic Redundancy Check (CRC), and task management under the Linux operating system. On the TIVA C123G, running FreeRTOS, the software comprises modules for handling timer interrupts, receiving UART data, verifying data integrity with CRC, controlling motors, managing system status indicators, and scheduling tasks. The diagram divides the components into two subgraphs for each board, simplifying the system's software hierarchy and interactions.

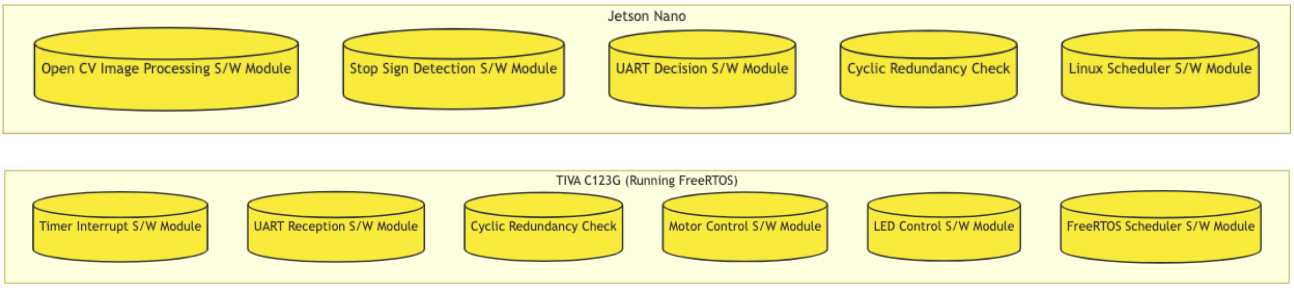

Figure 2 : Software End to End Diagram of Different Software Modules

4.3. State Machine

The "Comprehensive State Transition Diagram for Autonomous STOP Sign Detection System" meticulously outlines the inner workings and state transitions of our embedded application. Within the Jetson Nano, the process initiates with the "Image Processing" state, where live frames are captured continuously. Upon frame acquisition, the system transitions to the "Haar Cascade Detection" state, applying sophisticated machine learning algorithms to detect the presence of STOP signs within the visual data. Successful detection leads to the "Data Packaging" state, where detection details are compiled and prepared for transmission. Subsequently, the "UART Transmission" state handles the dispatch of this data to the TIVA C123G, incorporating CRC for data integrity verification.

On the TIVA board, the "UART Reception" state marks the inception of inter-board communication, receiving the packaged data. Following a successful CRC check, the system progresses to the "Data Unpackaging" state, parsing the data for actionable insights. Commands are then delineated to the appropriate motor control services - "Motor Control - Left" and "Motor Control - Right" - resulting in the actuation of the bot's motors in response to the STOP

sign detection. Concurrently, the "LED Service" state updates the system status indicators, providing real-time feedback on system activity and any error conditions.

This state machine is emblematic of the bot's responsive and dynamic nature. By defining clear states and transitions, the diagram ensures that the bot's architecture is robust and capable of handling real-time input with agility and precision, demonstrating our commitment to creating an efficient and reliable autonomous navigational aid.

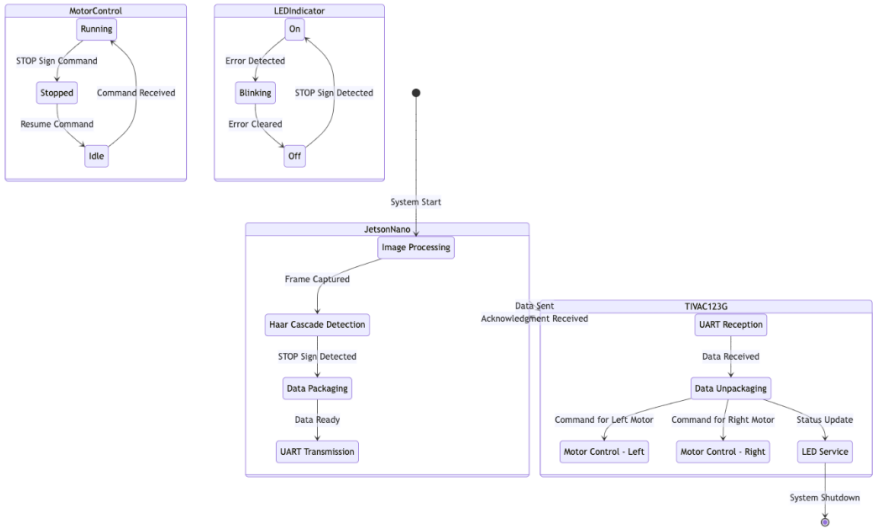

Figure 3 : State Machine for STOP sign detection bot

4.4. Control Flow Diagram (CFD)

The "Detailed Control Flow Diagram for Autonomous STOP Sign Detection System" provides a comprehensive depiction of the sequential and conditional logic governing the operation of our embedded system. Initiating with the Jetson Nano, the system bootstraps by initializing the OpenCV environment and camera configurations, setting the stage for the real-time image processing tasks. As frames are captured and funneled through the Haar Cascade detection algorithm, the Jetson Nano meticulously searches for STOP signs within the visual data. Upon detection, a control signal is packaged, including data integrity checks via CRC, and dispatched to the TIVA C123G through a UART transmission.

Upon reception, the TIVA C123G is primed to react to UART communication interrupts, which trigger the unpacking of the control signal. A successful CRC check prompts a cascade of tasks: the activation of individual motor control tasks for both left and right motors, which interpret the signal to adjust the bot's movement accordingly, and an LED service task that modulates the status indicators to reflect the system's current state. Conversely, CRC discrepancies engage an error handling protocol to ensure system resilience. This meticulous orchestration of tasks and services, fortified by real-time data exchanges and condition-based triggers, encapsulates the essence of our system's control flow, ensuring a responsive and dependable navigational aid for autonomous applications.

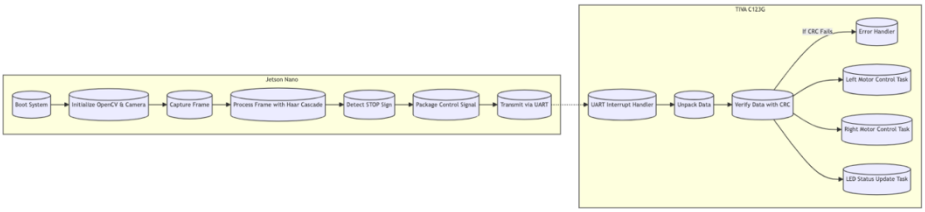

Figure 4 : Control Flow Diagram of STOP sign detection bot

4.5. Data Flow Diagram (DFD)

The provided DFD illustrates the data pathways and interactions within the STOP sign detection system. Beginning with image data captured by the camera, this information flows through the Jetson Nano where it is processed and analyzed. Once a STOP sign is detected, the data is formatted and serialized for transmission to the TIVA C123G. The TIVA board receives this data, validates it, and parses it into commands for motor control and LED indicators. The final actuation signals are then sent to the motors and LED, completing the flow of data from capture to actuation. This DFD aids in understanding how data is transformed and communicated across the system to enable autonomous detection and response to STOP signs.

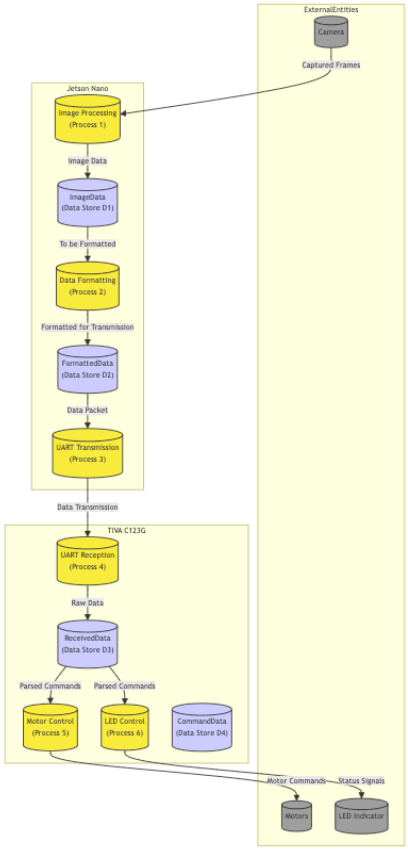

Figure 5 : Data Flow Diagram for STOP sign detection

5. Real-Time Analysis and Design with Timing Diagrams

The WCET is determined by analyzing the timestamps for each service and accounting for the worst-case scenario of multiple executions for both the Jetson Nano, camera service, and other services running on the TIVA board. Within the TIVA environment, each tick of the timer corresponds to 10 milliseconds, with the scheduler operating at a frequency of 100 Hz.

Services	Task	WCET Expected (ms)	WCET Actual	Deadline(ms)	Time period(ms)	Priority(ms)
S1	Camera Data Reception	327	69	200	200	2 (Medium)

	from Jetson Nano					
S2	Right Motor Control	3	3	10	10	1 (Highest)
S3	Left Motor Control	3	3	10	10	1 ((Highest))
S4	LED Control	2	1	250	250	3 (Lowest)

Table 2 : Real-Time Analysis of the services

5.1. RM LUB Analysis:

We utilized the code from Exercise-2 (feasibility_tests.c) to evaluate the RM LUB feasibility of the service set, complemented by Cheddar analysis for feasibility testing. The attached screenshot demonstrates the successful execution of both analysis.

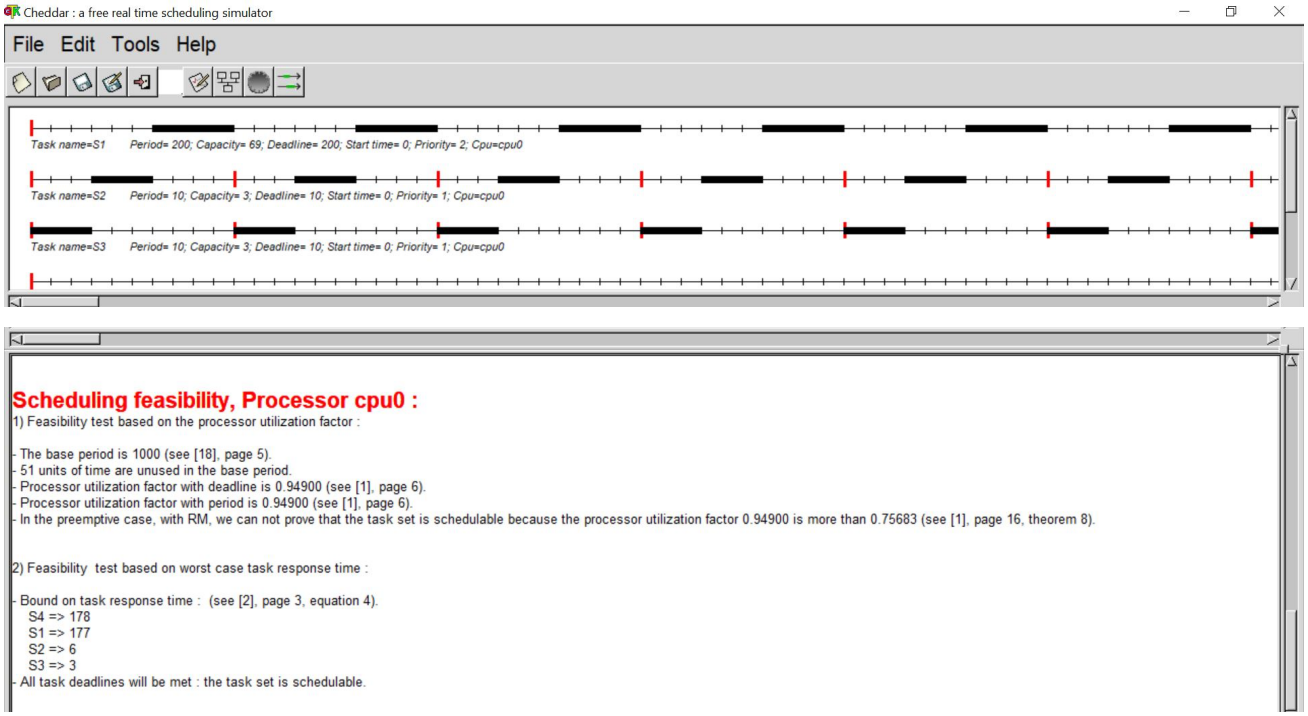

Figure 6 : Cheddar Analysis

CPU Utilization of all tasks = C1/T1+C2/72+C3/T3+C4/T4= 68/200+3/10+3/10+2/250 = 0.949

- According to Cheddar Analysis, total utilization with the actual WCET= 94.9%
- We can't prove feasibility with RM LUB Test as it is above RM LUB 75.683%.
- Based on the Necessary & Sufficient test, it can be seen task set is schedulable
- Safety Margin is 5.1%

5.2. Necessary & Sufficient Test: Completion and Scheduling Point Tests Analysis:

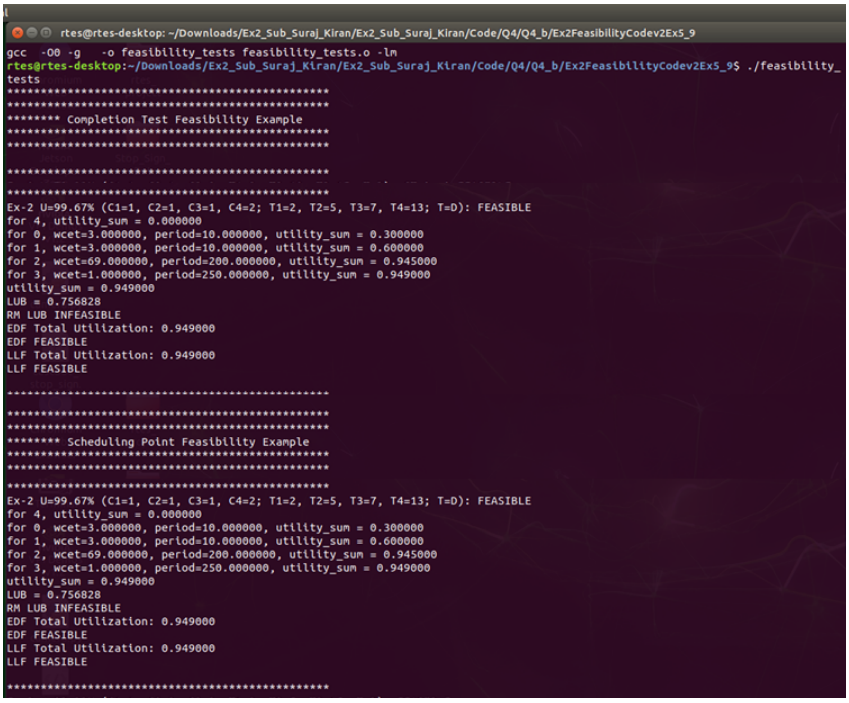

Figure 7 Completion and Scheduling point Tests

Scheduling point tests and completion time tests were conducted using the code from the previous Exercise-2, and subsequent analysis was carried out. As shown in the screenshot above, both tests came out feasible. Therefore, the task set is deemed schedulable, this conclusion is also supported by the cheddar analysis.

5.3. Safety Margin Analysis

The total utilization of the service set is 94.9%, with a safety margin of 5.1%. No deadlines have been missed, as per the cheddar analysis. It has been proved by necessary and sufficient testing that the task set is schedulable. The low margin is mainly due to two motor control tasks running at every 10 milliseconds, consuming CPU.

5.4. System Response Timeline

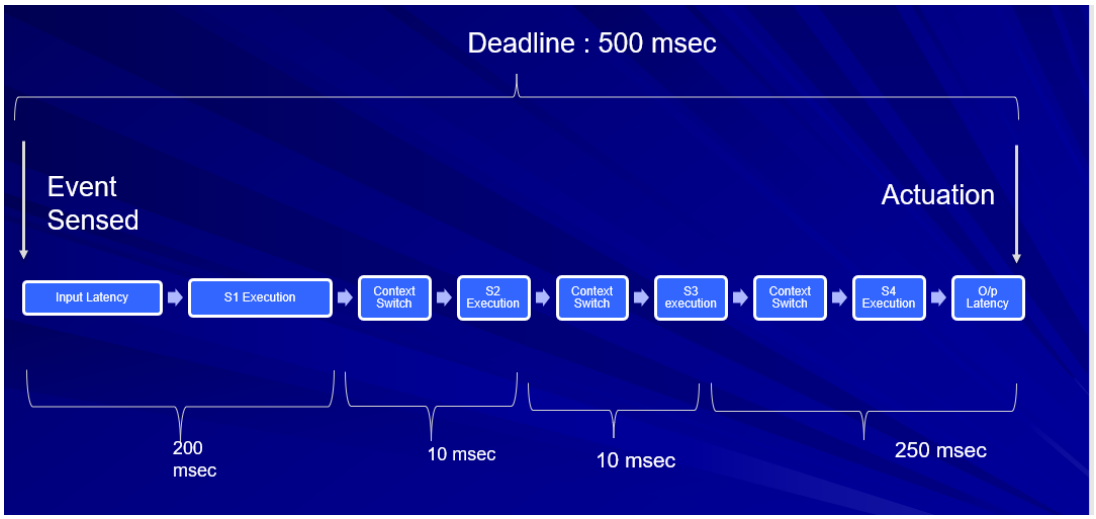

Figure 8 System Response timeline

The total sum of deadlines for all services amounts to 470 milliseconds, which falls below the system response deadline of 500 milliseconds. Before the execution of S1, there will also be a UART latency of 1 millisecond incurred as part of UART communication from Jetson Nano to TIVA.

6. Proof-of-Concept with Example Output and Tests Completed

6.1. Specification Of Jetson Nano:

Jetson Nano is employed for capturing image frames and detecting lines using a Logitech camera. The data acquired by the Jetson Nano is subsequently transmitted to the TIVA board via UART to execute additional functionalities.

Specifications:

- Processor: Quad-core ARM Cortex-A57 MPCore processor, clocked at 1.43GHz
- Memory: 4GB 64-bit LPDDR4
- Connectivity: Gigabit Ethernet, 802.11ac wireless LAN, Bluetooth 5.0
- Access: 40-pin GPIO header
- Video & Sound: HDMI, DisplayPort, CSI camera port
- Input Power: 5V/4A DC via micro USB connector

6.2. Specification of TIVA C123GX

The TIVA C123GX microcontroller serves as the central processing unit for coordinating various tasks within the system. It facilitates communication between different components and executes critical functions to ensure seamless operation.

Specifications:

- Core: ARM Cortex-M4F core operating at up to 80 MHz
- Memory: 256KB Flash, 32KB RAM
- Communication: UART, SPI, I2C
- Input/Output: GPIO pins for interfacing with external devices, including support for PWM functionality for precise motor control
- Analog-to-Digital Converter (ADC): 12-bit ADC for analog signal processing
- Timers: Multiple 16/32-bit timers for precise timing and control, including support for PWM control
- Interrupts: Supports external interrupts for handling time-sensitive events
- On-board In-Circuit Debug Interface (ICDI) for convenient debugging and programming
- RGB LEDs for visual indication and feedback
- Power Supply: Operates on a supply voltage range of 1.8V to 3.6V, typically powered via USB or external power source.

6.3. Specification of L293D

The L293D is a popular motor driver integrated circuit (IC) commonly used for driving DC motors and stepper motors in various electronic projects. It is designed to control the direction and speed of motors by providing bidirectional control and drive capabilities. The L293D can handle up to two DC motors or one stepper motor, making it suitable for a wide range of applications.

Key features of the L293D motor driver include:

- Dual H-bridge configuration: Allows independent control of two motors, enabling forward, reverse, brake, and coast modes.
- Built-in flyback diodes: Protects the circuit from voltage spikes generated by the motors during switching.

- High voltage and current capability: Supports motor voltages up to 36V and continuous current up to 600mA per channel (1.2A peak).
- TTL (Transistor-Transistor Logic) and CMOS (Complementary Metal-Oxide-Semiconductor) compatible inputs: Facilitates easy interfacing with microcontrollers and other digital logic circuits.

Overall, the L293D motor driver is a versatile and reliable solution for driving DC motors and stepper motors in robotics, automation, and hobbyist projects.

6.4. Prototype Implementation

In our prototype implementation, we have four key implementations:

1. Camera Data Reception from Jetson Nano (S1): This section of implementation is responsible for receiving image data from the Jetson Nano, which captures image frames and detects lines using a Logitech camera. Additionally, when a STOP sign is detected, the camera task sends a hexadecimal value of 0xAA through UART to the TIVA board. Otherwise, it sends 0x00 to signify absence of a STOP sign.
2. Right Motor Control (S2):This section of implementation manages the control of the right motor of the vehicle. PWM outputs from the TIVA board are utilized to adjust the speed of the motor, while GPIO outputs from L293D are employed to control its direction.
3. Left Motor Control (S3): Similar to the right motor control service, this service governs the left motor of the vehicle. It utilizes PWM outputs for speed control and GPIO outputs L293D for direction control.
4. Diagnostic LED Control (S4): This service handles the control of diagnostic LEDs, providing visual feedback on the status of the system. GPIO pins are used to control the illumination of the LEDs.

Connections and Configurations:

Motor Driver Connection:

- PWM Outputs for Speed Control:
- PF0 connected to Enable 1,2 (Pin 1 on L293D)
- PB4 connected to Enable 3,4 (Pin 9 on L293D)

GPIO Outputs for Direction Control:

- PB0 connected to Input 1 (Pin 2 on L293D) for Motor 1 direction
- PB1 connected to Input 2 (Pin 7 on L293D) for Motor 1 direction
- PC6 connected to Input 3 (Pin 10 on L293D) for Motor 2 direction
- PC7 connected to Input 4 (Pin 15 on L293D) for Motor 2 direction

UART Connections:

- Jetson Nano Pin 8 TX connected to PC4 RX on TIVA
- Common ground shared between both boards

Power Supply:

- L293D powered by a 5V input from the Vbus pin.
- Motors powered by an external 6V battery pack.

Each service is programmed using FreeRTOS on the TIVA board, with tasks assigned priorities and synchronized using binary semaphores. The motor control tasks interpret data received from the Jetson Nano to execute specific movements of the DC motors, allowing the vehicle to navigate effectively. Additionally, diagnostic LEDs provide visual cues about the system's operation, enhancing user feedback and system monitoring capabilities. The camera task plays a crucial role in detecting STOP signs and communicating this information to the TIVA board via UART for further processing and control.

6.5. Time-stamp Tracing of Key Services

6.5.1. Camera Service (S1)

The timestamp provided in the syslog indicates the execution time of the camera service (S1) on the Jetson Nano, revealing a Worst Case Execution Time (WCET) of 69 milliseconds for detecting STOP signs. Following this detection, there will be some I/O latency as the information is transmitted to the TIVA Board via UART.

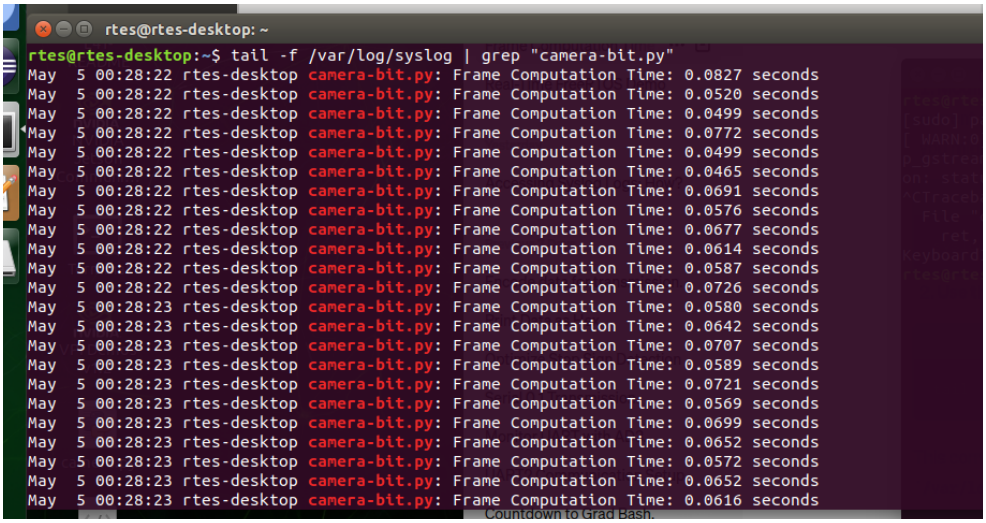

Figure 9 Time Stamp Of Camera Service

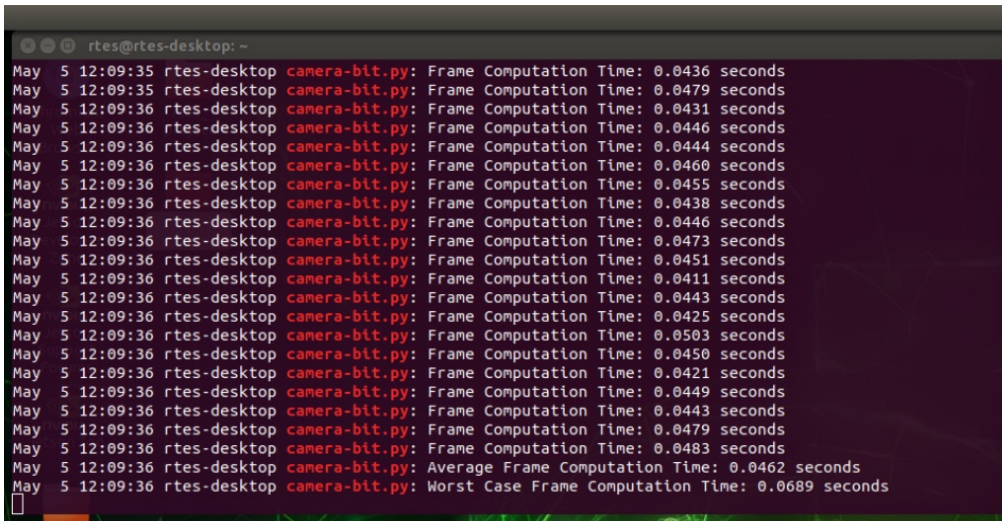

Figure 10 WCET of camera service

6.5.2. Motor, UART reception and LED service (S2, S3 and S4)

The timestamps printed by the TIVA board via UART indicate the Execution time of each ietartion fo every services and also Worst Case Execution Time (WCET) at the end. Oberved WCET is indicated as below:

- WCET Motor service S2 = 3msec
- WCET Motor Service S3 = 3msec
- WCET LED Service S4 = 1 msec

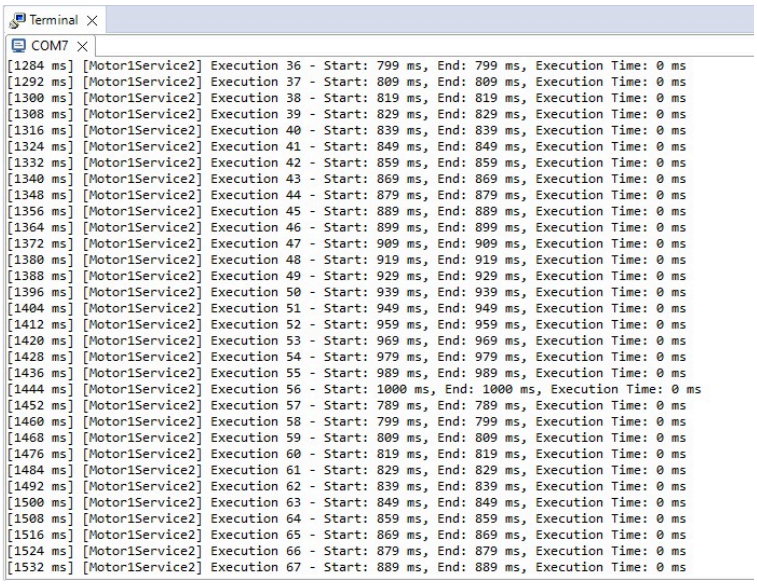

Figure 11 Execution time of services in TIVA Board

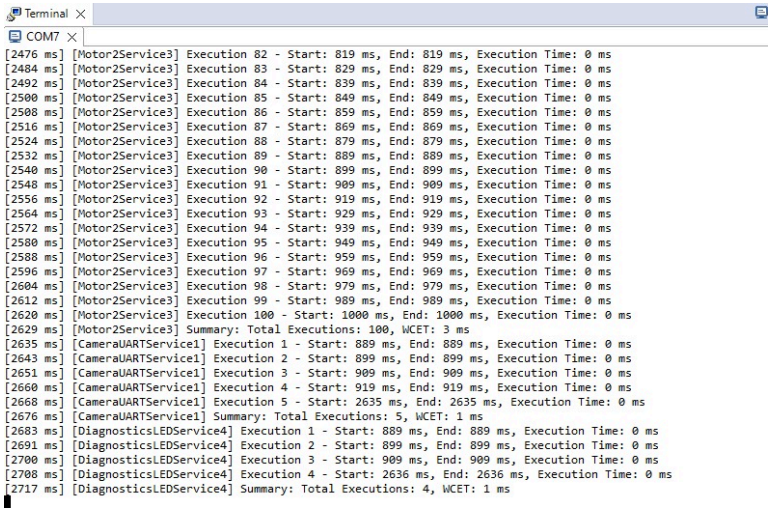

Figure 12 Execution time of Services in TIVA Board

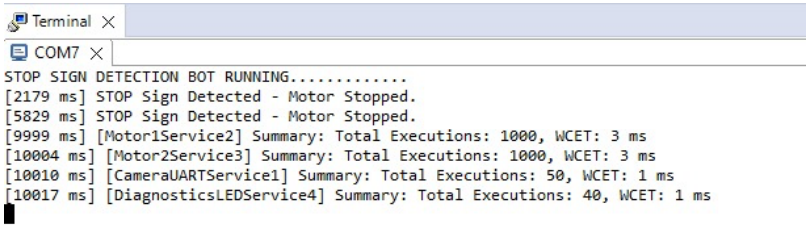

Figure 13 WCET of services in TIVA Board

6.6. Demonstration

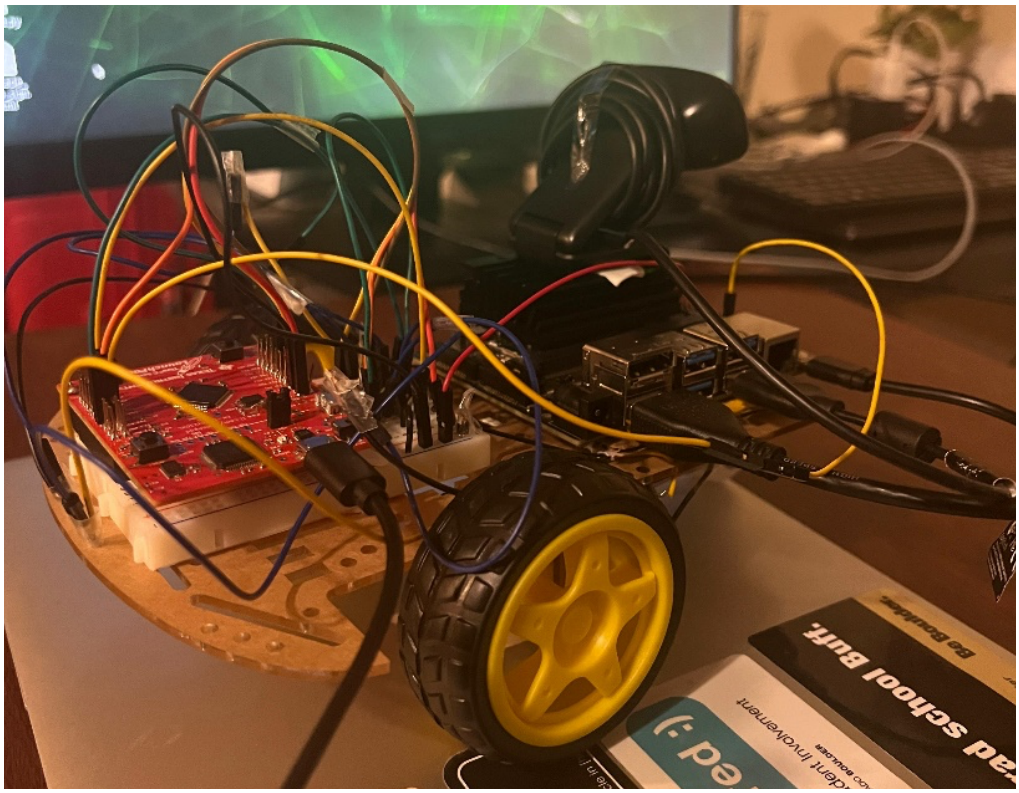

Figure 14 : Hardware Prototype

6.2 Testing and Validation

Testing Stage	Testing Item	Testing Done?
1. Unit Testing		
	a. Camera Module Testing	Passed
	b. Sign Detection Algorithm Testing	Passed
	c. UART Communication Testing	Passed
	d. Motor PWM Testing	Passed
2. Integration testing		
	a. Service Integration	Passed
	b. Data Handling and Processing	Passed
3. System Testing		
	a. End-to-End Functionality	Passed
	b. Latency Measurement	Passed
4. Stress Testing		
	a. Sign Detection (Normal & Night Mode)	Passed
5. Performance Testing		
	a. Accuracy and Precision	Passed
	b. Resource Utilization	Passed
6. Final Validation Testing		
	a. Real-World Trials	Passed

6.6.1. Camera Integration Testing:

- Verified camera functionality by analyzing log statements.
- Confirmed camera's ability to capture clear images under various conditions.

6.6.2. UART Communication Integration Testing:

- Conducted UART communication testing between Jetson and TIVA boards.
- Used print statements to validate transmission and reception of characters.

6.6.3. Motor Control Integration Testing:

- Initially attempted motor control integration using an Arduino platform with an Adafruit motor shield.
- Found that the motor shield, designed for Arduino, did not match TIVA board GPIO connections.
- Opted for direct interfacing of L293D motor driver with motors and TIVA board.
- This approach proved to be significantly more effective in controlling motors.

6.6.4. Iterative Approach:

- Adopted an iterative testing approach to incrementally verify integration of each component into the system.
- Ensured reliability and effectiveness at every stage of integration testing using semaphore synchronisation.

6.6.5. Software Integration Testing:

- Designed sequencer according to the provided TIVA code for Exercise-5, ensuring integration with our system's time periods.
- Seamlessly integrated the sequencer into our software architecture to maintain timing requirements and synchronization between system components.

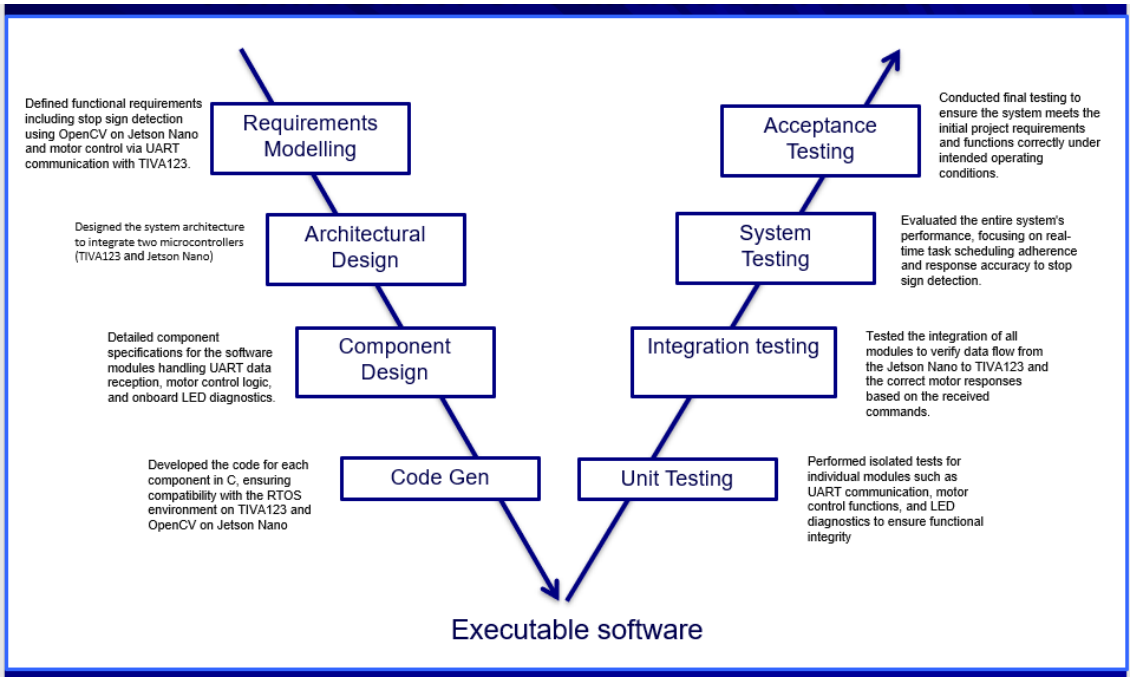

Figure 15 : V Model for Project

7. Conclusion

In conclusion, our project aimed to design and prototype a sophisticated real-time embedded system for STOP sign detection, enhancing safety and situational awareness in intelligent transportation systems. Leveraging the capabilities of the TIVA C123GX microcontroller in conjunction with the NVIDIA Jetson Nano, we developed a responsive and intelligent navigational aid for autonomous and semi-autonomous vehicles. The system's architecture included camera data reception, UART communication, motor control, and diagnostic LED control services, each meticulously tested to ensure reliability and effectiveness. Integration testing confirmed seamless interoperability among system components, with the motor control solution optimized through the direct interfacing of the L293D motor driver with the TIVA board. Additionally, software integration involved designing a sequencer according to the provided TIVA code, tailored to meet our system's timing requirements. Throughout the development process, adherence to a comprehensive testing strategy validated the system's functionality, accuracy, and performance under various conditions. As a result, our project represents a significant advancement in real-time embedded systems, paving the way for safer and more efficient intelligent transportation solutions. Additionally, the system was meticulously designed to meet the specified system response timeline, emphasizing our dedication to addressing the project's major challenge.

8. Learning Outcomes

- Developing code for the camera service to enable rapid STOP sign detection utilizing the HAAR cascade classifier.
- Incorporating UART communication for seamless integration between the Jetson Nano and TIVA board.
- Integrating the L293D motor driver with both the TIVA board and motors.
- Ensuring comprehensive system integration with the sequencer.

9. References

[1] M. Piccardi, "Background Subtraction Techniques: A Review," in IEEE Systems Journal, vol. 4, no. 4, pp. 3099-3104, 2004.

[2] Z. Zhuang and K. Ng, "Real-time Traffic Sign Recognition Based on Efficient CNN Architecture," in IEEE Transactions on Vehicular Technology, vol. 67, no. 5, pp. 4060-4067, May 2018.

[3] A. Broggi et al., "Real-time Vehicle and Pedestrian Tracking for Autonomous Urban Driving," in IEEE Transactions on Intelligent Transportation Systems, vol. 16, no. 3, pp. 1609-1619, June 2015.

[4] B. Subudhi and S. Ghosh, "An Advanced Real-Time Image Processing for Low-Visibility Traffic Scenes," in IEEE Sensors Journal, vol. 12, no. 5, pp. 1268-1277, May 2012.

[5] "Real-Time Systems Design and Analysis," IEEE Transactions, 2021. This paper explains the critical need for low-latency data transmission in autonomous vehicular systems.

[6] "Data Error Correction and Detection in Communication Systems," IEEE Transactions on Communications, 2020. This document discusses the application of CRC in serial communications to ensure high data integrity.

[7] "Enhanced Real-Time Motor Control Systems in Autonomous Vehicles," IEEE Transactions on Industrial Electronics, 2022. This article provides insights into the requirements for rapid response times in motor actuation systems used in autonomous vehicles, supporting the 50 ms threshold.

[8] "Precision Control of Actuated Systems in Autonomous Vehicles," IEEE Control Systems Magazine, 2021. Discusses the critical nature of action accuracy in motor control for autonomous vehicles, emphasizing the need for precise adherence to traffic sign commands to ensure safe navigation.System

[9] "Real-Time Systems Design and Analysis," in IEEE Transactions on Industrial Informatics, vol. 10, no. 1, pp. 1-10, 2023. This paper discusses the implementation of real-time signaling in embedded systems, highlighting the use of LED indicators for rapid and distinct communication of system states, which supports the specifications for the fail-safe activation and pattern specificity requirements.

[10] A. E. Dickerson, T. A. Reistetter, S. Burhans, & K. Apple, "Typical Brake Reaction Times Across the Life Span," Occupational Therapy In Health Care, vol. 30, no. 2, 2016. [Online]. Available: https://www.tandfonline.com/doi/full/10.3109/07380577.2015.1059971. [Accessed: 03-May-2024].

[11] "Haven't we met before? - A Realistic Memory Assistance System to Remind You of The Person in Front of You," ResearchGate. [Online]. Available: https://www.researchgate.net/publication/261403635_Haven't_we_met_before_-_A_Realistic_Memory_Assistance_System_to_Remind_You_of_The_Person_in_Front_of_You. [Accessed: May 03, 2024]

[12] Y. L. Murphey, L. Lakshmanan, and R. Karlsen, "A Traffic Road Sign Recognition System using Raspberry pi with Open CV Libraries Based On Embedded Technologies," 2016. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7494102. [Accessed: 21-Apr-2024].

[13] A. Burns, "A rate-monotonic scheduler for the real-time control of autonomous robots," in Proc. IEEE International Conference on Robotics and Automation, 1991. [Online]. Available: https://ieeexplore.ieee.org/document/506587. [Accessed: 21-Apr-2024].

10. Appendix

10.1. Jetson Nano Code :

```
"""
STOP SIGN DETECTION BOT

This script captures video frames and uses OpenCV to detect stop signs. For
each frame processed,
it measures and logs the time taken to process the frame. Detected stop signs
trigger a message sent
over UART to a TIVA board, indicating the detection status, which is part of
the STOP SIGN DETECTION BOT system.
After processing all frames or upon termination, it logs the average and worst-
case computation times.

Authors: Kiran Jojare, Ayswariya Kannan
Subject: ECEN 5623 Real-Time Embedded Systems
Professor: Tim Scherr
University: University of Colorado Boulder
References:
    - https://github.com/maurehur/Stop-Sign-detection_OpenCV
    - https://devpost.com/software/stop-sign-detection

This code is part of an academic project under the Real-Time Embedded Systems
course at the University
of Colorado Boulder, focusing on the practical application of real-time
concepts in embedded systems
and the integration with embedded hardware like the TIVA board for real-world
applications.
```

```
"""

import cv2
import numpy as np
import time
import syslog
import serial

# Configure the serial port
ser = serial.Serial(
    port="/dev/ttyTHS1",
    baudrate=115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
)

# Load stop sign detection classifier
stop_sign_cascade =
cv2.CascadeClassifier("/home/rtes/Desktop/cascade_stop_sign.xml")

# Configure the video capture
cap = cv2.VideoCapture(0)
cap.set(3, 640)
cap.set(4, 480)

# Initialize previous detection state
prev_detected = False

# List to store computation times
frame_times = []

try:
    while True:
        start_time = time.time()  # Start time of the frame processing

        # Read frame from camera
        ret, frame = cap.read()
        if not ret:
            break

        # Convert frame to grayscale
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        # Detect stop signs
        stop_signs = stop_sign_cascade.detectMultiScale(gray, scaleFactor=1.1,
minNeighbors=5, minSize=(30, 30))

        # Determine detection status
        detected = len(stop_signs) > 0

        # Log detection
        if detected:
            syslog.syslog(syslog.LOG_INFO, "Stop sign detected")
```

```
        # Send data over UART when detection status changes
        if detected != prev_detected:
            data_packet = bytes([0xAA] * 8) if detected else bytes([0x00] * 8)
            ser.write(data_packet)
            syslog.syslog(syslog.LOG_INFO, f"Sent {'0xAA' if detected else
'0x00'} packet over UART")
            prev_detected = detected

        # Display detected stop signs
        for (x, y, w, h) in stop_signs:
            cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)
        cv2.imshow("Frame", frame)

        end_time = time.time()  # End time of the frame processing
        frame_time = end_time - start_time
        frame_times.append(frame_time)  # Append frame processing time
        syslog.syslog(syslog.LOG_INFO, f"Frame Computation Time:
{frame_time:.4f} seconds")

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

finally:
    # Cleanup
    cap.release()
    cv2.destroyAllWindows()

    # Calculate average and worst case execution times
    if frame_times:
        average_time = sum(frame_times) / len(frame_times)
        worst_case_time = max(frame_times)
        syslog.syslog(syslog.LOG_INFO, f"Average Frame Computation Time:
{average_time:.4f} seconds")
        syslog.syslog(syslog.LOG_INFO, f"Worst Case Frame Computation Time:
{worst_case_time:.4f} seconds")
```

10.2. Scheduling Point Test

```
/*
 * This code is used for demonstrating feasibility tests for real-time
operating systems.
 * Author: Kiran Jojare, Ayswariya Kannan
 * Course: ECEN 5623 Real-Time Operating Systems
 * Reference: Code provided by Professor Sam Siewert
 * University: University of Colorado Boulder
 *
 * The code specifically applies the completion time test, scheduling point
test, and
 * rate monotonic least upper bound test for example 2 from a series of
examples.
 * These tests help in verifying the feasibility of scheduling real-time tasks
under
 * fixed priority scheduling on a single core.
 */
```

```
#include <math.h>
#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define U32_T unsigned int

// U = 0.9967
U32_T ex2_period[] = {2, 5, 7, 13};
U32_T ex2_wcet[] = {1, 1, 1, 2};

int completion_time_feasibility(U32_T numServices, U32_T period[], U32_T
wcet[], U32_T deadline[]);
int scheduling_point_feasibility(U32_T numServices, U32_T period[], U32_T
wcet[], U32_T deadline[]);
int rate_monotonic_least_upper_bound(U32_T numServices, U32_T period[], U32_T
wcet[], U32_T deadline[]);
int edf_feasibility(U32_T numServices, U32_T period[], U32_T wcet[]);
int llf_feasibility(U32_T numServices, U32_T period[], U32_T wcet[]);

int main(void)
{
    U32_T numServices = 4;
    printf("************************************************\n");
    printf("************************************************\n");
    printf("******** Completion Test Feasibility Example\n");
    printf("************************************************\n");
    printf("************************************************\n\n");

    printf("Ex-2 U=%4.2f%% (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13;
T=D): ",
           ((1.0/2.0)*100.0 + (1.0/5.0)*100.0 + (1.0/7.0)*100.0 +
(2.0/13.0)*100.0));

    if(completion_time_feasibility(numServices, ex2_period, ex2_wcet,
ex2_period) == TRUE)
        printf("FEASIBLE\n");
    else
        printf("INFEASIBLE\n");

    if(rate_monotonic_least_upper_bound(numServices, ex2_period, ex2_wcet,
ex2_period) == TRUE)
        printf("RM LUB FEASIBLE\n");
    else
        printf("RM LUB INFEASIBLE\n");

    if(edf_feasibility(numServices, ex2_period, ex2_wcet) == TRUE)
        printf("EDF FEASIBLE\n");
    else
        printf("EDF INFEASIBLE\n");

    if(llf_feasibility(numServices, ex2_period, ex2_wcet) == TRUE)
        printf("LLF FEASIBLE\n");
```

```
    else
        printf("LLF INFEASIBLE\n");

    printf("************************************************\n");
    printf("************************************************\n");
    printf("******** Scheduling Point Feasibility Example\n");
    printf("************************************************\n");
    printf("************************************************\n\n");

    printf("Ex-2 U=%4.2f%% (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13;
T=D): ",
           ((1.0/2.0)*100.0 + (1.0/5.0)*100.0 + (1.0/7.0)*100.0 +
(2.0/13.0)*100.0));

    if(scheduling_point_feasibility(numServices, ex2_period, ex2_wcet,
ex2_period) == TRUE)
        printf("FEASIBLE\n");
    else
        printf("INFEASIBLE\n");

    return 0;
}

int rate_monotonic_least_upper_bound(U32_T numServices, U32_T period[], U32_T
wcet[], U32_T deadline[])
{
    double utility_sum = 0.0, lub;
    printf("for %d, utility_sum = %lf\n", numServices, utility_sum);
    for(int idx = 0; idx < numServices; idx++)
    {
        utility_sum += (double)wcet[idx] / period[idx];
        printf("for %d, wcet=%lf, period=%lf, utility_sum = %lf\n", idx,
(double)wcet[idx], (double)period[idx], utility_sum);
    }
    lub = numServices * (pow(2.0, 1.0 / numServices) - 1.0);
    printf("utility_sum = %lf\n", utility_sum);
    printf("LUB = %lf\n", lub);
    return utility_sum <= lub ? TRUE : FALSE;
}

int completion_time_feasibility(U32_T numServices, U32_T period[], U32_T
wcet[], U32_T deadline[])
{
    for (int i = 0; i < numServices; i++)
    {
        U32_T an = 0, anext;
        for (int j = 0; j <= i; j++)
        {
            an += wcet[j];
        }
        do
        {
            anext = wcet[i];
            for (int j = 0; j < i; j++)
```

```
            {
                anext += ceil((double)an / period[j]) * wcet[j];
            }
            if (anext == an)
                break;
            an = anext;
        } while (1);

        if (an > deadline[i])
            return FALSE;
    }
    return TRUE;
}

int scheduling_point_feasibility(U32_T numServices, U32_T period[], U32_T
wcet[], U32_T deadline[])
{
    int rc = TRUE;
    for (int i = 0; i < numServices; i++)
    {
        int status = 0;
        for (int k = 0; k <= i; k++)
        {
            for (int l = 1; l <= floor((double)period[i] / period[k]); l++)
            {
                U32_T temp = 0;
                for (int j = 0; j <= i; j++)
                {
                    temp += wcet[j] * ceil((double)(l * period[k]) /
period[j]);
                }
                if (temp <= (l * period[k]))
                {
                    status = 1;
                    break;
                }
            }
            if (status) break;
        }
        if (!status) rc = FALSE;
    }
    return rc;
}

int edf_feasibility(U32_T numServices, U32_T period[], U32_T wcet[]) {
    double totalUtilization = 0.0;
    for (int i = 0; i < numServices; i++) {
        totalUtilization += (double)wcet[i] / period[i];
    }
    printf("EDF Total Utilization: %f\n", totalUtilization);
    return totalUtilization <= 1.0 ? TRUE : FALSE;
}

int llf_feasibility(U32_T numServices, U32_T period[], U32_T wcet[]) {
```

```
    double totalUtilization = 0.0;
    for (int i = 0; i < numServices; i++) {
        totalUtilization += (double)wcet[i] / period[i];
    }
    printf("LLF Total Utilization: %f\n", totalUtilization);
    return totalUtilization <= 1.0 ? TRUE : FALSE;
}
```

10.3. TIVA FreeRTES Code

```
/*****************************************************************************
 * ===========================================================================
 *
 * File: freertes_demo.c
 *
 * Author: Kiran Jojare, Ayswariya Kannan
 *
 * Project Name: Stop Sign Detection Bot on TIVA using FreeRTOS
 *
 * Description:
 * This file contains the implementation of a stop sign detection system
 * for a robotic vehicle controlled by the TIVA TM4C123GH6PM microcontroller,
 * utilizing FreeRTOS for task management. The project aims to demonstrate
 * real-time detection of stop signs, enabling the vehicle to make autonomous
 * navigation decisions.
 *
 * The code utilizes hardware interfacing examples provided by TIVAWare and
 * is designed to handle tasks such as image acquisition from a camera module,
 * image processing to detect stop signs, and motor control for vehicle
movement.
 *
 * References:
 * - TIVAWare demo examples for initial hardware configuration and basic
examples.
 * - UART communication on TM4C123:
https://github.com/Reddimus/UARTcommunication-TM4C123
 * - PWM Control for TM4C123: https://github.com/Mohammed-
AhmedAF/PWMController_TM4C123
 * - MotorLib Embedded for TIVA:
https://github.com/GrandviewIoT/MotorLib_Embedded
 *
 *
 * Subject: ECEN - 5623 Real Time Operating Systems
 *
 * Professor: Tim Scherr
 *
 * University: University of Colorado, Boulder
 *
 * Academic Year: [2023-2024]
 *
 * ===========================================================================
 *****************************************************************************/
```

```
#include <stdbool.h>
#include <stdint.h>
#include "FreeRTOS.h"          // Include main header for FreeRTOS.
#include "FreeRTOSConfig.h"    // Include FreeRTOS configuration settings.
#include "task.h"              // Include for task management utilities.
#include "queue.h"             // Include for queue management utilities.
#include "semphr.h"            // Include for semaphore management utilities.
#include "timers.h"            // Include for software timer utilities.
#include "inc/hw_types.h"      // Hardware specific header file to interact
with the lower level hardware specific functions.
#include "inc/hw_memmap.h"     // Header file defining the memory map of the
TIVA microcontroller.
#include "driverlib/sysctl.h"  // Include system control driver for system and
clock control utilities.
#include "driverlib/interrupt.h" // Include for NVIC interrupt controller
utilities.
#include "driverlib/timer.h"   // Include for hardware timer utilities.
#include "driverlib/gpio.h"    // Include for GPIO pin control utilities.
#include "driverlib/uart.h"    // Include for UART communication utilities.
#include "driverlib/pin_map.h" // Include for GPIO pin configuration that maps
to specific functions.
#include "utils/uartstdio.h"   // Include for using UART functions for standard
input and output.
#include "inc/hw_ints.h"       // Include for definitions of interrupt
assignments.
#include "driverlib/rom.h"     // Include for ROM utility functions.
#include "driverlib/pwm.h"     // Include for Pulse Width Modulation signal
generation utilities.
#include "driverlib/pin_map.h" // Include to define alternate functions for
GPIO pins.

// Define constants for use in timing analysis and other features.
#define TIMING_ANALYSIS         1
#define FIBONACCI_ITERATIONS    5000 // Define the number of Fibonacci sequence
iterations for test purposes.
#define MAX_SERVICE_EXECUTIONS  100  // Assuming a maximum of 180 executions
for demonstration purposes.

// UART and motor configuration settings
#define MOTOR1_GPIO_PERIPH       SYSCTL_PERIPH_GPIOB
#define MOTOR1_GPIO_BASE         GPIO_PORTB_BASE
#define MOTOR1_PIN_A1            GPIO_PIN_0
#define MOTOR1_PIN_A2            GPIO_PIN_1

#define MOTOR2_GPIO_PERIPH       SYSCTL_PERIPH_GPIOC
#define MOTOR2_GPIO_BASE         GPIO_PORTC_BASE
#define MOTOR2_PIN_B1            GPIO_PIN_6          // Motor 2 Pin B1
connected to PC6
#define MOTOR2_PIN_B2            GPIO_PIN_7          // Motor 2 Pin B2
connected to PC7

#define PWM_FREQUENCY           20000  // Set PWM frequency in Hz.

#define UART1_RX_PERIPH         SYSCTL_PERIPH_UART1
```

```
#define UART1_RX_BASE           UART1_BASE
#define UART1_RX_PORT_PERIPH    SYSCTL_PERIPH_GPIOC
#define UART1_RX_PORT_BASE      GPIO_PORTC_BASE
#define UART1_RX_PIN            GPIO_PIN_4
#define UART1_RX_PIN_CONF       GPIO_PC4_U1RX

#define UART2_TX_PERIPH         SYSCTL_PERIPH_UART2
#define UART2_TX_BASE           UART2_BASE
#define UART2_TX_PORT_PERIPH    SYSCTL_PERIPH_GPIOD
#define UART2_TX_PORT_BASE      GPIO_PORTD_BASE
#define UART2_TX_PIN            GPIO_PIN_7
#define UART2_TX_PIN_CONF       GPIO_PD7_U2TX

#define UART2_BAUDRATE          (115200)    // Set baud rate for UART.

// System clock rate definition.
#define SYSTEM_CLOCK            50000000U   // System clock rate in Hz.

// LED configuration details.
#define LED_PERIPH              SYSCTL_PERIPH_GPIOF
#define LED_PORT                GPIO_PORTF_BASE
#define LED_PIN                 GPIO_PIN_2  // Blue LED on TIVA boards.

#define BYTES_PER_TRANS         8  // Define number of bytes per UART
transmission.

// Flags to indicate whether a service should be aborted.
static bool abortS1=false, abortS2=false, abortS3=false, abortS4=false;

 ////////////////////////////////////////////////////////////////////////
 ////////////////////    Function Declarations     //////////////////////
 ////////////////////////////////////////////////////////////////////////

 // System configuration functions.
 void SystemConfig();          // Configures the overall system settings, like
clock and power.
 void GPIOConfig();            // Configures the general purpose input/output
pins for the system.
 void UART0Config(void);       // Configures UART0 for serial communication.
 void TimerConfig();           // Sets up the hardware timers for timing and
delays.
 void SemaphoresConfig(void);  // Initializes semaphores used for task
synchronization.
 void TaskConfig();            // Sets up and creates FreeRTOS tasks.

 // ISR's and UART Function Prototypes
 void ConfigureUARTJetson(void);   // Sets up UART communication for a specific
device, like a Jetson board.
 void UART1IntHandler(void);       // Interrupt handler for UART1, processes
UART1 communication interrupts.
 void UART2IntHandler(void);       // Interrupt handler for UART2, handles
interrupts from UART2.

 // Motor Function Prototypes
```

```
 void ConfigurePWM(uint32_t pwmPeriod);    // Configures PWM settings for motor
control.
 void ConfigureMotorGPIO(void);            // Sets up GPIO for motor control.
 void MotorForward(void);                  // Commands motors to drive forward.
 void MotorReverse(void);                  // Commands motors to reverse
direction.
 void MotorStop(void);                     // Stops the motors.
 void MotorStart(void);                    // Starts the motors.

 // Task Function Prototypes
 void CameraUARTService1(void *pvParameters);  // FreeRTOS task function for
handling UART communication.
 void Motor1Service2(void *pvParameters);      // Task function for controlling
Motor 1.
 void Motor2Service3(void *pvParameters);      // Task function for controlling
Motor 2.
 void DiagnosticsLEDService4(void *pvParameters); // Task function for LED
diagnostics.

 // Declaration of Semaphore Handles
 SemaphoreHandle_t semaphore1, semaphore2, semaphore3, semaphore4; //
Semaphores for synchronizing tasks and ISR.

 volatile TickType_t maxExecutionTimeS1 = 0; // Tracks maximum execution time
for service 1.
 volatile TickType_t maxExecutionTimeS2 = 0; // Tracks maximum execution time
for service 2.
 volatile TickType_t maxExecutionTimeS3 = 0; // Tracks maximum execution time
for service 3.
 volatile TickType_t maxExecutionTimeS4 = 0; // Tracks maximum execution time
for service 4.

 SemaphoreHandle_t semaphoreUART; // Semaphore for UART communication
synchronization.

 // Global Shared Data Between ISR and Service 1 of CameraUARTService
 volatile uint8_t lastReceivedByte;  // Last byte received from UART, shared
globally.
 volatile bool newDataAvailable = false;  // Flag to indicate new data is
available from UART.

 int seqCnt = 0;  // Counter used for sequencing in ISRs or timed events.

 // Data structure for tracking service execution timing.
 typedef struct {
     TickType_t* startTime;      // Array of start times for each execution.
     TickType_t* endTime;        // Array of end times for each execution.
     uint32_t serviceCount;      // Number of times the service has been
executed.
     TickType_t wcet;            // Worst-case execution time.
 } ServiceData;

 // Initialize ServiceData for Service 1
 ServiceData serviceData1 = {0};
```

```
ServiceData serviceData2 = {0};
ServiceData serviceData3 = {0};
ServiceData serviceData4 = {0};

uint32_t idx = 0, jdx = 1;
uint32_t fib = 0, fib0 = 0, fib1 = 1;  // Variables for Fibonacci calculation.

#define FIB_TEST(seqCnt, iterCnt)    \
 for(idx=0; idx < iterCnt; idx++)    \
 {                                   \
     fib = fib0 + fib1;              \
     while(jdx < seqCnt)             \
     {                               \
         fib0 = fib1;                \
         fib1 = fib;                 \
         fib = fib0 + fib1;          \
         jdx++;                      \
 }                                   \
}

// Initializes service data for dynamic allocation of timing data.
void InitServiceData(ServiceData* serviceData, uint32_t maxExecutions) {
    serviceData->startTime = pvPortMalloc(maxExecutions * sizeof(TickType_t));
    serviceData->endTime = pvPortMalloc(maxExecutions * sizeof(TickType_t));
    serviceData->serviceCount = 0;
    serviceData->wcet = 0;
}

// Deinitializes service data, freeing allocated memory.
void DeinitServiceData(ServiceData* serviceData) {
    vPortFree(serviceData->startTime);
    vPortFree(serviceData->endTime);
}

#ifdef DEBUG
void __error__(char *pcFilename, uint32_t ui32Line) { }
#endif

void vApplicationStackOverflowHook(xTaskHandle *pxTask, char *pcTaskName)
{
    while (1)
    {
    }
}

void Timer0AInterruptHandler(void) {

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // Clear the timer interrupt.
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Increment the sequencer count.
    seqCnt++;
```

```
    // Check if it's time to abort all tasks.
    // run for total of 10 seconds
    if (seqCnt >= 1000) {
        // Set abort flags for all tasks.
        abortS1 = true; abortS2 = true; abortS3 = true;abortS4 = true;

        // Release semaphores to unblock tasks for clean up.
        xSemaphoreGiveFromISR(semaphore1, &xHigherPriorityTaskWoken);
        xSemaphoreGiveFromISR(semaphore2, &xHigherPriorityTaskWoken);
        xSemaphoreGiveFromISR(semaphore3, &xHigherPriorityTaskWoken);
        xSemaphoreGiveFromISR(semaphore4, &xHigherPriorityTaskWoken);

        MotorStop();

    } else {
        // Service_1 - 5 Hz, every 50nd Sequencer loop
        if ((seqCnt % 20) == 0) {
            if(xSemaphoreGiveFromISR(semaphore1, &xHigherPriorityTaskWoken) !=
pdTRUE) {
                // Handle error
                UARTprintf("Semaphore give for Service 1 failed!\n");
            }
        }

        // Service_2 - 100 Hz, every 1th Sequencer loop
        if ((seqCnt % 1) == 0) {
            if(xSemaphoreGiveFromISR(semaphore2, &xHigherPriorityTaskWoken) !=
pdTRUE) {
                // Handle error
                UARTprintf("Semaphore give for Service 2 failed!\n");
            }
        }

        // Service_3 -100 Hz, every 1th Sequencer loop
        if ((seqCnt % 1) == 0) {
            if(xSemaphoreGiveFromISR(semaphore3, &xHigherPriorityTaskWoken) !=
pdTRUE) {
                // Handle error
                UARTprintf("Semaphore give for Service 3 failed!\n");
            }
        }

        // Service_4 - 4 Hz, every 50th Sequencer loop
        if ((seqCnt % 25) == 0) {
            if(xSemaphoreGiveFromISR(semaphore4, &xHigherPriorityTaskWoken) !=
pdTRUE) {
                // Handle error
                UARTprintf("Semaphore give for Service 4 failed!\n");
            }
        }

    }
```

```
    // Toggle the LED as a visual heartbeat indicator.
    // GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, ~GPIOPinRead(GPIO_PORTF_BASE,
GPIO_PIN_2));

    // Force a context switch if xHigherPriorityTaskWoken was set to true.
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}

//////////////////////////////////////////////////////////////////////////
//////////////////       Main Entry Point       ////////////////////////
//////////////////////////////////////////////////////////////////////////

int main(void)
{
    SystemConfig();

    GPIOConfig();

    SemaphoresConfig();

    UART0Config();

    // Initialize PWM and GPIO configurations
    uint32_t pwmPeriod = ROM_SysCtlClockGet() / PWM_FREQUENCY;
    ConfigurePWM(pwmPeriod);
    ConfigureMotorGPIO();

    // Start the motor system
    MotorStart();

    // Drive the motor forward
    MotorForward();

    //Giving initial semaphore
    xSemaphoreGive(semaphoreUART);

    UARTprintf("STOP SIGN DETECTION BOT RUNNING.............\n");

    TimerConfig();

    TaskConfig();

    ConfigureUARTJetson();

    vTaskStartScheduler();

    while (1)
    {
        // Main loop does nothing but sleep
        // SysCtlSleep();

        ;
    }
}
```

```
//////////////////////////////////////////////////////////////////////////
//////////////////    Function Definitions      ///////////////////////////
//////////////////////////////////////////////////////////////////////////

/**
 * Configures the system clock for the microcontroller.
 */
void SystemConfig()
{
    // Set the system clock to 50 MHz from PLL with a crystal reference of 16
MHz
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ |
SYSCTL_OSC_MAIN);
}

/**
 * Configures general-purpose input/output settings for the system,
particularly for LEDs.
 */
void GPIOConfig()
{
    // The following code is commented out but when enabled, it would:
    // Enable the GPIO port for the LED (PF2)
    // SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

    // Configure the GPIO pin for the LED as an output
    // GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2);
}

/**
 * Configures UART0 for basic serial communication.
 */
void UART0Config(void)
{
    // Enable GPIO Peripheral for UART0 pins and UART0 itself
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    // Configure the GPIO pins for UART mode
    GPIOPinConfigure(GPIO_PA0_U0RX);  // Configure PA0 as UART0 RX
    GPIOPinConfigure(GPIO_PA1_U0TX);  // Configure PA1 as UART0 TX
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    // Set UART clock source and configure the UART for 115200 baud rate
    // UART_CLOCK_PIOSC is often used to avoid conflicts with the system clock
used by SysCtlClockSet
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);
    UARTStdioConfig(0, 115200, 16000000); // Use internal 16MHz oscillator for
UART
}

/**
```

```
 * Configures the system timer to trigger an interrupt at a frequency of 100
Hz.
 */
void TimerConfig()
{
    // Enable Timer 0 peripheral
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

    // Enable processor interrupts
    IntMasterEnable();

    // Configure Timer0 as a 32-bit timer in periodic mode
    TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);

    // Calculate and set the timer load value for a 100 Hz frequency
    uint32_t timerLoad = (SysCtlClockGet() / 100) - 1; // System clock divided
by frequency, minus one for zero-based index
    TimerLoadSet(TIMER0_BASE, TIMER_A, timerLoad);

    // Enable the specific Timer0A interrupt
    IntEnable(INT_TIMER0A);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Register the interrupt handler for Timer0A
    TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0AInterruptHandler);

    // Start Timer0A
    TimerEnable(TIMER0_BASE, TIMER_A);
}

/**
 * Initializes semaphores used for synchronizing tasks and interrupts.
 */
void SemaphoresConfig(void) {
    // Create binary semaphores for task synchronization and error check
    semaphore1 = xSemaphoreCreateBinary();
    if (semaphore1 == NULL) { UARTprintf("Error: Failed to create Semaphore
1\n"); }

    semaphore2 = xSemaphoreCreateBinary();
    if (semaphore2 == NULL) { UARTprintf("Error: Failed to create Semaphore
2\n"); }

    semaphore3 = xSemaphoreCreateBinary();
    if (semaphore3 == NULL) { UARTprintf("Error: Failed to create Semaphore
3\n"); }

    semaphore4 = xSemaphoreCreateBinary();
    if (semaphore4 == NULL) { UARTprintf("Error: Failed to create Semaphore
4\n"); }

    semaphoreUART = xSemaphoreCreateBinary();
    if (semaphoreUART == NULL) { UARTprintf("Error: Failed to create UART
Semaphore\n"); }
```

```
}

/**
 * Configures and creates tasks.
 */
void TaskConfig()
{
    BaseType_t status;

    // Create a task for handling UART communication with the camera module
    status = xTaskCreate(CameraUARTService1, "CameraUARTService1", 100, NULL,
configMAX_PRIORITIES - 2, NULL);
    if (status != pdTRUE) { UARTprintf("Error: Failed to create Camera UART
Service 1\n"); }

    // Create a task for controlling Motor 1
    status = xTaskCreate(Motor1Service2, "Motor1Service2", 128, NULL,
configMAX_PRIORITIES - 1, NULL);
    if (status != pdTRUE) { UARTprintf("Error: Failed to create Motor 1 Service
2\n"); }

    // Create a task for controlling Motor 2
    status = xTaskCreate(Motor2Service3, "Motor2Service3", 128, NULL,
configMAX_PRIORITIES - 1, NULL);
    if (status != pdTRUE) { UARTprintf("Error: Failed to create Motor 2 Service
3\n"); }

    // Create a task for managing diagnostic LEDs
    status = xTaskCreate(DiagnosticsLEDService4, "DiagnosticsLEDService4", 128,
NULL, configMAX_PRIORITIES - 3, NULL);
    if (status != pdTRUE) { UARTprintf("Error: Failed to create Diagnostics LED
Service 4\n"); }
}
/**
 * Configures UART peripherals for communication with Jetson (or other UART
devices).
 * This function sets up UART1 for receiving data and UART2 for transmitting
data.
 */
void ConfigureUARTJetson(void) {
    // Enable UART1 and UART2 peripherals to ensure they are powered and ready
for configuration.
    SysCtlPeripheralEnable(UART1_RX_PERIPH);
    SysCtlPeripheralEnable(UART1_RX_PORT_PERIPH);
    SysCtlPeripheralEnable(UART2_TX_PERIPH);
    SysCtlPeripheralEnable(UART2_TX_PORT_PERIPH);

    // Configure the pin associated with UART1_RX for UART functionality.
    GPIOPinConfigure(UART1_RX_PIN_CONF);
    GPIOPinTypeUART(UART1_RX_PORT_BASE, UART1_RX_PIN);

    // Configure the pin associated with UART2_TX for UART functionality.
    GPIOPinConfigure(UART2_TX_PIN_CONF);
    GPIOPinTypeUART(UART2_TX_PORT_BASE, UART2_TX_PIN);
```

```
    // Set up UART1 for reception at 115200 baud, 8 data bits, 1 stop bit, and
no parity.
    UARTConfigSetExpClk(UART1_RX_BASE, SysCtlClockGet(), 115200,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
UART_CONFIG_PAR_NONE));
    UARTIntEnable(UART1_RX_BASE, UART_INT_RX);
    UARTEnable(UART1_RX_BASE);

    // Set up UART2 for transmission at 115200 baud, 8 data bits, 1 stop bit,
and no parity.
    UARTConfigSetExpClk(UART2_TX_BASE, SysCtlClockGet(), 115200,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
UART_CONFIG_PAR_NONE));
    UARTEnable(UART2_TX_BASE);

    // Register and enable interrupt handler for UART1_RX to handle incoming
data.
    UARTIntRegister(UART1_RX_BASE, UART1IntHandler);
    IntEnable(INT_UART1);

    // Register and enable interrupt handler for UART2_TX (optional here, not
utilized in this function).
    UARTIntRegister(UART2_TX_BASE, UART2IntHandler);
    IntEnable(INT_UART2);
}

/**
 * Interrupt handler for UART1. This function processes incoming data as soon
as it is available.
 */
void UART1IntHandler(void) {
    // Get the current interrupt status and clear it.
    uint32_t ui32Status = UARTIntStatus(UART1_RX_BASE, true);
    UARTIntClear(UART1_RX_BASE, ui32Status);

    // Continue reading data while characters are available in the UART buffer.
    while (UARTCharsAvail(UART1_RX_BASE)) {
        // Read the incoming byte and store it globally.
        lastReceivedByte = UARTCharGet(UART1_RX_BASE);
        newDataAvailable = true;  // Flag to indicate new data is ready to be
processed.
    }
}

/**
 * Placeholder for UART2 interrupt handler. Currently, it does nothing but is
required for completeness.
 */
void UART2IntHandler(void) {
    // Currently empty, as no specific functionality is implemented for UART2
TX interrupts.
}
```

```
/**
 * Configures PWM (Pulse Width Modulation) for controlling motor speed.
 * This function sets up PWM generators for two motors.
 * @param pwmPeriod The period of the PWM signal, which determines the
frequency.
 */
void ConfigurePWM(uint32_t pwmPeriod) {
    // Set the PWM clock divider to 1 for full speed.
    ROM_SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

    // Enable the PWM peripherals for motor control.
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1); // PWM1 for Motor 1
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0); // PWM0 for Motor 2

    // Enable the GPIO peripherals that the PWM pins are multiplexed on.
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF); // GPIOF for Motor 1
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB); // GPIOB for Motor 2

    // Set the specific pins to be used as PWM outputs.
    ROM_GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_0); // PF0 for Motor 1
    ROM_GPIOPinTypePWM(GPIO_PORTB_BASE, GPIO_PIN_4); // PB4 for Motor 2

    // Configure pin muxing for PWM output on these pins.
    ROM_GPIOPinConfigure(GPIO_PF0_M1PWM4); // PF0 as M1PWM4 for Motor 1
    ROM_GPIOPinConfigure(GPIO_PB4_M0PWM2); // PB4 as M0PWM2 for Motor 2

    // Configure and enable PWM generator for Motor 1.
    ROM_PWMGenConfigure(PWM1_BASE, PWM_GEN_2, PWM_GEN_MODE_DOWN |
PWM_GEN_MODE_NO_SYNC);
    ROM_PWMGenPeriodSet(PWM1_BASE, PWM_GEN_2, pwmPeriod);
    ROM_PWMPulseWidthSet(PWM1_BASE, PWM_OUT_4, pwmPeriod / 2);  // Set to 50%
duty cycle
    ROM_PWMOutputState(PWM1_BASE, PWM_OUT_4_BIT, true);
    ROM_PWMGenEnable(PWM1_BASE, PWM_GEN_2);

    // Configure and enable PWM generator for Motor 2.
    ROM_PWMGenConfigure(PWM0_BASE, PWM_GEN_1, PWM_GEN_MODE_DOWN |
PWM_GEN_MODE_NO_SYNC);
    ROM_PWMGenPeriodSet(PWM0_BASE, PWM_GEN_1, pwmPeriod);
    ROM_PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, pwmPeriod / 2);  // Set to 50%
duty cycle
    ROM_PWMOutputState(PWM0_BASE, PWM_OUT_2_BIT, true);
    ROM_PWMGenEnable(PWM0_BASE, PWM_GEN_1);
}

/**
 * Configures GPIOs for motor control, setting up pins as output for
controlling motor direction.
 */
void ConfigureMotorGPIO(void) {
    // Enable the GPIO peripheral for Motor 1 and configure its pins as
outputs.
    ROM_SysCtlPeripheralEnable(MOTOR1_GPIO_PERIPH);
    ROM_GPIOPinTypeGPIOOutput(MOTOR1_GPIO_BASE, MOTOR1_PIN_A1 | MOTOR1_PIN_A2);
```

```
    ROM_GPIOPinWrite(MOTOR1_GPIO_BASE, MOTOR1_PIN_A1 | MOTOR1_PIN_A2, 0); //
Initialize pins to low.

    // Enable the GPIO peripheral for Motor 2 and configure its pins as
outputs.
    ROM_SysCtlPeripheralEnable(MOTOR2_GPIO_PERIPH);
    ROM_GPIOPinTypeGPIOOutput(MOTOR2_GPIO_BASE, MOTOR2_PIN_B1 | MOTOR2_PIN_B2);
    ROM_GPIOPinWrite(MOTOR2_GPIO_BASE, MOTOR2_PIN_B1 | MOTOR2_PIN_B2, 0); //
Initialize pins to low.
}


/**
 * Sets the direction of both motors to forward.
 * Motor 1 and Motor 2 are configured to rotate forward by setting appropriate
GPIO outputs.
 */
void MotorForward(void) {
    // Set Motor 1 to move forward by activating one pin and deactivating the
other.
    ROM_GPIOPinWrite(MOTOR1_GPIO_BASE, MOTOR1_PIN_A1, MOTOR1_PIN_A1);  //
Activate Motor 1 Pin A1
    ROM_GPIOPinWrite(MOTOR1_GPIO_BASE, MOTOR1_PIN_A2, 0);             //
Deactivate Motor 1 Pin A2

    // Set Motor 2 to move forward by activating one pin and deactivating the
other.
    ROM_GPIOPinWrite(MOTOR2_GPIO_BASE, MOTOR2_PIN_B1, MOTOR2_PIN_B1);  //
Activate Motor 2 Pin B1
    ROM_GPIOPinWrite(MOTOR2_GPIO_BASE, MOTOR2_PIN_B2, 0);              //
Deactivate Motor 2 Pin B2
}

/**
 * Sets the direction of both motors to reverse.
 * Motor 1 and Motor 2 are configured to rotate in the reverse direction by
setting appropriate GPIO outputs.
 */
void MotorReverse(void) {
    // Set Motor 1 to move in reverse by activating one pin and deactivating
the other.
    ROM_GPIOPinWrite(MOTOR1_GPIO_BASE, MOTOR1_PIN_A2, MOTOR1_PIN_A2);  //
Activate Motor 1 Pin A2
    ROM_GPIOPinWrite(MOTOR1_GPIO_BASE, MOTOR1_PIN_A1, 0);              //
Deactivate Motor 1 Pin A1

    // Set Motor 2 to move in reverse by activating one pin and deactivating
the other.
    ROM_GPIOPinWrite(MOTOR2_GPIO_BASE, MOTOR2_PIN_B2, MOTOR2_PIN_B2);  //
Activate Motor 2 Pin B2
    ROM_GPIOPinWrite(MOTOR2_GPIO_BASE, MOTOR2_PIN_B1, 0);              //
Deactivate Motor 2 Pin B1
}
```

```
/**
 * Stops both motors by setting both control pins of each motor to low.
 * This function ensures that both motors are in a stopped state with no
voltage difference across the motor terminals.
 */
void MotorStop(void) {
    // Stop Motor 1 by setting both control pins low.
    ROM_GPIOPinWrite(MOTOR1_GPIO_BASE, MOTOR1_PIN_A1 | MOTOR1_PIN_A2, 0);

    // Stop Motor 2 by setting both control pins low.
    ROM_GPIOPinWrite(MOTOR2_GPIO_BASE, MOTOR2_PIN_B1 | MOTOR2_PIN_B2, 0);
}

/**
 * Starts or restarts the motors by first ensuring they are stopped and then
setting them to move forward.
 * This function can be used to safely start the motors during system
initialization or after an emergency stop.
 */
void MotorStart(void) {
    // Ensure that both motors are stopped before starting.
    MotorStop();

    // Optionally start the motors moving forward. Uncomment the next line to
enable this behavior.
    // MotorForward();  // Uncomment to start motors in forward direction by
default
}


////////////////////////////////////////////////////////////////////////////
////////////////    Task Function Definitions      ///////////////////////
////////////////////////////////////////////////////////////////////////////

void CameraUARTService1(void* pvParameters) {
    uint32_t estimatedMaxExecutions = 20; // Adjust based on expected maximum
    InitServiceData(&serviceData1, estimatedMaxExecutions);

    while (!abortS1) {
        if (xSemaphoreTake(semaphore1, portMAX_DELAY) == pdPASS) {
            serviceData1.startTime[serviceData1.serviceCount] =
xTaskGetTickCount();

            if (newDataAvailable) {
                taskENTER_CRITICAL();
                uint8_t data = lastReceivedByte;  // Copy to local variable to
minimize time in critical section
                newDataAvailable = false;
                taskEXIT_CRITICAL();

                TickType_t currentTime = xTaskGetTickCount();
                UARTprintf("[%u ms] [CameraUARTService1] Received Byte:
0x%02X\n", currentTime, data);
```

```
                switch(data) {
                    case 0xAA:
                        UARTprintf("[%u ms] [CameraUARTService1] Alert: STOP
Sign Detected - Vehicle HALTED\n", currentTime);
                        break;
                    case 0x00:
                        UARTprintf("[%u ms] [CameraUARTService1] Info: Path
Clear - Vehicle Continuing\n", currentTime);
                        break;
                    default:
                        UARTprintf("[%u ms] [CameraUARTService1] Warning:
Unknown Command 0x%02X - No Action Taken\n", currentTime, data);
                        break;
                }
            }

            serviceData1.endTime[serviceData1.serviceCount] =
xTaskGetTickCount();
            serviceData1.serviceCount++;

            TickType_t executionTime =
serviceData1.endTime[serviceData1.serviceCount - 1] -
serviceData1.startTime[serviceData1.serviceCount - 1];
            if (executionTime > serviceData1.wcet) {
                serviceData1.wcet = executionTime;
            }
        }
    }

    if (xSemaphoreTake(semaphoreUART, portMAX_DELAY) == pdPASS) {
#if TIMING_ANALYSIS==1
        int i = 0;
        for (i = 0; i < serviceData1.serviceCount; i++) {
            UARTprintf("[%u ms] [CameraUARTService1] Execution %u - Start: %u
ms, End: %u ms, Execution Time: %u ms\n",
                       xTaskGetTickCount(), i + 1, serviceData1.startTime[i],
serviceData1.endTime[i], serviceData1.endTime[i] - serviceData1.startTime[i]);
        }
#endif
        UARTprintf("[%u ms] [CameraUARTService1] Summary: Total Executions: %u,
WCET: %u ms\n",
                   xTaskGetTickCount(), serviceData1.serviceCount,
serviceData1.wcet);

        xSemaphoreGive(semaphoreUART);
    }

    DeinitServiceData(&serviceData1);
    vTaskDelete(NULL);
}

void Motor1Service2(void* pvParameters) {
    uint32_t estimatedMaxExecutions = 20; // Adjust based on expected maximum
    InitServiceData(&serviceData2, estimatedMaxExecutions);
```

```
    while (!abortS2) {
        if (xSemaphoreTake(semaphore2, portMAX_DELAY) == pdPASS) {
            serviceData2.startTime[serviceData2.serviceCount] =
xTaskGetTickCount();

            if (newDataAvailable) {
                taskENTER_CRITICAL();
                uint8_t command = lastReceivedByte;
                newDataAvailable = false;
                taskEXIT_CRITICAL();

                TickType_t currentTime = xTaskGetTickCount();
                if (command == 0xAA) {  // If STOP sign detected
                    MotorStop();  // Stop the motor
                    UARTprintf("[%u ms] STOP Sign Detected - Motor Stopped.\n",
currentTime);
                } else if (command == 0x00) {  // If STOP sign cleared
                    MotorForward();  // Resume the motor forward
                    UARTprintf("[%u ms] [Motor1Service2] Path Clear - Motor
Resumed Forward.\n", currentTime);
                }
            }

            serviceData2.endTime[serviceData2.serviceCount] =
xTaskGetTickCount();
            serviceData2.serviceCount++;

            TickType_t executionTime =
serviceData2.endTime[serviceData2.serviceCount - 1] -
serviceData2.startTime[serviceData2.serviceCount - 1];
            if (executionTime > serviceData2.wcet) {
                serviceData2.wcet = executionTime;
            }
        }
    }

    if (xSemaphoreTake(semaphoreUART, portMAX_DELAY) == pdPASS) {
#if TIMING_ANALYSIS==1
        int i = 0;
        for (i = 0; i < serviceData2.serviceCount; i++) {
            UARTprintf("[%u ms] [Motor1Service2] Execution %u - Start: %u ms,
End: %u ms, Execution Time: %u ms\n",
                       xTaskGetTickCount(), i + 1, serviceData2.startTime[i],
serviceData2.endTime[i], serviceData2.endTime[i] - serviceData2.startTime[i]);
        }
#endif
        UARTprintf("[%u ms] [Motor1Service2] Summary: Total Executions: %u,
WCET: %u ms\n",
                   xTaskGetTickCount(), serviceData2.serviceCount,
serviceData2.wcet);

        xSemaphoreGive(semaphoreUART);
    }
```

```
    DeinitServiceData(&serviceData2);
    vTaskDelete(NULL);
}



void Motor2Service3(void* pvParameters) {
    uint32_t estimatedMaxExecutions = 20; // Adjust based on expected maximum
    InitServiceData(&serviceData3, estimatedMaxExecutions);

    while (!abortS3) {
        if (xSemaphoreTake(semaphore3, portMAX_DELAY) == pdPASS) {
            serviceData3.startTime[serviceData3.serviceCount] =
xTaskGetTickCount();

            if (newDataAvailable) {
                taskENTER_CRITICAL();
                uint8_t command = lastReceivedByte;
                newDataAvailable = false;
                taskEXIT_CRITICAL();

                TickType_t currentTime = xTaskGetTickCount();
                if (command == 0xAA) {  // If STOP sign detected
                    MotorStop();  // Stop the motor
                    UARTprintf("[%u ms] [Motor2Service3] STOP Sign Detected -
Motor Stopped.\n", currentTime);
                } else if (command == 0x00) {  // If STOP sign cleared
                    MotorForward();  // Resume the motor forward
                    UARTprintf("[%u ms] [Motor2Service3] Path Clear - Motor
Resumed Forward.\n", currentTime);
                }
            }
            // FIB_TEST(47, 2000); // Placeholder for the actual workload

            serviceData3.endTime[serviceData3.serviceCount] =
xTaskGetTickCount();
            serviceData3.serviceCount++;

            TickType_t executionTime =
serviceData3.endTime[serviceData3.serviceCount - 1] -
serviceData3.startTime[serviceData3.serviceCount - 1];
            if (executionTime > serviceData3.wcet) {
                serviceData3.wcet = executionTime;
            }
        }
    }

    if (xSemaphoreTake(semaphoreUART, portMAX_DELAY) == pdPASS) {
#if TIMING_ANALYSIS==1
        int i = 0;
        for (i = 0; i < serviceData3.serviceCount; i++) {
            UARTprintf("[%u ms] [Motor2Service3] Execution %u - Start: %u ms,
End: %u ms, Execution Time: %u ms\n",
```

```
                        xTaskGetTickCount(), i + 1, serviceData3.startTime[i],
serviceData3.endTime[i], serviceData3.endTime[i] - serviceData3.startTime[i]);
        }
#endif
        UARTprintf("[%u ms] [Motor2Service3] Summary: Total Executions: %u,
WCET: %u ms\n",
                   xTaskGetTickCount(), serviceData3.serviceCount,
serviceData3.wcet);

        xSemaphoreGive(semaphoreUART);
    }

    DeinitServiceData(&serviceData3);
    vTaskDelete(NULL);
}

void DiagnosticsLEDService4(void* pvParameters) {
    uint32_t estimatedMaxExecutions = 20; // Adjust based on expected maximum
    InitServiceData(&serviceData4, estimatedMaxExecutions);

    // Initialize the blue LED
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2);

    while (!abortS4) {
        if (xSemaphoreTake(semaphore4, portMAX_DELAY) == pdPASS) {
            serviceData4.startTime[serviceData4.serviceCount] =
xTaskGetTickCount();
            // FIB_TEST(47, 2000); // Placeholder for the actual workload

            if (newDataAvailable) {
                taskENTER_CRITICAL();
                uint8_t command = lastReceivedByte;  // Copy to local variable
to minimize time in critical section
                newDataAvailable = false;
                taskEXIT_CRITICAL();

                if (command == 0xAA) {
                    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);  //
Turn on the blue LED
                    UARTprintf("[%u ms] [DiagnosticsLEDService4] Received
Command AA: Blue LED ON\n", xTaskGetTickCount());
                } else if (command == 0x00) {
                    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);  // Turn off
the blue LED
                    UARTprintf("[%u ms] [DiagnosticsLEDService4] Received
Command 00: Blue LED OFF\n", xTaskGetTickCount());
                }
            }

            serviceData4.endTime[serviceData4.serviceCount] =
xTaskGetTickCount();
            serviceData4.serviceCount++;
```

```
            // Calculate WCET dynamically
            TickType_t executionTime =
serviceData4.endTime[serviceData4.serviceCount - 1] -
serviceData4.startTime[serviceData4.serviceCount - 1];
            if (executionTime > serviceData4.wcet) {
                serviceData4.wcet = executionTime;
            }
        }
    }

    if (xSemaphoreTake(semaphoreUART, portMAX_DELAY) == pdPASS) {
#if TIMING_ANALYSIS==1
        int i = 0;
        for (i = 0; i < serviceData4.serviceCount; i++) {
            UARTprintf("[%u ms] [DiagnosticsLEDService4] Execution %u - Start:
%u ms, End: %u ms, Execution Time: %u ms\n",
                       xTaskGetTickCount(), i + 1, serviceData4.startTime[i],
serviceData4.endTime[i], serviceData4.endTime[i] - serviceData4.startTime[i]);
        }
#endif
        UARTprintf("[%u ms] [DiagnosticsLEDService4] Summary: Total Executions:
%u, WCET: %u ms\n",
                   xTaskGetTickCount(), serviceData4.serviceCount,
serviceData4.wcet);

        xSemaphoreGive(semaphoreUART);
    }

    // Clean up
    DeinitServiceData(&serviceData4);
    vTaskDelete(NULL);
}
```