Menoufia University

Faculty of Electronic Engineering

Department of Computer Science and Engineering

ADAS: Powered Driver Monitoring and Enhanced Child Safety in vehicles

By:

Mohamed Ahmed Salama Ahmed

Osama El-Sayed Mohammedy Gabr

Ahmed Gaber Mahmoud El-Falah

Khaled Abdelhamed Rashad Salama

Mohamed Khairy El-Sayed Atia

Mohamed Anwar Fouad Mahgoub

Supervised By:

Dr. Abdullah Nabil

2023/2024

Menoufia University

Faculty of Electronic Engineering

Department of Computer Science and Engineering

Graduation project 2023/2024

ADAS: Powered Driver Monitoring and Enhanced Child Safety in vehicles

By:

Mohamed Ahmed Salama Ahmed

Osama El-Sayed Mohammedy Gabr

Ahmed Gaber Mahmoud El-Falah

Khaled Abdelhamed Rashad Salama

Mohamed Khairy El-Sayed Atia

Mohamed Anwar Fouad Mahgoub

Supervised By:

Dr. Abdullah Nabil

Dean

Prof. Ayman El-Sayed

Head of the Department

Prof. Mohamed Berbar

Abstract

The advancement of vehicular technology has led to significant improvements in road safety, yet the presence of driver distractions and the potential danger to children left unattended in vehicles remain pressing concerns. This graduation project explores an innovative Driver Monitoring and Child Detection System (DMCDS) designed to enhance vehicle safety through the integration of advanced sensor technologies and intelligent algorithms.

The DMCDS utilizes a combination of cameras and machine learning algorithms to monitor driver behavior and detect the presence of children within the vehicle. By analyzing facial expressions, eye movements, and head positions, the system can identify signs of driver fatigue, distraction, and drowsiness, issuing real-time alerts to prevent potential accidents. Simultaneously, the system employs thermal imaging and motion sensors to ensure that children are not left unattended in the car, thereby mitigating the risks of heatstroke and other hazards.

This project encompasses the design, implementation, and testing phases of the DMCDS, including hardware selection, software development, and the integration of various sensor modalities. Extensive testing has been conducted in both controlled environments and real-world scenarios to validate the system's accuracy and reliability.

The results demonstrate that the DMCDS can significantly enhance vehicular safety by providing timely warnings and interventions, thereby reducing the likelihood of accidents caused by driver inattentiveness and protecting children from potential harm. This project highlights the potential of combining modern sensor technologies with intelligent algorithms to create safer and smarter vehicles, paving the way for future innovations in automotive safety systems.

Table of Contents

Contents

List of Figures

List of Abbreviations

ADAS	Advanced Driver Assistance Systems
DMS	Driver Monitoring Systems
AI	Artificial Intelligence
SSD	Single Shot Multi-Box Detector
YOLO	You Only Look Once
CNN	Convolutional Neural Network
RTOS	Real Time Operating System
AUTOSAR	Automotive Open System Architecture
ECU	Electronic Control Unit
GSM	Global System for Mobile Communications
GPS	Global Positioning System
CAN	Controller Area Network
TDMA	Time division multiple access
UART	Universal Asynchronous Receiver-Transmitter
SMS	Short Message Service
GPRS	General Packet Radio Service
ISR	interrupt service routines
NVIC	Nested Vectored Interrupt Controller
MCAL	Microcontroller Abstraction Layer
ADC	Analog-to-Digital Converter
GPIO	General-Purpose Input/Output
RCC	Reset and Clock Control
HAL	Hardware Abstraction Layer
APP	Application Layer

1. Chapter1 : Introduction

1.1. Introduction

The rapid evolution of automotive technology has significantly transformed the landscape of road safety, integrating advanced systems designed to protect both drivers and passengers. Amid these innovations, the Advanced Driver Assistance Systems (ADAS) have emerged as a critical focus, particularly in enhancing driver monitoring and child safety within vehicles. This project delves into the intricacies of ADAS, emphasizing the necessity of integrating sophisticated technology to safeguard the most vulnerable passengers—children.

1.1.1. Context and Significance

Child safety in vehicles is an urgent global issue. In the United States alone, statistics reveal a disturbing trend: between 1998 and 2017, approximately 800 children under the age of 14 died due to being left unattended in vehicles, often succumbing to heatstroke. On average, this translates to 37 deaths per year, with 53% of these tragedies occurring because a child was forgotten in the car. These alarming figures have propelled legislative bodies, such as the U.S. Congress, to propose bills mandating child safety alert systems in all new passenger motor vehicles.

1.1.2. The Imperative of ADAS

The advent of ADAS represents a pivotal leap in automotive safety, intertwining cutting-edge technology with the imperative of protecting lives. These systems are designed not only to monitor and assist drivers but also to ensure the safety of passengers, particularly children, through enhanced detection and alert mechanisms. The integration of computer vision and artificial intelligence (AI) in ADAS has opened new avenues for real-time monitoring and response, aiming to prevent accidents and save lives.

1.2. Project Overview

The core objective of this project is to develop a comprehensive Advanced Driver Assistance System (ADAS) that integrates state-of-the-art technologies for enhanced driver monitoring and child safety in vehicles. This dual-focus approach aims to address two critical aspects of road safety: the prevention of driver errors and the protection of vulnerable passengers,

particularly children. By leveraging advancements in computer vision and artificial intelligence (AI), the project aspires to create a system that not only improves driving habits but also ensures the safety of children through real-time detection and alerts.

1.3. Enhanced Driver Monitoring

1.3.1. Real-Time Monitoring and Feedback

Driver monitoring systems (DMS) are at the forefront of this project's objectives. Utilizing sophisticated AI algorithms, these systems continuously analyze various parameters related to driver behavior. Key features include monitoring eye movements to detect drowsiness, analyzing facial expressions to gauge attentiveness, and observing overall driving patterns to identify unsafe behaviors. The system provides real-time feedback to the driver, promoting immediate corrective actions and fostering safer driving habits.

1.3.2. Behavioral Analysis:

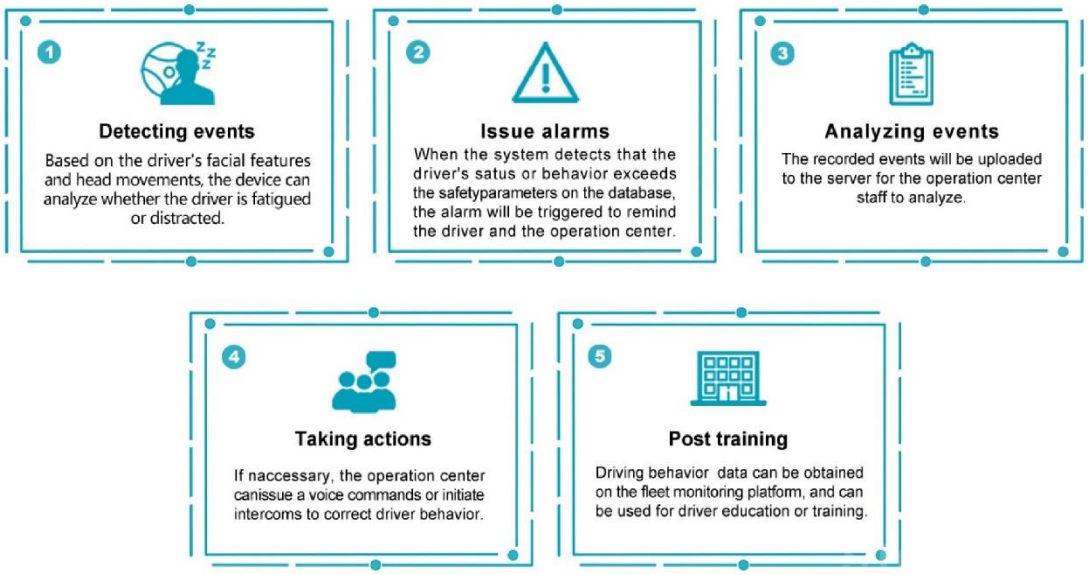


Figure 1 1.3.2 Behavioral Analysis

The incorporation of deep learning models enables the system to learn from vast datasets of driver behavior. This allows for the identification of subtle signs of fatigue or distraction that might not be immediately evident. The AI models are trained to recognize these indicators, providing alerts and recommendations tailored to the

individual driver's needs. Over time, the system adapts to the specific patterns of each driver, enhancing its accuracy and effectiveness.

1.3.3. Integration with Vehicle Systems:

The ADAS integrates seamlessly with the vehicle's existing systems, ensuring comprehensive monitoring without requiring extensive modifications. This integration includes utilizing the vehicle's internal sensors, cameras, and other data sources to provide a holistic view of the driving environment. By doing so, the system can offer precise and actionable insights, significantly reducing the likelihood of accidents caused by human error.

1.4. Technological Foundation

1.4.1. Computer Vision and AI:

The integration of computer vision and AI is central to this project, enabling systems to interpret visual data and make intelligent decisions. Techniques such as deep learning and machine learning are employed to analyze images, recognize patterns, and execute timely interventions. By combining these technologies, the project seeks to create a comprehensive safety net for drivers and passengers alike.

1.4.2. Algorithmic Contributions

The algorithms employed in this project are designed to enhance the accuracy and reliability of child detection and driver monitoring systems. Haar Cascade, known for its real-time face detection capabilities, is adapted to identify children's faces. SSD and YOLO, both renowned for their speed and precision, offer robust solutions for real-time object detection. CNNs, with their feature extraction capabilities, provide a versatile approach to recognizing children in various environments.

1.4.3. Broader Implications

The implementation of ADAS holds profound implications for road safety, promising to significantly reduce accidents and enhance the protection of all vehicle occupants. AI-powered driver monitoring systems not only improve individual driving skills but also contribute to community safety by fostering a culture of accountability and awareness on the roads. The collective adoption of these technologies can lead to safer roads, benefiting entire communities and reducing the burden on emergency services.

1.5. Future Directions

As the project progresses, future work will focus on refining these technologies and expanding their applicability. The development of real-time operating systems (RTOS) and layered architectures such as AUTOSAR will be crucial in ensuring the scalability and reliability of ADAS. Ongoing research and development efforts aim to create a seamless integration of these systems into a wide range of vehicles, enhancing their effectiveness and accessibility.

The journey toward redefining automotive safety through ADAS is a testament to the potential of technology to save lives and create a safer driving environment. By focusing on driver monitoring and enhanced child safety, this project underscores the transformative power of AI and computer vision in addressing critical safety challenges. As we embrace these advancements, we move closer to realizing the vision of accident-free roads, where technology not only assists but actively protects those on the journey.

1.6. Overview of component used

- Raspberry Pi 4
- raspberry pi 7 Touch Screen
- Raspberry Pi Camera Module 3
- Ultrasonic Sensors
- GSM Module
- GPS Module
- STM32 Blue Pill

2. Chapter 2: Problem definition and Project Objectives

2.1. Introduction

In recent years, the integration of advanced driver assistance systems (ADAS) has become a crucial focus in the automotive industry. Our project aims to enhance vehicle safety through a dual-function system: a driver monitoring system and a child detection system. The driver monitoring system identifies 14 different classes of driver behaviors, ranging from normal driving to various forms of distraction. Simultaneously, the child detection system ensures that no child or toddler is left unattended in the vehicle.

2.2. Problem Definition

2.2.1. Driver Monitoring System

Distracted driving is one of the leading causes of traffic accidents worldwide. According to the National Highway Traffic Safety Administration (NHTSA), distracted driving claimed 3,142 lives in 2019 and 3,308 lives in 2022. Data from NHTSA also indicates that 91,000 crashes involved drowsy driving, leading to approximately 50,000 injuries and nearly 800 deaths.

Global authorities are aware of the serious problem of driver distraction. The World Health Organization (WHO) says that driver distraction is a major risk factor for road accidents. The United States Department of Transportation calls distracted driving one of the most dangerous driving behaviours, made worse by the increase in mobile phone use. This highlights the urgent need for a system that can detect when drivers are distracted and alert them in real-time to prevent accidents. Such a system could greatly reduce the number of accidents and deaths caused by distracted driving. Researchers have been working on this issue and have developed various deep learning models to detect driver distractions. As autonomous vehicles become more common, there is a need for a reliable model that can be used in real-time, like in driver assistance systems, to quickly alert drivers about distractions.

Common forms of distraction include using a mobile phone, interacting with in-car entertainment systems, and reaching for objects. Our driver monitoring system categorizes driver behaviours into 14 classes:

Distracted Behind, Distracted Glovebox, Distracted Hand off wheel, Distracted Leaning forward, Distracted Left, Distracted Phone, Distracted Radio, Distracted Right, Normal Driving, Drinking, Drowsy, Hands on Seatbelt, and Smoking.

Figure 2 2.2.1: Relevant Statistics Highlighting the Need for a Driver Monitoring System

By accurately detecting and classifying these behaviours, we aim to reduce the number of accidents caused by distracted driving.

2.2.2. Child Detection System

Another critical safety issue is the occurrence of children being left unattended in vehicles, leading to heatstroke fatalities. The National Safety Council reports that on average, 38 children die each year from heatstroke after being left in hot cars. Our child detection system addresses this problem by identifying the presence of a child or a teddy bear (as a proxy for child-like objects) in the vehicle, alerting the driver or guardians if a child is left behind.

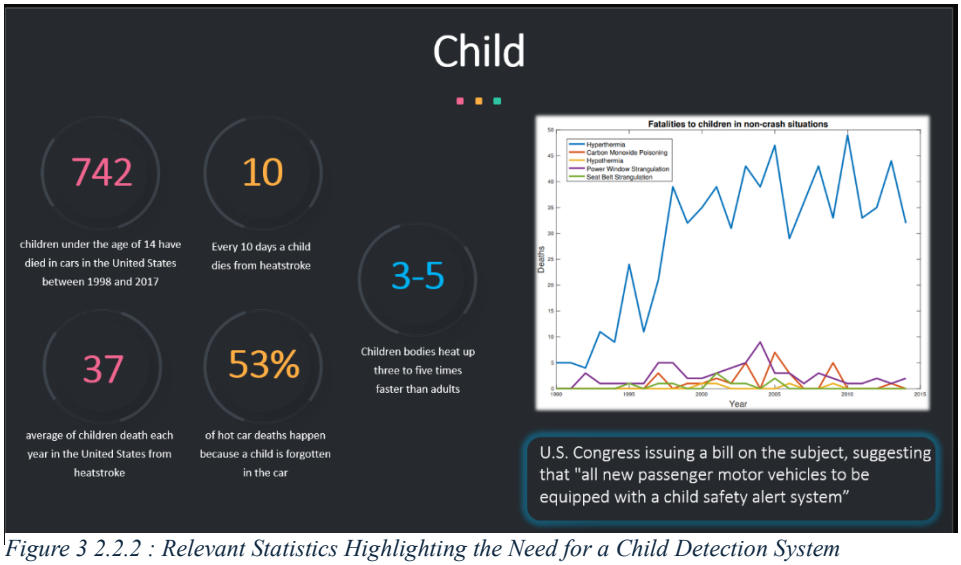


Figure 3 2.2.2 : Relevant Statistics Highlighting the Need for a Child Detection System

2.3. Project Objectives

1- Enhance Road Safety:

Develop a reliable driver monitoring system to detect and classify 14 distinct driver behaviours, thereby reducing the risk of accidents caused by distracted driving.

2- Prevent Child Heatstroke Fatalities:

Implement a child detection system to identify the presence of a child or child-like object in the vehicle, providing alerts to prevent children from being left unattended.

3- Improve Driver Awareness:

Provide real-time alerts and feedback to drivers about their behaviour, promoting safer driving habits.

4- Integrate Seamlessly with Existing Vehicle Systems:

Ensure the solution can be integrated with current in-vehicle technologies without significant modifications.

5- Adapt to Diverse Environments:

Design the system to function effectively across different regions and vehicle types, accommodating varying environmental conditions and user behaviours

2.4. Existing Systems

There are several existing models and solutions aimed at addressing distracted driving and child detection

2.4.1. Existing Driver Monitoring Systems

Existing Models and Their Limitations

1. **Facial Recognition Systems:**
   - These systems use cameras to monitor the driver's face for signs of drowsiness or distraction.
   - Limitation: They can be affected by lighting conditions, occlusions (e.g., sunglasses or hands), and variations in facial expressions.
2. **Eye-Tracking Systems:**
   - These systems track the driver's eye movements to detect signs of drowsiness or inattention.

- Limitation: They require precise calibration and can be disrupted by rapid head movements or obstructions such as glasses or contact lenses.

3. **Behavioral Monitoring Systems:**

- These systems analyze the driver's behavior, such as steering patterns and pedal usage, to identify irregularities that may indicate fatigue or distraction.
- Limitation: They may not accurately detect subtle signs of driver impairment and can produce false positives during unusual but safe driving conditions (e.g., avoiding obstacles).

4. **Heart Rate and Biometric Sensors:**

- These systems monitor the driver's physiological signals to detect stress, fatigue, or other issues.
- Limitation: They require wearable devices, which can be uncomfortable or intrusive for the driver, and may not provide real-time data due to sensor lag or connectivity issues.

2.4.2. Existing Child Detection Systems

1. Rear Seat Reminder (GM's Rear Seat Reminder)

Some automakers have added rear seat reminders to help prevent children from being left in the backseat. These systems activate when the car is turned off and the driver's door is opened, prompting drivers to check the backseat. General Motors introduced this feature in 2016, and Nissan followed in 2017, planning to include it in all four-door models by 2022.

2. Wi-Fi Sensing

Wi-Fi Sensing technology, developed by Origin, uses Wi-Fi waves in vehicles to detect movement and human presence. This technology can even detect motionless children by analyzing breathing and activity. It is expected to be included in new car models and work with in-car alarm systems and mobile apps to prevent children from being left behind.

3. Ultrasonic Sensors (Hyundai Rear Occupant Alert)

Ultrasonic sensors emit sound waves and analyze their reflections to detect objects or people, triggering alerts. However, their accuracy can be affected by vehicle size, temperature changes, and other interferences.

They may also fail to detect a motionless child, or one covered by a blanket.

4. In-Cabin Radar

Radar-based systems detect small movements like breathing and can distinguish between adults, children, and pets. Hyundai and Tesla are developing this technology for their vehicles. However, these systems can sometimes detect movement outside the car, causing false alarms.

2.5. Our System

2.5.1. Our Driver Monitoring System

Our system aims to overcome these limitations by leveraging YOLOv8 with deep learning to detect 14 different classes related to driver monitoring. This system uses a camera to provide real-time monitoring and alerting.

Advantages of Our Model

1. Comprehensive Detection: The system can identify 14 classes, including signs of drowsiness, distraction, phone usage, seatbelt usage, and more, offering a broader scope of monitoring compared to existing systems.
2. High Accuracy: Using YOLOv8 ensures precise detection of various driver behaviors and states, even in challenging conditions.
3. Real-Time Monitoring: The system processes camera feed in real-time, providing immediate alerts and feedback to the driver.
4. Non-Intrusive: Unlike biometric sensors, our system does not require any wearable devices, making it more comfortable and less intrusive for the driver.
5. Robust Performance: The model is trained to handle different lighting conditions, occlusions, and rapid movements, ensuring reliable performance across various scenarios.

Project Projection and Overcoming Limitations

Our driver monitoring system aims to enhance safety and reliability by addressing the limitations of existing models. By integrating YOLOv8 and deep learning, we provide a more accurate and comprehensive monitoring solution. The system's ability to

distinguish between multiple driver behaviors and conditions ensures that it can effectively detect and alert drivers to potential safety issues.

Key Features and Projections:

- Real-Time Detection: Immediate alerts for detected issues, improving driver response time.
- Enhanced Safety: By monitoring a wide range of behaviors, the system helps prevent accidents caused by distraction, drowsiness, and other impairments.
- Scalability: The system can be easily scaled and integrated into various vehicle models without requiring additional hardware, making it cost-effective and versatile.
- User-Friendly: The non-intrusive nature of the system ensures driver comfort and compliance.

By addressing the limitations of existing models and providing a robust, real-time monitoring solution, our driver monitoring system aims to significantly improve road safety and driver awareness.

2.5.2. Our Child Detection System

Our System uses YOLOv8 with deep learning to detect children and teddy bears through a camera on the dashboard. This approach aims to overcome the limitations of existing models by providing:

- **High Accuracy**: Advanced deep learning ensures precise detection of children and child-like objects.
- **Real-Time Alerts**: Immediate notifications to the driver or guardians if a child is left behind.
- **Robust Detection**: Effective in various conditions, including different vehicle sizes and minimal child movement.
- **Teddy Bear Distinction**: We train the model to recognize teddy bears to avoid false alarms, sending alerts only when a child is present.

- **Ease of Integration**: Easily incorporated into existing vehicle systems without the need for additional hardware like Wi-Fi or ultrasonic sensors.

This system enhances child safety in vehicles by offering a reliable, accurate, and easy-to-implement solution.

3. Chapter 3: System Design

3.1. Introduction

The system design and architecture form the backbone of the ADAS project, which aims to enhance driver monitoring and child safety within vehicles. This chapter delves into the structural blueprint of our project, highlighting how various components integrate and interact to achieve the overall objectives.

The primary focus of this chapter is on the interaction between the Electronic Control Units (ECUs), the block diagram of the system (excluding peripheral components), and the software design. By understanding the system design and architecture, we can appreciate the complexity and innovation involved in developing a robust ADAS solution.

The key objectives of this chapter are:

- To provide a comprehensive overview of the system's architecture through detailed block diagrams.
- To elucidate the interaction between the core components, specifically the ECUs and their communication protocols.
- To explain the integration process of these components and their roles within the overall system.
- To present the software design and the Software Development Life Cycle (SDLC) applied in developing the ADAS system.

In the subsequent sections, we will present high-level and detailed block diagrams to visualize the system architecture. We will then explore the intricate interactions between the ECUs, emphasizing how they communicate and work together to monitor the driver and ensure child safety. Additionally, we will discuss the software design and the SDLC, providing insight into the methodologies and processes that underpin the software development for the ADAS project. This foundational understanding is crucial for grasping the implementation details discussed in later chapters.

3.2. Block Diagram

3.2.1. High-Level Block Diagram

In this section, we present the high-level block diagram of the Advanced Driver Assistance System (ADAS) designed to enhance driver monitoring and child safety in vehicles. This diagram offers an overview of the primary components and their interactions, serving as a foundational reference for understanding the system's architecture and operational flow.

At the core of the system is the Raspberry Pi 4, which functions as the central processing unit. It runs sophisticated AI models for detecting both the driver and the child within the vehicle, processes data from multiple sensors, and manages communication with other system components.

Three STM32F103C6 microcontrollers are integrated into the system, each dedicated to specific critical functions: GSM communication, GPS data management, and monitoring of temperature and humidity sensors. However, one of the microcontrollers acts as a gateway, managing the communication between the Raspberry Pi and the other two microcontrollers. This gateway microcontroller receives commands from the Raspberry Pi and relays them to the other microcontrollers. It also collects data from these microcontrollers and sends it back to the Raspberry Pi. The communication between the Raspberry Pi and the microcontrollers is facilitated by the CAN bus protocol, ensuring reliable and real-time data transmission.

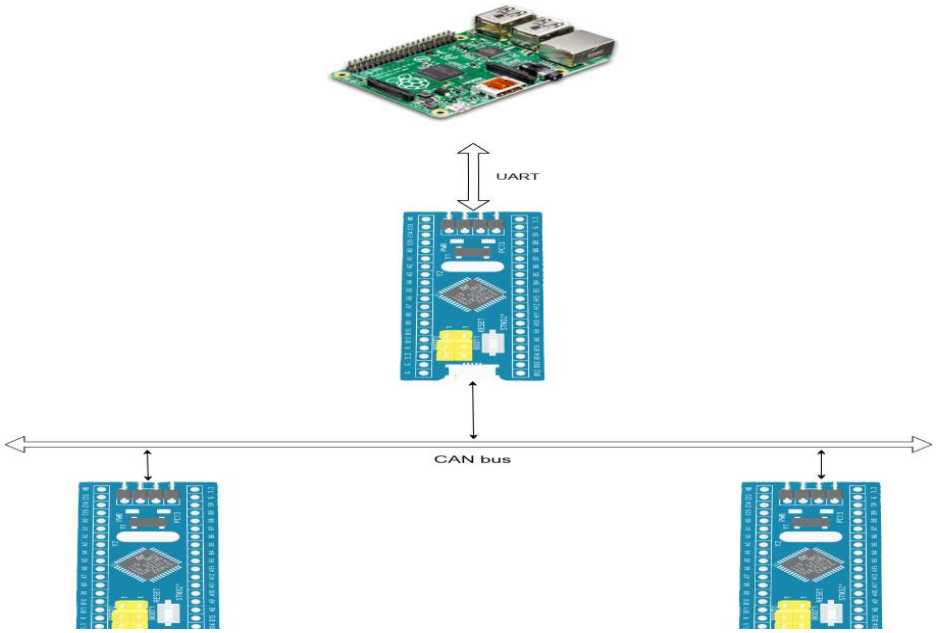


Figure 4 3.1: High-Level of ADAS

3.2.2. Detailed Block Diagram

In this section, we delve into the detailed block diagram of the ADAS, providing a comprehensive view of the system components and their specific roles. This detailed diagram expands upon the high-level overview by illustrating the finer aspects of the interactions and data flows between the core elements of the system.
The detailed block diagram (Figure5 3.2) captures the complexities of the ADAS, highlighting how each component is meticulously integrated to achieve the project's objectives. The diagram includes the central processing unit (Raspberry Pi 4), the three STM32F103C6 microcontrollers, and the CAN bus communication protocol, along with their specific interconnections and data exchanges.
At the heart of the system is the Raspberry Pi 4, which serves as the main processing unit. It runs advanced AI models designed for driver and child detection, processes data from various sensors, and handle communication with the microcontrollers. The AI models implemented on the Raspberry Pi utilize deep learning algorithms, such as Convolutional Neural Networks (CNNs), to accurately identify and monitor the presence of a driver and a child within the vehicle. The Raspberry Pi also manages the user interface, including a touch screen dashboard that displays real-time information and alerts.

The three microcontrollers are specialized for different tasks, ensuring that each critical function of the ADAS is handled efficiently. The STM32F103 microcontroller acts as a gateway between the Raspberry Pi and the other microcontrollers. It is specialized for GSM communication, enabling the system to send alerts and notifications via mobile networks. This feature is vital for emergency situations where immediate communication with external entities is required. The ECU1 receives commands from the Raspberry Pi, relays them to the ECU2 and ECU3 microcontrollers, and collects data from these microcontrollers via the CAN bus. It plays a crucial role in managing the flow of information within the system.

The ECU2 microcontroller is dedicated to managing GPS data for precise location tracking. It is connected to a GPS module that receives GPS signals and processes them to calculate the vehicle's exact location. The ECU1 gateway communicates with the ECU2 microcontroller via the CAN bus, enabling the transmission of location data back to the Raspberry Pi.

The ECU3 microcontroller monitors environmental conditions inside the vehicle using temperature and humidity sensors. These sensors provide real-time data on the internal climate, which is crucial for ensuring passenger comfort and safety. The ECU3 communicates with the ECU1 gateway via the CAN bus, allowing environmental data to be sent to the Raspberry Pi for processing and analysis.

The CAN bus protocol plays a crucial role in facilitating robust communication between the Raspberry Pi and the microcontrollers. This vehicle bus standard supports real-time data exchange, ensuring that all components remain synchronized and can respond swiftly to changing conditions. The detailed block diagram emphasizes the pathways and data flows that occur via the CAN bus, underscoring its importance in maintaining system integrity and performance.

Figure5 3.2 illustrates the detailed block diagram of the ADAS. In this diagram, the Raspberry Pi 4 is connected to various peripherals, including the camera and touch screen. It communicates with the three STM32F103C6 microcontrollers through the CAN bus. Each microcontroller is shown with its specific connections and data paths, highlighting its role within the system. The diagram also indicates the flow of data from sensors to the processing unit and the subsequent dissemination of information to the user interface and external communication modules.

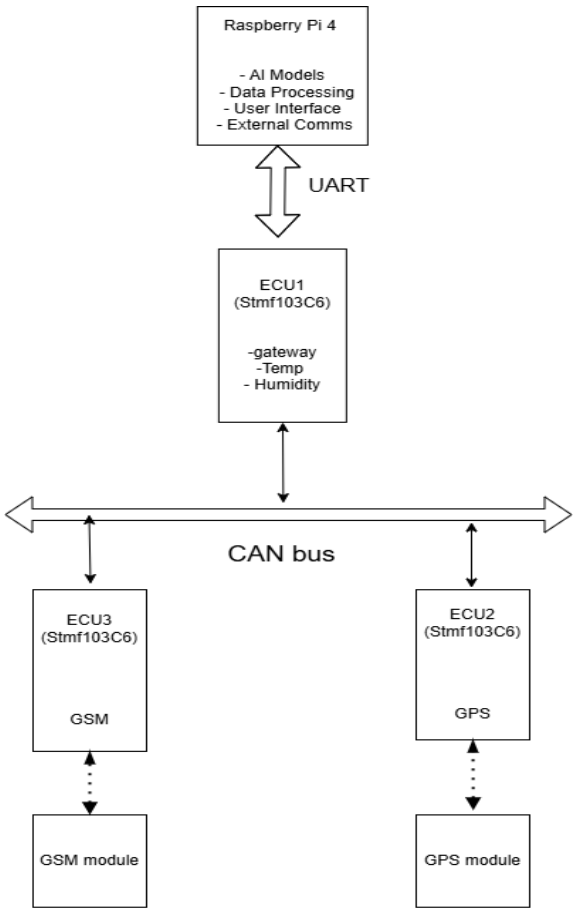


Figure 5 3.2 Detailed Block Diagram

Data Exchange and Synchronization

The CAN bus protocol facilitates real-time data exchange and synchronization between the Raspberry Pi 4 and the ECU1, ECU2, and ECU3 microcontrollers. It ensures that all ECUs within the ADAS system operate in harmony, sharing vital information and responding promptly to changing conditions. The data exchanged includes commands, sensor data, and status updates, enabling the system to function accurately and reliably.

3.3. Software Design and SDLC

3.3.1. Overview of Software Design

In this section, we provide an overview of the software design for the Advanced Driver Assistance System (ADAS). The software architecture is crucial for ensuring that the system operates smoothly, processes data efficiently, and delivers reliable performance in detecting drivers and ensuring child safety within vehicles

3.1.1.1. Software Architecture

The software architecture of the ADAS is designed to integrate various components, including the Raspberry Pi 4, STM32F103C6 microcontrollers.

Components of software Architecture:

1. **AI Models**: The AI models implemented on the Raspberry Pi 4 utilize Convolutional Neural Networks (CNNs) and other deep learning algorithms. These models are trained to identify and monitor the presence of a driver and a child inside the vehicle, ensuring accurate detection and timely alerts.
2. **Embedded Linux**: The Raspberry Pi 4 runs on an embedded Linux operating system, which provides a stable platform for running applications, managing system resources, and facilitating communication between software components.
3. **User Interface**: The touch screen dashboard connected to the Raspberry Pi 4 provides real-time information, alerts, and notifications to the vehicle's occupants. It allows users to interact with the ADAS system and view critical data related to driver monitoring and child safety.
4. **Communication Protocols**: The CAN bus protocol is used for communication between the Raspberry Pi 4 and the STM32F103C6 microcontrollers. It enables real-time data exchange and synchronization, ensuring that all components of the ADAS system operate in harmony.
5. **GSM and GPS Integration**: The GSM module in the STM32F103C6 microcontroller facilitates communication via mobile networks, allowing the ADAS system to send alerts and notifications in emergency situations. The GPS module provides precise location data, which is essential for navigation and geofencing applications.

3.3.2. Software Development Life Cycle (SDLC)

The software for the ADAS system is developed following a structured approach known as the Software Development Life Cycle (SDLC). This approach includes several phases:

Requirements Gathering: Gathering and analysing requirements for the ADAS system, including functionalities, performance criteria, and user expectations.

Design: Designing the software architecture, including the selection of AI models, development of user interfaces, and definition of communication protocols.

Implementation: Implementing the software components, including coding, integration of AI models, and development of the user interface on the Raspberry Pi 4.

Testing: Conducting unit testing, integration testing, and system testing to ensure that the software meets the specified requirements and operates correctly under various conditions.

Deployment: Deploying the software on the Raspberry Pi 4 and STM32F103C6 microcontrollers, configuring the system, and ensuring compatibility with the vehicle's environment.

3.3.2.1. Requirement Analysis

The SDLC begins with Requirement Analysis, where the needs of the end-users and stakeholders are identified and documented. For ADAS, this phase involves gathering detailed requirements related to driver monitoring, child safety, real-time processing, and integration with vehicle sensors and peripherals

3.3.2.1.1. Driver Monitoring Requirements

in the ADAS: The following tables outline the key requirements for driver monitoring

Table 1: Driver Monitoring Alerts and Notifications

Alert/Notification	Description
Drowsiness Detection	Detection of signs of drowsiness in the driver, such as eyelid closure or head nodding.

Distraction Alert	Alerting the driver when signs of distraction are detected, such as looking away from the road for an extended period.
Eyes-off-the-road Alert	Notifying the driver when their eyes are off the road for an unsafe duration.
Fatigue Warning	Warning the driver when fatigue levels are detected to be high, based on behavior patterns and physiological signals.

Table 2: Real-Time Processing Requirements

Requirement	Description
Response Time	The maximum allowable time for detecting and responding to driver alerts.
Processing Speed	The rate at which data is processed to ensure real-time monitoring.
Accuracy of Detection	The level of accuracy required for detecting drowsiness, distraction, etc.
Robustness to Environmental Factors	Ability to perform effectively under varying lighting and environmental conditions.

Table 3: User Interface Requirements

Requirement	Description
Alert Display	Displaying alerts and notifications prominently on the dashboard.
User Interaction	Allowing the driver to acknowledge or dismiss alerts through intuitive controls.
Visual Feedback	Providing visual feedback to the driver on their current state and behavior.
Auditory Feedback	Offering auditory alerts and warnings to supplement visual feedback.

3.3.2.1.2. Child Safety Requirements

The following tables outline the key requirements for child safety in the ADAS:

Table 1: Child Presence Detection Requirements

Requirement	Description
Child Presence Detection	Ability to detect the presence of a child inside the vehicle, including in blind spots.
Object Classification	Accurately classify the detected object as a child or other objects.
Real-time Alerts	Provide real-time alerts to the driver when a child is detected in or near the vehicle.

Table 2: Environmental Monitoring Requirements

Requirement	Description
Temperature Monitoring	Monitor the temperature inside the vehicle to ensure it is within safe limits.
Humidity Monitoring	Monitor humidity levels inside the vehicle to ensure comfort and safety for the child.
Alerts and Notifications	Provide alerts to the driver if the temperature or humidity levels become unsafe.

Table 3: Child Safety Seat Monitoring Requirements

Requirement	Description
Child Safety Seat Detection	Detect the presence of a child safety seat and ensure it is properly secured.
Seat Belt Monitoring	Monitor the status of the child safety seat belt to ensure it is correctly fastened.
Alerts and Warnings	Provide alerts to the driver if the child safety seat is not properly secured or if the seat belt is unfastened.

3.3.2.1.3. Integration Requirements

The following table outlines the key integration requirements for the ADAS:

Table 1: Integration Requirements

Requirement	Description
Sensor Integration	Integrate sensor data from various sources, including cameras, temperature, humidity, and GPS sensors.
Microcontroller Integration	Ensure seamless communication and data exchange between the Raspberry Pi and STM32F103C6 microcontrollers using the CAN bus.
AI Model Integration	Integrate AI models for driver and child detection with the main processing unit (Raspberry Pi 4).
User Interface Integration	Integrate the user interface with the main processing unit to display real-time information and alerts.
External Communication Integration	Integrate with external systems for communication and emergency alerts using GSM and GPS modules.

3.3.2.2. Design

Following Requirement Analysis, the System Design phase outlines the architecture and components of the ADAS software. This phase includes designing:

System Architecture: The architecture is designed to support the integration of AI models for driver and child detection, real-time data processing, and user interface management. It specifies the roles and responsibilities of each component, ensuring they work together seamlessly.

Hardware and Software Components: This involves selecting and specifying hardware components (Raspberry Pi 4, STM32F103C6 microcontrollers) and software frameworks (TensorFlow, PyTorch) for AI model development.

Data Flow Diagrams: Diagrams illustrate the flow of data within the system, from sensors to the processing unit and user interface. This includes data preprocessing, AI model inference, and alert generation.

Interaction Between Modules: Detailed interactions between the Raspberry Pi 4, microcontrollers, and peripherals (cameras, touch screens) are designed to ensure efficient communication and data exchange via the CAN bus protocol.

3.3.2.3. Implementation

The Implementation phase involves translating the system design into actual code. In ADAS, this includes:

AI Model Development: Developing AI models using TensorFlow or PyTorch for driver and child detection. This includes training the models on labelled datasets and optimizing them for real-time performance on the Raspberry Pi 4.

Programming Microcontrollers: Writing firmware for the STM32F103C6 microcontrollers to manage GSM communication, GPS data processing, and environmental sensing. This includes implementing communication protocols (CAN bus) and integrating sensors.

User Interface Development: Creating a touch screen dashboard on the Raspberry Pi 4 to display real-time information, alerts, and system status. This involves designing a user-friendly interface for drivers and caregivers.

Integration Testing: Conducting integration testing to verify the correct interaction between hardware components (Raspberry Pi 4, microcontrollers) and software modules (AI models, sensor drivers). This ensures that all components work together as intended.

3.3.2.4. Testing

Testing is a critical phase in the SDLC, ensuring that the ADAS software meets its specified requirements and functions reliably in various scenarios. The testing phase includes:

Unit Testing: Testing individual components such as AI models, microcontroller functionalities, and sensor integrations to verify their correctness and performance.

Integration Testing: Testing the integration of different software modules and hardware components to ensure they work together seamlessly.

System Testing: Testing the entire ADAS system to validate its functionality, performance, and reliability in real-world conditions. This includes stress testing, edge case testing, and simulated driving scenarios.

3.3.2.5. Deployment

Deployment involves installing the ADAS software on the target hardware (vehicles) and integrating it into the vehicle's existing systems. This phase includes:

System Configuration: Configuring the ADAS system settings, including communication protocols (CAN bus), sensor calibration, and initial setup.

Acceptance Testing: Conducting acceptance testing to ensure that the ADAS software meets end-user requirements and specifications.

3.3.2.6. Maintenance

The Maintenance phase involves monitoring, updating, and maintaining the ADAS software throughout its lifecycle. This includes:

Bug Fixes and Updates: Addressing any bugs or issues identified during testing or deployment. This ensures the continued reliability and performance of the ADAS system.

Performance Optimization: Optimizing the performance of AI models, microcontroller firmware, and user interface to improve responsiveness and efficiency.

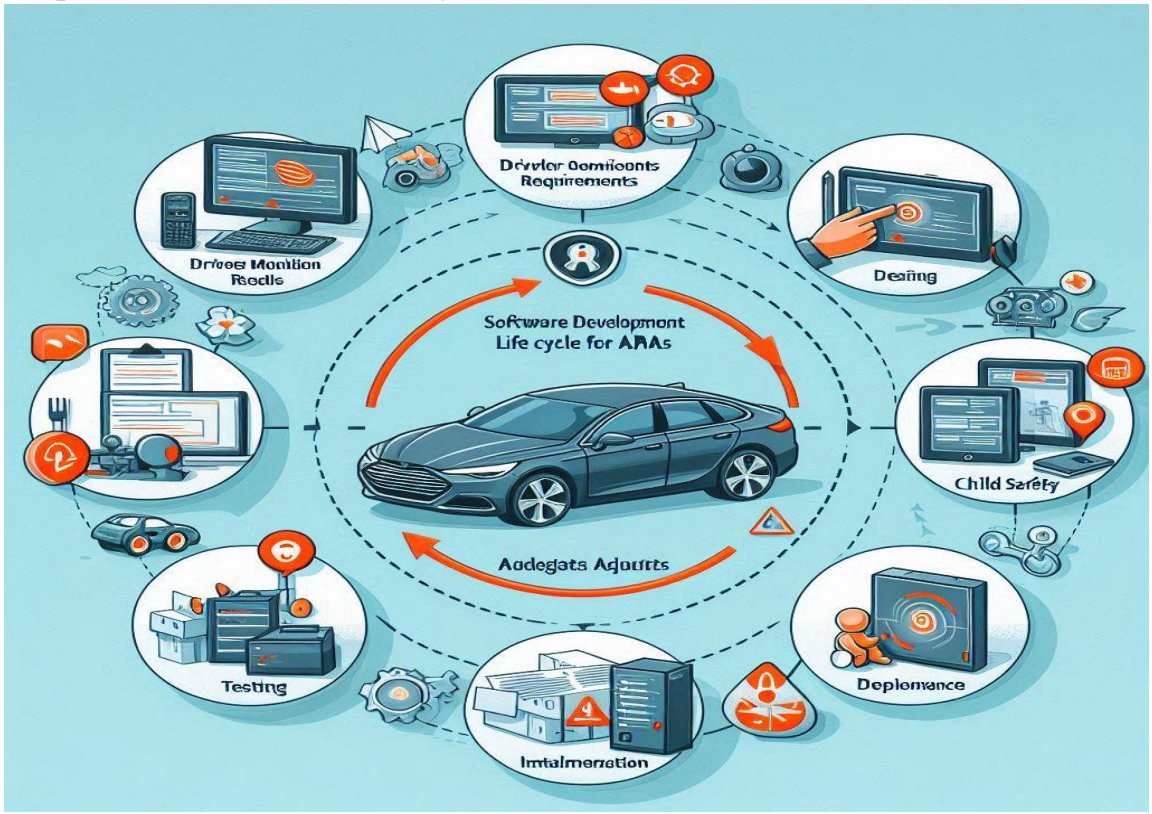


Figure 6 3.3 Software Development Life Cycle (SDLC) for ADAS

4. Chapter 4: Implementation

In this chapter, we dive into how we actually put our ideas into action for the driver monitoring system. Here, we explain step-by-step how we integrated advanced technology to make driving safer

4.1. AI Implementation

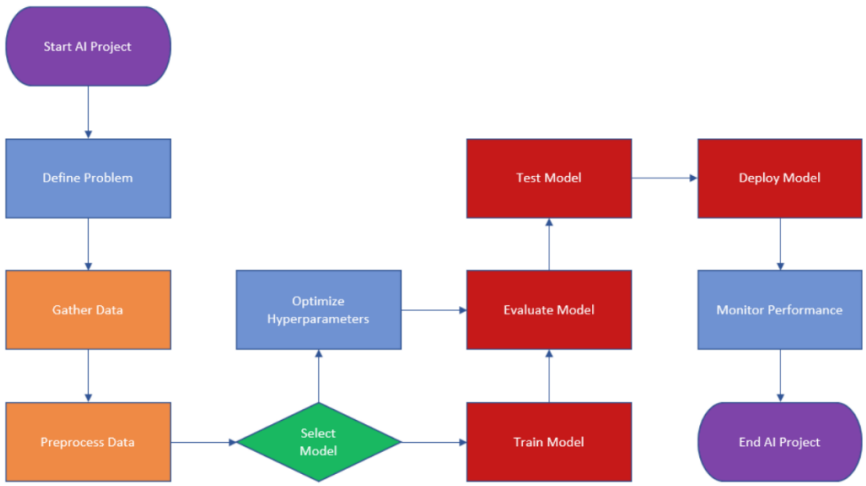


Figure 7 4.1 Computer vision pipeline for AI implementation

In this section, we lay out the steps we took to build our AI-powered driver monitoring system in a visual flowchart. Think of it like a roadmap that shows how everything connects together.

The flowchart breaks down the process as follows:

- **Start AI Project**: Initiating the project with a clear plan and setting up the necessary infrastructure.
- **Define the Problem**: Clearly articulating the specific problem we aim to solve with our AI system.
- **Gathering Data**: Collecting the required images using Roboflow datasets. This step involves sourcing and curating a diverse set of images to train the AI model.
- **Preprocessing**: Preparing the collected data to enhance and organize the dataset, making it suitable for training.

- **Select Model**: Choosing the most appropriate AI model that aligns with our project goals and requirements.
- **Train Model**: Training the selected model using the prepared dataset to learn and recognize patterns relevant to our problem.
- **Optimize Hyperparameters**: Fine-tuning the model's hyperparameters to improve its performance and accuracy.
- **Evaluate Model**: Assessing the trained model's performance using validation datasets to ensure it meets the desired criteria.
- **Test Model**: Conducting thorough testing to verify the model's reliability and accuracy in real-world scenarios.
- **Deploy Model**: Integrating the trained and tested model into real-life applications, ready for practical use.
- **Monitor Performance**: Continuously monitoring the model's performance to ensure it remains effective and making necessary adjustments as needed.
- **End AI Project**: Concluding the project with a comprehensive review and documentation of the outcomes and insights gained.

This flowchart is like a behind-the-scenes look at how we've brought our driver monitoring system to life using AI technology. It's a visual guide that helps us explain the careful planning and effort that went into making our project a reality.

4.1.2. Data Collection

Importance of Data

The quality and comprehensiveness of the dataset are crucial for training effective AI models. High-quality data ensures that the model can learn to recognize patterns accurately and perform well in real-world scenarios. In the context of our project, which involves both driver monitoring and child detection, having a diverse and representative dataset is particularly important.

This diversity includes various poses, lighting conditions, and environmental factors that the model might encounter in actual use.

Diverse data helps the model generalize better, reducing the risk of overfitting to specific conditions. For instance, in driver monitoring, capturing different angles and lighting conditions can help the model accurately identify behaviors like drowsiness or distracted driving, regardless of the time of day or the driver's position. Similarly, in child detection, including various poses and conditions ensures that the model can reliably distinguish between children and other objects, such as teddy bears, enhancing its accuracy and reliability.

4.1.2.1. Driver Monitoring Dataset

We collected a comprehensive set of images for driver monitoring using the [Dataset Name and Link Here].

Dataset Name: Driver Monitoring

Link to Dataset Driver Mentoring > Browse (roboflow.com)

Number of Images: 10,000 images .

Data Composition

The dataset includes images depicting 14 different driver behaviors

Some samples of the dataset classes :

1 – Distracted Behind

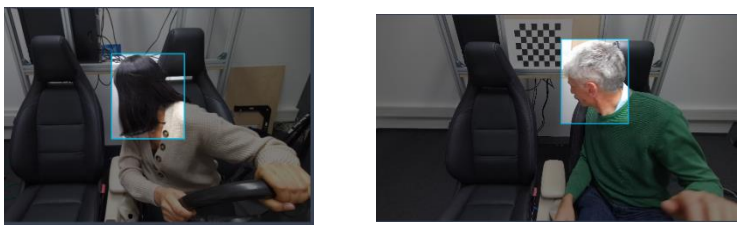
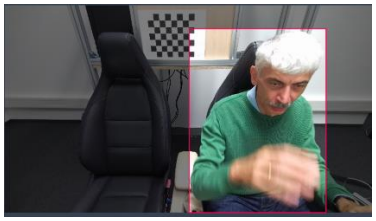
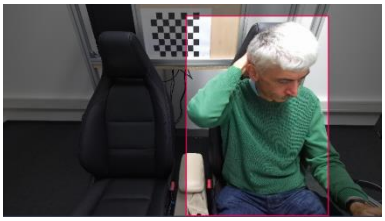


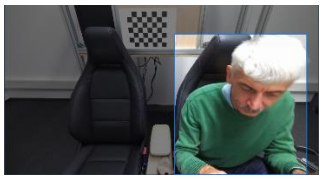
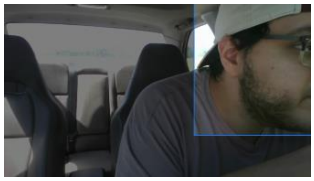
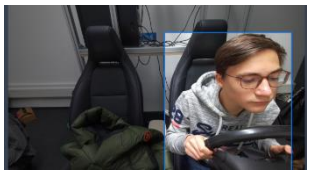
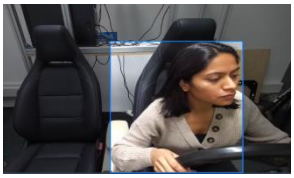
Figure 8 4.1.2.1. Driver Monitoring Data Set

2- Distracted Glovebox

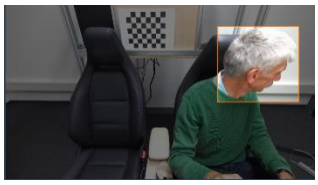
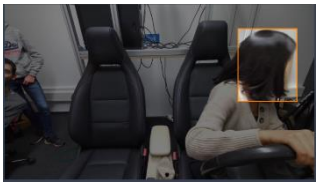
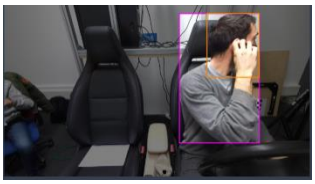
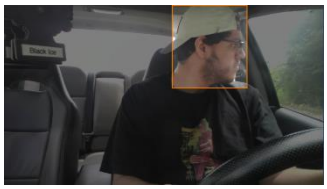
3- Distracted Hand off wheel

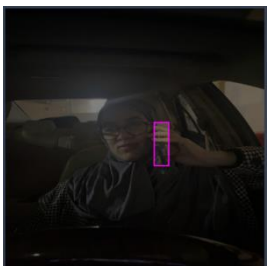
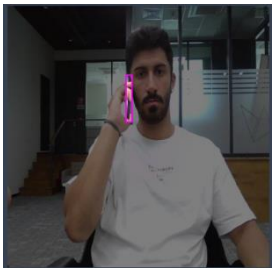
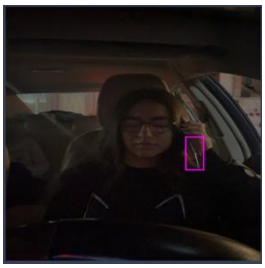
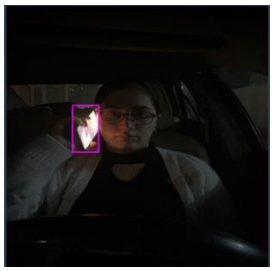
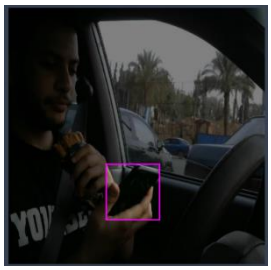
4- Distracted Leaning Forward

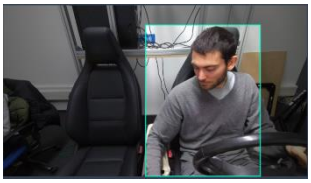
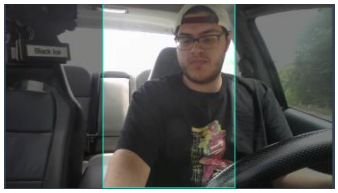
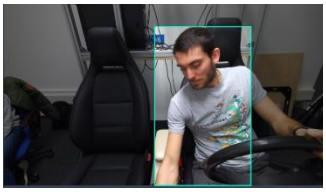
5- Distracted Left

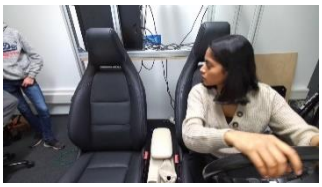
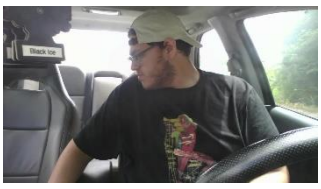
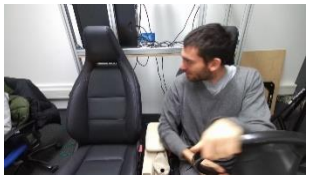
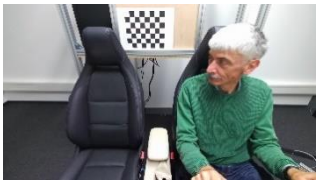
6 – Distracted Phone

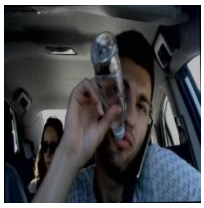
7 - Distracted Radio

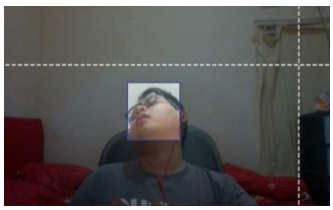
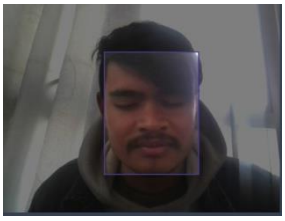

8- Distracted Right

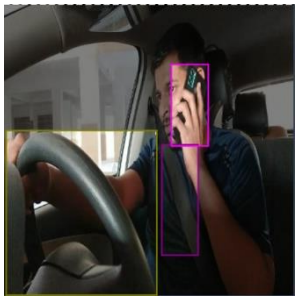
9- Drinking



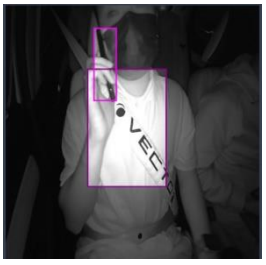
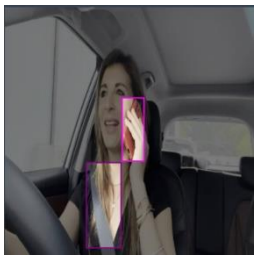

10 – Drowsy

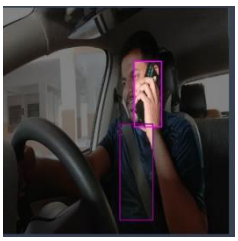





11 – Hands on

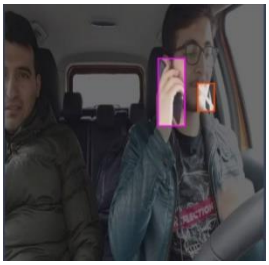
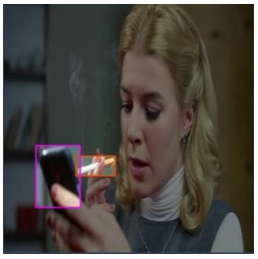
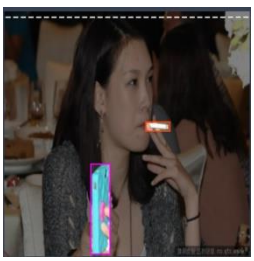



12 – Seat Belt

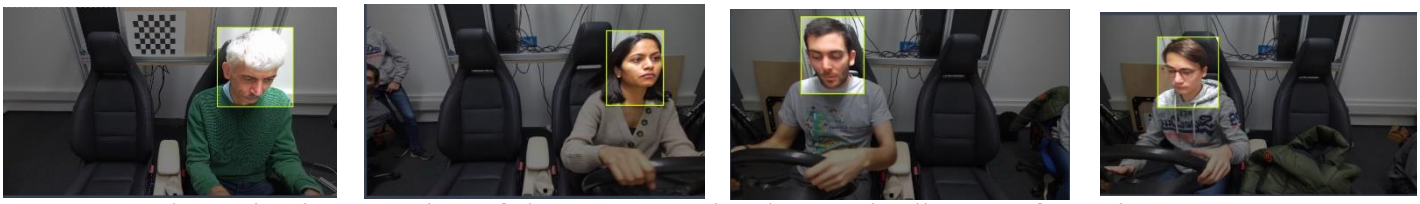





13 – Smoking

14 – Normal Driving



As seen above in the samples of the Dataset, the dataset is diverse, featuring a variety of faces, poses, and lighting conditions. This diversity is crucial for capturing all the necessary details, ensuring that the model becomes robust and capable of performing accurately in real-world scenarios. The inclusion of different people and varying environmental factors helps the model generalize better, enhancing its reliability and effectiveness in detecting driver behaviors and child presence under various conditions.

Data Preprocessing :

Before training, the collected data undergoes several preprocessing steps to enhance its quality and suitability for the model:

Auto-Orient:

Ensures that all images have a consistent orientation.

Static Crop:

Crops the images to 20-85% of the horizontal region and 5-85% of the vertical region to focus on the relevant parts of the image.

Resize: Stretches the images to a standard size of 640x640 pixels, ensuring uniformity across the dataset.

Grayscale: Converts the images to grayscale to simplify the data and focus on the essential features.

Figure 9 4.1.2.1. Driver Monitoring Data Processing

These preprocessing steps help in standardizing the dataset, reducing variability, and enhancing the model's ability to learn from the data effectively.

Data Augmentation

To enhance the diversity and robustness of the dataset, we applied several augmentation techniques:

Flip Horizontal: Flips the images horizontally to simulate different viewing perspectives.

Rotation: Randomly rotates images between -20° and +20° to introduce variability in orientation.

Shear: Applies horizontal and vertical shear transformations of up to ±10° to adjust the image perspective.

Saturation: Adjusts the saturation of the images between -33% and +33% to simulate varying lighting conditions.

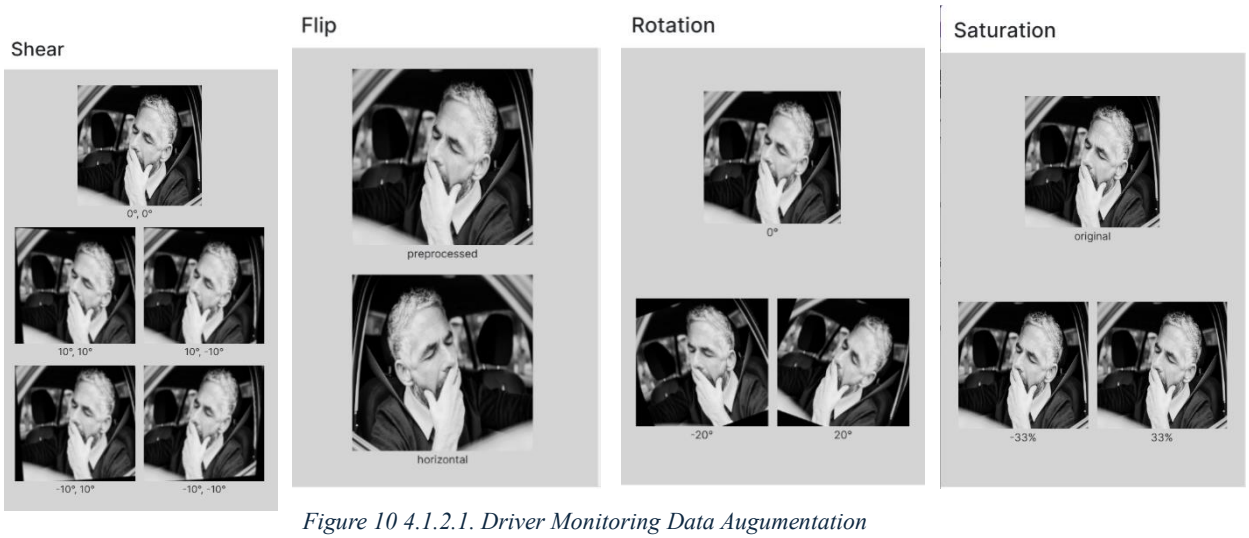


Figure 10 4.1.2.1. Driver Monitoring Data Augumentation

These augmentation techniques help in creating a more extensive and varied dataset, which improves the model's ability to generalize and perform well across different conditions and scenarios.

4.1.2.2. Child Detection Dataset

Dataset Collection We collected a comprehensive set of images for child detection .

Dataset Name: Child Detection

Link to Dataset: child detection > Browse (roboflow.com)

Number of Images: 2645 before augmentation

Data Composition The dataset includes images depicting two classes : Child and Teddy Bear

Some samples of the dataset classes:

Child

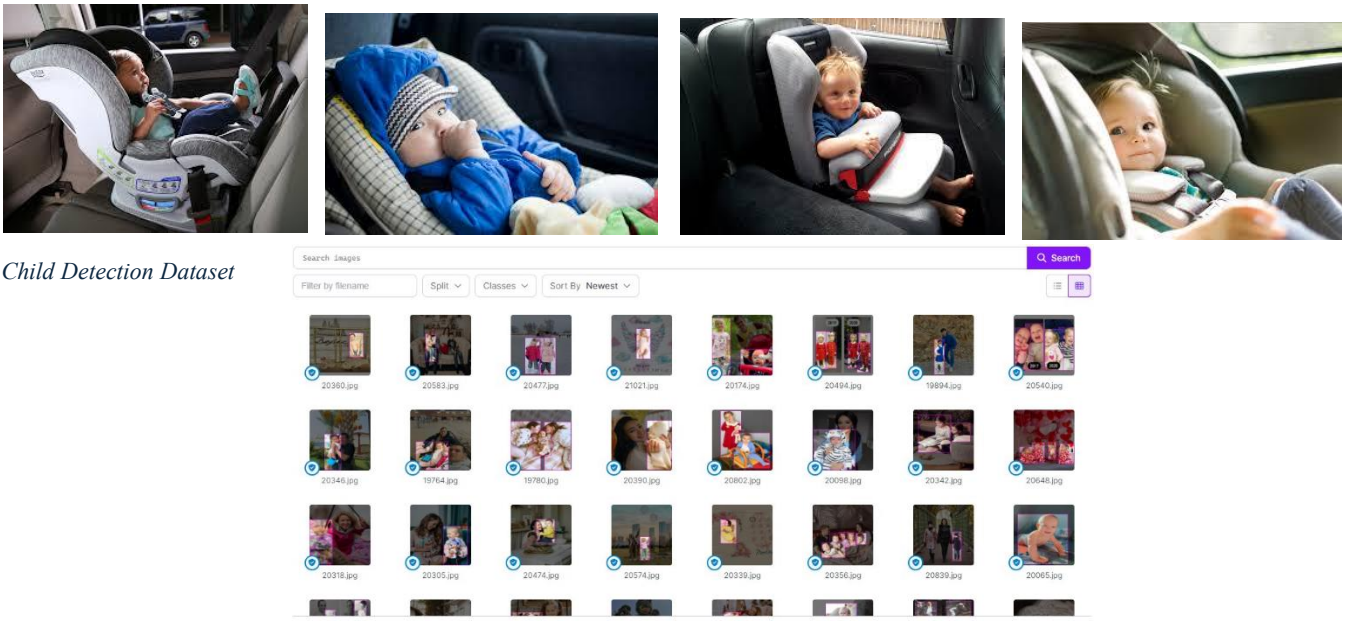


Figure 11 4.1.2.2. Child Detection Dataset

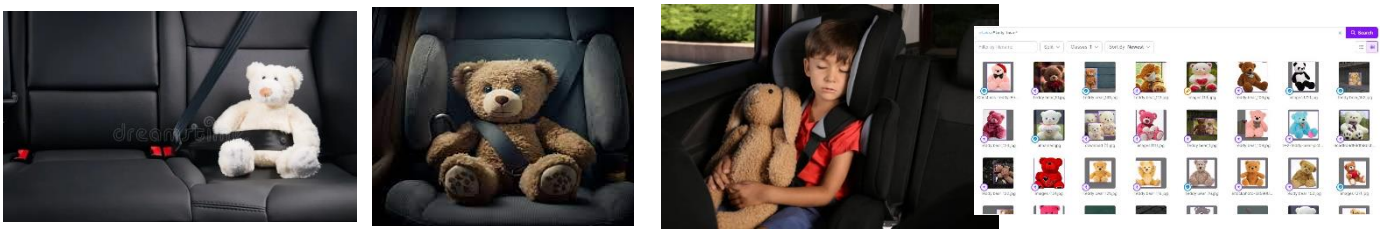


Figure 12 4.1.2.2. Child Detection Teddy Bear Dataset

Teddy Bear

As seen in the samples, the dataset is diverse, featuring various poses, environments, and lighting conditions. This diversity is crucial for capturing all the necessary details, ensuring that the model becomes robust and capable of performing accurately in real-world scenarios. The inclusion of different subjects and varying environmental factors helps the model generalize better, enhancing its reliability and effectiveness in detecting children under various conditions.

Dataset Split The dataset was split into training, validation, and test sets as follows:

Training Set: 7,053 images (96%)

Validation Set: 263 images (4%)

Test Set: 31 images (0%)

Preprocessing and Augmentation The dataset underwent preprocessing and augmentation to improve the model's robustness and performance. The key steps included:

Preprocessing:

Auto-Orient: Applied

Resize: Stretch to 640x640

Auto-Adjust Contrast: Using Contrast Stretching

Augmentation:

Outputs per training example: 3

Flip: Horizontal

Crop: 0% Minimum Zoom, 14% Maximum Zoom

Rotation: Between -15° and +15°

Brightness: Between -15% and +15%

This comprehensive preprocessing and augmentation strategy ensures that the model is well-prepared to handle various real-world scenarios and conditions, enhancing its generalizability and effectiveness in detecting children and distinguishing them from other objects like teddy bears.Fig 4.1.2.2 showing data split , augmentation and preprocessing steps for the child detection model .

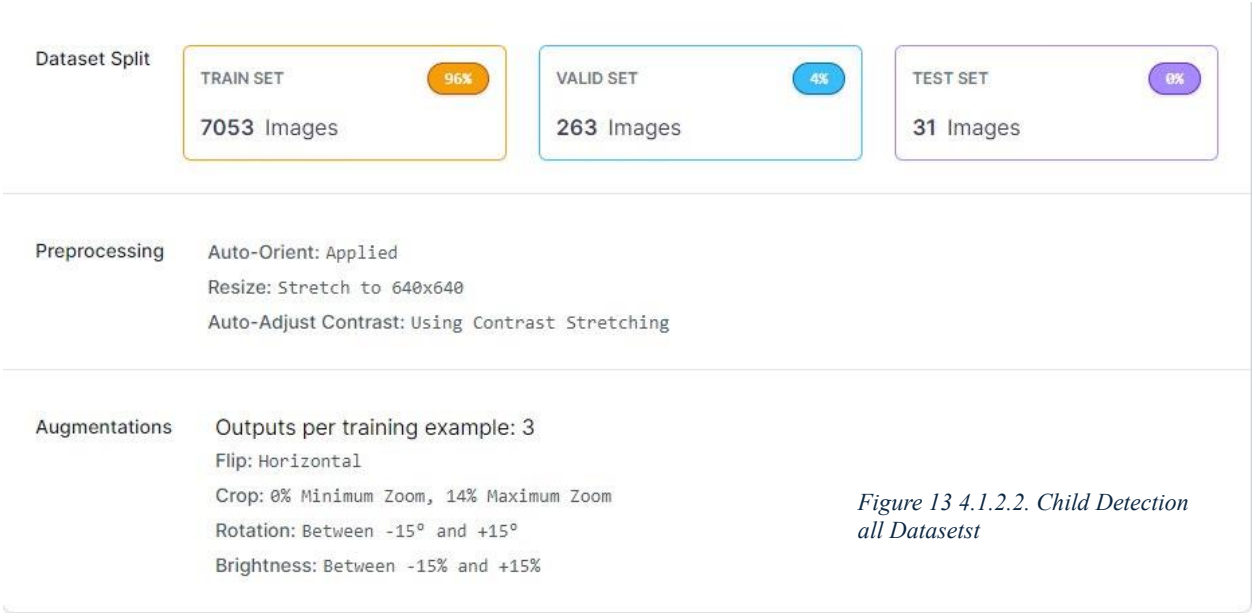


Figure 13 4.1.2.2. Child Detection all Datasetst

4.1.3. Model Selection

Following the dataset overview, we now delve into the rationale behind selecting the AI model for our computer vision system. Choosing the right model is crucial for ensuring the accuracy and effectiveness of our driver monitoring and child detection system.

4.1.3.1. Model Selection Criteria

We considered several factors while selecting the AI model:

Accuracy: The model's ability to correctly identify and classify objects, particularly in real-time scenarios.

Speed: The model's processing speed, which is essential for providing immediate feedback and alerts.

Flexibility: The model's adaptability to different environments and conditions within the vehicle.

Resource Efficiency: The model's ability to operate effectively with the computational resources available in an embedded system

4.1.3.2. Chosen Model: YOLOv10

YOLOv10: A New Era in Real-Time Object Detection

YOLOv10, released in May 2024, is set to redefine the landscape of real-time object detection. Building on the innovations introduced in YOLOv9, this model addresses critical challenges such as reliance on non-maximum suppression (NMS) and computational redundancy, which have previously hampered the efficiency and performance of YOLO models. YOLOv10 introduces significant improvements in speed, accuracy, and efficiency, making it a game-changer for applications requiring real-time insights. In this document, we will delve into the advancements of YOLOv10, explore its architecture, and discuss its performance compared to previous versions.

YOLOv10 is a cutting-edge computer vision architecture designed for real-time object detection, building upon the advancements of its predecessors. The YOLOv10 model achieves a higher mean Average Precision (mAP) compared to earlier YOLO models such as YOLOv9, YOLOv8, and YOLOv7 when benchmarked against the MS COCO dataset

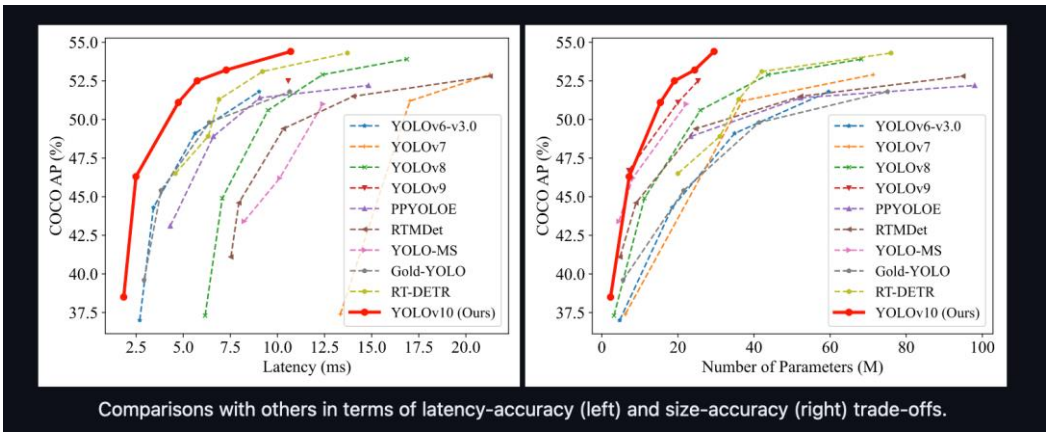


Figure 14 4.1.3.2. YOLOv10 performance

Key Innovations in YOLOv10

YOLOv10 introduces several key innovations aimed at overcoming the limitations of previous YOLO versions:

NMS-Free Training Strategy: Employing a consistent dual assignments strategy, YOLOv10 eliminates the need for Non-Maximum Suppression (NMS) during training, significantly reducing inference latency while maintaining competitive performance.

Holistic Efficiency-Accuracy Driven Design: Comprehensive optimization of model components enhances both efficiency and accuracy, minimizing computational redundancy.

Lightweight Classification Head: Designed to reduce computational redundancy without significantly impacting performance.

Spatial-Channel Decoupled Downsampling: Reduces computational cost and parameter count while retaining more information.

Rank-Guided Block Design: Utilizes intrinsic rank analysis to identify and address redundancy, optimizing the architecture for higher efficiency.

Large-Kernel Convolution: Enhances the model's capability to capture detailed features.

Effective Partial Self-Attention Module: Boosts accuracy with minimal computational cost.

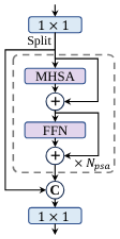


Figure 15 4.1.3.2. The partial self-attention module (PSA)

These enhancements allow YOLOv10 to achieve remarkable performance while maintaining high efficiency.

4.1.3.2.1. How YOLOv10 Works?

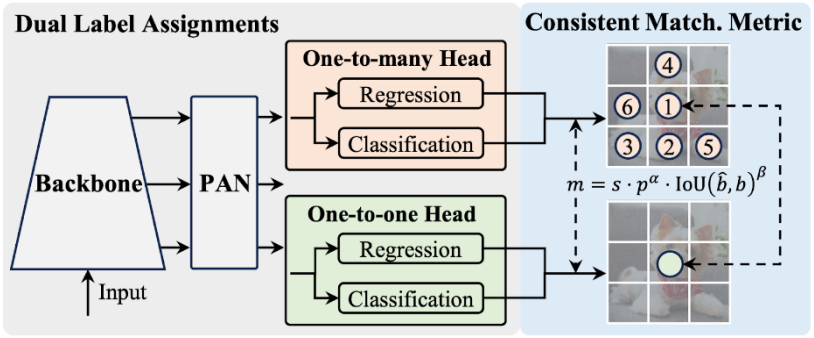
YOLOv10 introduces significant advancements in object detection through a novel training strategy and architectural enhancements:

Consistent Matching Metric: Harmonizes supervision across the dual heads, enhancing the quality of predictions during inference.

Classification Head, Decoupled Down sampling, Rank-Guided Block Design: Optimizes various components of the model to enhance efficiency and performance.

Kernel Convolution and Self-Attention: Incorporates large-kernel convolution and partial self-attention mechanisms to enhance accuracy with minimal computational cost

Figure 16 4.1.3.2. 1. Consistent dual assignments for NMS-free training



YOLOv10

Performance Comparison

YOLOv10 achieves state-of-the-art performance and end-to-end latency across various model scales. Compared to previous YOLO models, YOLOv10 demonstrates significant improvements in Average Precision (AP), parameter efficiency, and inference speed.

Lightweight and Small Models (YOLOv10-N / S): Achieve improvements of 1.5 AP and 2.0 AP with 51% / 61% fewer parameters and 41% / 52% less computation, respectively.

Medium-Sized Models (YOLOv10-B / M): Enjoy 46% / 62% lower latency compared to YOLOv9-C / YOLO-MS, with similar or better performance.

Large Models (YOLOv10-L): Outperform Gold-YOLO-L with 68% fewer parameters, 32% lower latency, and a 1.4% AP improvement.

Comparison with RT-DETR: YOLOv10-S / X achieves 1.8× and 1.3× faster inference speeds under similar performance.

Figure 17 4.1.3.2.1. Comparisons of the real-time object detectors on MS COCO dataset in terms of latency-accuracy (left) and size-accuracy (right) trade-offs, measured using the official pre-trained models for end-to-end latency

Model	#Param.(M)	FLOPs(G)	AP^{val}(%)	Latency(ms)	$Latency^{f}$(ms)
YOLOv6-3.0-N [27]	4.7	11.4	37.0	2.69	1.76
Gold-YOLO-N [54]	5.6	12.1	39.6	2.92	1.82
YOLOv8-N [20]	3.2	8.7	37.3	6.16	1.77
YOLOv10-N (Ours)	**2.3**	**6.7**	**38.5 / 39.5**†	**1.84**	**1.79**
YOLOv6-3.0-S [27]	18.5	45.3	44.3	3.42	2.35
Gold-YOLO-S [54]	21.5	46.0	45.4	3.82	2.73
YOLO-MS-XS [7]	4.5	17.4	43.4	8.23	2.80
YOLO-MS-S [7]	8.1	31.2	46.2	10.12	4.83
YOLOv8-S [20]	11.2	28.6	44.9	7.07	2.33
YOLOv9-S [59]	7.1	26.4	46.7	-	-
RT-DETR-R18 [71]	20.0	60.0	46.5	4.58	4.49
YOLOv10-S (Ours)	**7.2**	**21.6**	**46.3 / 46.8**†	**2.49**	**2.39**
YOLOv6-3.0-M [27]	34.9	85.8	49.1	5.63	4.56
Gold-YOLO-M [54]	41.3	87.5	49.8	6.38	5.45
YOLO-MS [7]	22.2	80.2	51.0	12.41	7.30
YOLOv8-M [20]	25.9	78.9	50.6	9.50	5.09
YOLOv9-M [59]	20.0	76.3	51.1	-	-
RT-DETR-R34 [71]	31.0	92.0	48.9	6.32	6.21
RT-DETR-R50m [71]	36.0	100.0	51.3	6.90	6.84
YOLOv10-M (Ours)	**15.4**	**59.1**	**51.1 / 51.3**†	**4.74**	**4.63**
YOLOv6-3.0-L [27]	59.6	150.7	51.8	9.02	7.90
Gold-YOLO-L [54]	75.1	151.7	51.8	10.65	9.78
YOLOv9-C [59]	25.3	102.1	52.5	10.57	6.13
YOLOv10-B (Ours)	**19.1**	**92.0**	**52.5 / 52.7**†	**5.74**	**5.67**
YOLOv8-L [20]	43.7	165.2	52.9	12.39	8.06
RT-DETR-R50 [71]	42.0	136.0	53.1	9.20	9.07
YOLOv10-L (Ours)	**24.4**	**120.3**	**53.2 / 53.4**†	**7.28**	**7.21**
YOLOv8-X [20]	68.2	257.8	53.9	16.86	12.83
RT-DETR-R101 [71]	76.0	259.0	54.3	13.71	13.58
YOLOv10-X (Ours)	**29.5**	**160.4**	**54.4 / 54.4**†	**10.70**	**10.60**

.

4.1.3.2.2. Application in Our Project

The YOLOv10 model helps us achieve the primary goals of our project:

Driver Monitoring: By classifying 14 different driver behaviors, our model can detect distractions, drowsiness, and other critical behaviors that impact road safety.

Child Detection: The model distinguishes between children and teddy bears, ensuring that alerts are only triggered when a child is detected, reducing false alarms and enhancing reliability.

In summary, our choice of the YOLOv10 model is driven by its high accuracy, real-time processing capabilities, robustness, and efficiency. This model meets our specific needs and contributes significantly to the effectiveness of our driver monitoring and child detection system.

4.1.4. Training Process

4.1.4.1. Dataset Split

To ensure robust training and evaluation, we split our dataset into three subsets: training, validation, and testing. This split helps in effectively

training the model, validating its performance during training, and evaluating its final performance on unseen data.

Train Set: 93% of the total dataset was allocated for training the model. This set consisted of 36,170 images, which included both original and augmented images. The augmentation techniques applied included flipping, rotation, zooming, and brightness adjustments to improve the model's generalization.

Validation Set: 5% of the dataset was used for validation, comprising 1,842 pure images. This set was used to tune the model parameters and prevent overfitting during the training process by providing a checkpoint to evaluate the model's performance on unseen data.

Test Set: The remaining 2% of the dataset, totaling 924 pure images, was reserved for the final evaluation of the model. This test set was used to assess the model's accuracy, precision, recall, and other performance metrics on completely unseen data.

Figure 18 4.1.4.1. Dataset Split

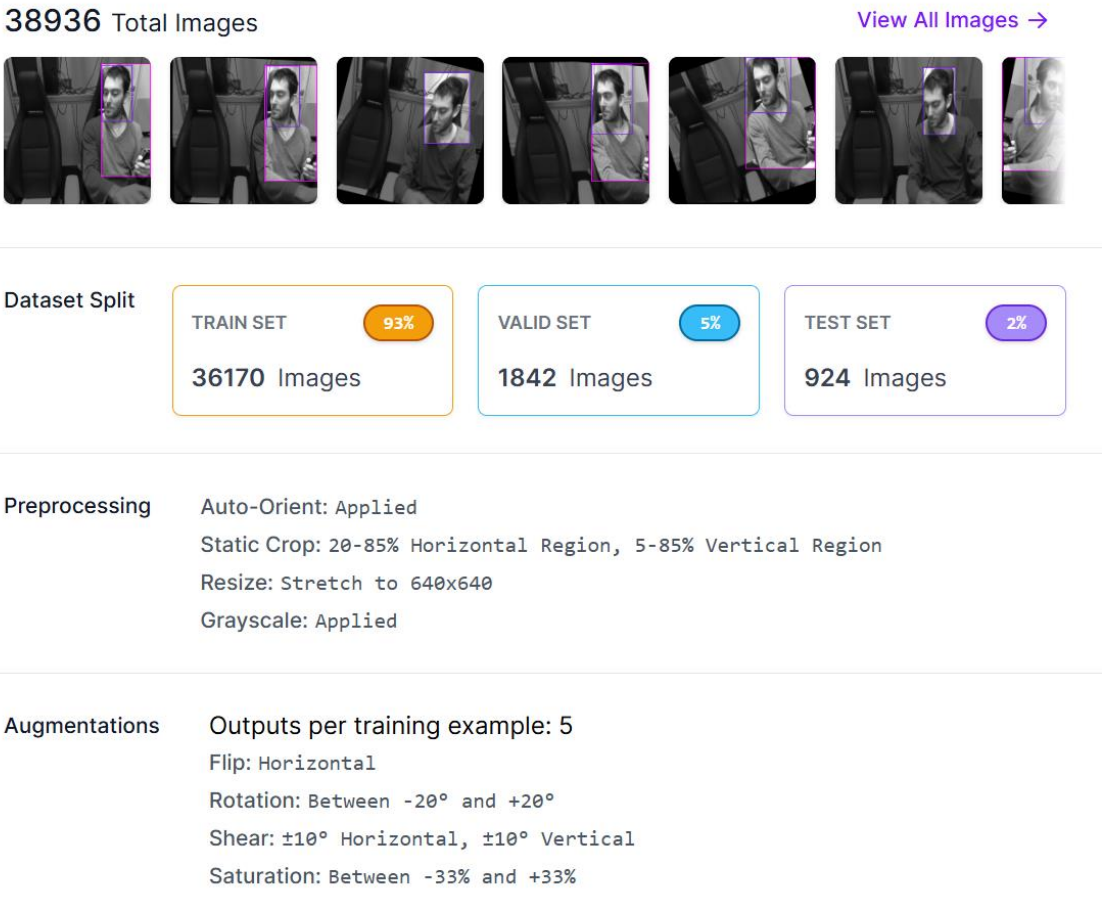


Fig 4.6.5 shows dataset split , preprocessing and Augmentation steps .

Model Selection and Training with YOLOv8 on Kaggle After the preprocessing and augmentation steps on Roboflow, we chose to train the YOLOv8 model on Kaggle due to its powerful computational resources and ease of integration with our workflow. Here are the main steps:

Choosing the Architecture We selected YOLOv8 for its state-of-the-art performance in object detection tasks. YOLOv8's architecture is well-suited for real-time applications, making it ideal for our driver monitoring and child detection system.

Configuring the Training Pipeline We configured the training pipeline on Kaggle, setting parameters such as learning rate, batch size, and number of epochs. Using Kaggle's robust GPU resources, we were able to efficiently train the YOLOv10 model.

Training the Model The model was trained on our dataset using Kaggle's computational power. The training process involved multiple iterations to fine-tune the model and achieve the best performance.

4.2. Embedded Linux Implementation

4.2.1. What is Embedded Linux and Why

Embedded Linux is a specialized subset of the Linux operating system designed to run on embedded systems. In our project, we utilize Embedded Linux on a Raspberry Pi to harness its robust features, extensive hardware support, and flexible development environment. The choice of Embedded Linux is driven by several critical factors that align perfectly with the requirements of our project.

Firstly, Embedded Linux provides a versatile and stable platform for interfacing with various hardware components, such as our ECU (Electronic Control Unit) which works like a gateway and the Raspberry Pi's touch screen. The ECU (gateway) communicates with the Raspberry Pi through the UART (Universal Asynchronous Receiver-Transmitter) protocol, a serial communication method that is well-supported by Linux. The reliability and efficiency of Embedded Linux in handling UART

communication ensures seamless data transmission from the ECUs to the Raspberry Pi, which is crucial for real-time monitoring and control.

Secondly, the open-source nature of Linux offers a wealth of libraries and tools that streamline the development process. For instance, libraries such as OpenCV for computer vision and various UART communication libraries are readily available and easily integrated into the project. This availability reduces development time and allows us to focus on customizing the software to meet our specific needs. The ability to leverage pre-existing, well-maintained libraries not only accelerate the development cycle but also enhances the reliability and performance of the final application.

Moreover, the modularity and configurability of Embedded Linux make it an ideal choice for our project. The Raspberry Pi, equipped with its touch screen and camera, requires an operating system that can be tailored to optimize both user interaction and computational tasks. Embedded Linux allows us to create a streamlined, efficient environment that prioritizes the necessary drivers and services, ensuring that the system runs smoothly even with the limited resources typical of embedded systems.

In the context of our project, Embedded Linux also supports the integration of advanced functionalities such as computer vision. Utilizing the Raspberry Pi's camera, we can implement a computer vision model to analyze visual data in real time. The powerful processing capabilities of Embedded Linux, combined with the Raspberry Pi’s hardware, enable sophisticated image processing and analysis, which are essential for applications requiring high levels of automation and precision.

This section will delve into the specifics of setting up the Embedded Linux environment on the Raspberry Pi, detailing the configuration of the necessary libraries and tools. We will explore the steps required to install and configure the operating system, set up the UART communication with the ECUs, and integrate the touch screen interface. By the end of this chapter, readers will have a comprehensive understanding of how Embedded Linux underpins the functionality and performance of our graduation project, providing a robust foundation for further development and innovation.

4.2.2. Setting up the environment

Setting up the environment for our project involves configuring the Raspberry Pi 4 along with its peripherals, including the 7-inch touch

screen, camera module, and microSD card loaded with the Raspbian OS. This section provides a step-by-step guide to ensure a smooth and efficient setup process, laying the groundwork for the subsequent development and deployment of our application.

Components and Requirements

Before diving into the setup process, let's briefly outline the components we'll be working with:

Raspberry Pi 4 (8 GB): A powerful, versatile single-board computer that serves as the project's central processing unit.

Raspberry Pi 7-inch Touch Screen: Provides a user-friendly interface for interacting with the system.

Raspberry pi camera module 3: Captures images and videos for the computer vision model.

MicroSD Card (16GB or higher): Stores the operating system and project files.

Raspbian OS: A Debian-based operating system optimized for the Raspberry Pi, offering a robust platform for development.

Step-by-Step Setup

Figure 19 4.2.2. Configuring the Raspberry pi imager

Preparing the MicroSD Card

Download Raspbian OS:

Visit the official Raspberry Pi website and download the latest version of Raspbian OS. We recommend using the "Raspberry Pi OS with desktop" version for ease of use.

Flash Raspbian OS to the MicroSD Card:

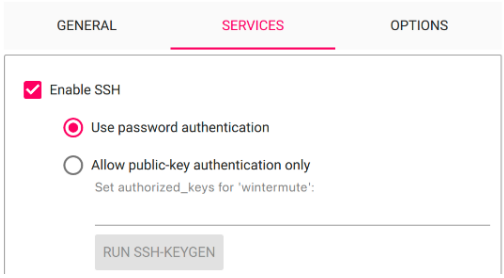


Figure 20 4.2.2. Enable the SSH

Use a tool like Raspberry Pi Imager to write the downloaded Raspbian OS image to your microSD card. Insert the microSD card into your computer, select the Raspbian OS image, the version of the used raspberry pi, and select the storage device which will be used to boot the OS. Then set the configurations that will appear to you after you

click “next” and don’t forget to enable the SSH which is a fundamental tool for remotely connecting and managing your Raspberry Pi.

Setting Up the Raspberry Pi

Insert the MicroSD Card:

Carefully insert the prepared microSD card into the microSD slot on the Raspberry Pi 4.

Connect the Touch Screen:

Attach the Raspberry Pi 7-inch touch screen to the Pi using the provided adapter board and ribbon cable. Ensure that the connections are secure and that the display and touchscreen functionalities are properly connected to the GPIO pins and DSI port.

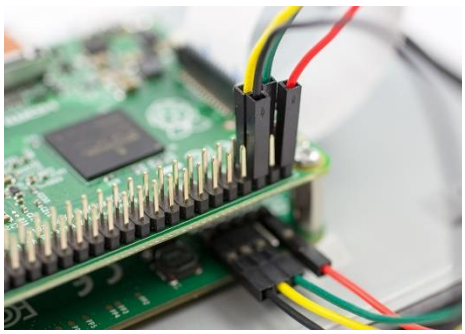
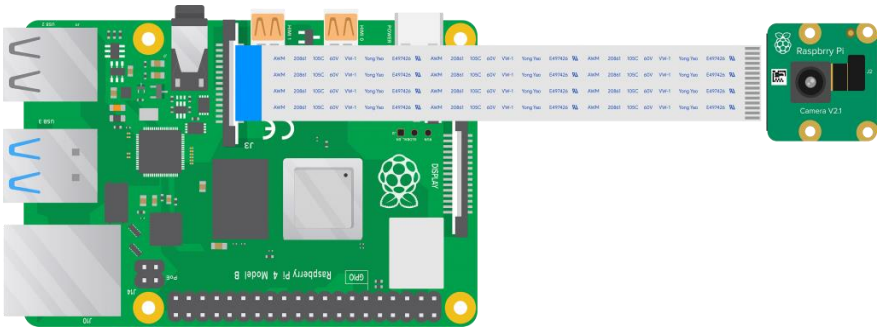


Figure 21 4.2.2. Pin connection of the display and the raspberry

Connect the Camera Module:

Attach the camera module to the Raspberry Pi via the CSI port. Secure the connection with the ribbon cable and ensure the camera module is firmly in place.

Figure 22 4.2.2. Camera connection with Raspberry pi via CSI port



Power Up the Raspberry Pi:

Connect a power supply to the Raspberry Pi. The recommended power supply is a 5V/3A adapter to ensure stable operation.

Initial Configuration

Boot UP:

Power on the Raspberry Pi. It should boot into the Raspbian OS. If this is the first time booting, you may need to go through the initial setup wizard

to configure basic settings such as language, time zone, and Wi-Fi connection.

Update and Upgrade:

Open a terminal and run the following commands to update the system

```
sudo apt update
```

```
sudo apt upgrade -y
```

Enable Interfaces:

Enable the camera and UART interfaces. Open the Raspberry Pi configuration tool by running the following command

```
sudo raspi-config
```

then Navigate to "Interfacing Options" and enable the camera and serial interfaces.

Configuring the Touch Screen

Calibrate the Touch Screen:

Depending on your specific touch screen model, you may need to install additional drivers or calibration tools. Follow the manufacturer's instructions to ensure the touch functionality is accurate.

Adjust Screen Resolution:

If necessary, adjust the screen resolution to optimize the display for your application. This can be done through the Raspberry Pi configuration tool or by editing the /boot/config.txt file.

Final Steps

With the hardware connected and the software environment configured, your Raspberry Pi setup is complete. This environment provides a stable and powerful foundation for developing the main application, enabling efficient communication with the ECUs, effective use of the touch screen interface, and integration of the computer vision model. In the next sections, we will dive into the specifics of writing and deploying the software that will run on this meticulously prepared environment, ensuring that our project functions seamlessly and meets all its intended objectives.

4.2.3. Libraries

Let's have a glimpse of the libraries used in this project and know how to install them if they aren't installed. We used three libraries to make the infotainment system, the libraries are TKinter, PIL, and datetime.

4.2.3.1. Tkinter: Building a Graphical User Interface

Tkinter is the standard Python library for creating graphical user interfaces (GUIs). It is cross-platform and runs on Windows, macOS, and Linux. Tkinter is widely used due to its simplicity and effectiveness in developing basic to moderately complex GUI applications. In our project, Tkinter plays a crucial role in providing an intuitive and interactive interface on the Raspberry Pi touch screen, allowing users to interact with the system seamlessly.

Features of Tkinter

- Widgets:

  Tkinter provides a wide variety of widgets, such as buttons, labels, text boxes, frames, and canvases, which are the building blocks of any GUI application. These widgets are highly customizable, allowing developers to modify their appearance and behavior to suit the application's needs.

- Geometry Management:

Tkinter offers several geometry managers, including pack, grid, and place, to control the layout of widgets within the window. These managers enable developers to organize widgets in a flexible and intuitive manner, ensuring that the GUI remains user-friendly and visually appealing.

- Event Handling:

One of Tkinter's strengths is its robust event handling system. It allows developers to bind functions or methods to various events, such as button clicks, key presses, and mouse movements. This feature is essential for creating responsive and interactive applications.

Installing Tkinter:

In the Linux terminal and run the following command

```
sudo apt-get install python-tk
```

4.2.3.2. PIL: Handling Images

The Python Imaging Library (PIL), now known as Pillow, is a powerful library for opening, manipulating, and saving many different image file formats. ImageTk is a module in PIL that provides support for displaying images in Tkinter applications. These libraries are integral to our project as they enable the display of the photos in the GUI.

PIL provides extensive capabilities for image processing, including resizing, cropping, rotating, filtering, and enhancing images.

PIL supports a wide range of image file formats, such as JPEG, PNG, BMP, GIF, and TIFF.

The ImageTk module in Pillow allows images to be displayed in Tkinter applications. It converts PIL images into a format that Tkinter can use.

4.2.3.2. Datetime: Handling Date and Time

The datetime module in Python provides classes for manipulating dates and times. It is a versatile and essential library for applications that require timestamping, scheduling, or time-based calculations. In our project, datetime is used to display the current date and time, enhancing the functionality and user experience of the GUI.

The module provides extensive support for formatting date and time objects into readable strings and parsing strings into date and time objects. This is essential for displaying timestamps in a user-friendly format and for handling user input.

In summary, the libraries TKinter, PIL.Image, ImageTk, and datetime each play vital roles in the functionality and user experience of our project. Tkinter provides a robust and user-friendly GUI framework, enabling interactive and visually appealing interfaces. PIL and ImageTk facilitate the handling and display of images, which is crucial for integrating photos with the GUI. Lastly, the datetime module allows for precise management and display of date and time information. Together, these libraries form the backbone of our project's software environment, ensuring a seamless and efficient user experience.

4.2.4. Infotainment Program

An infotainment system is an integrated in-car technology that combines entertainment and information delivery through a centralized interface.

These systems enhance both driver convenience and passenger experience. By offering intuitive controls and advanced functionalities, infotainment systems have become a standard component in modern vehicles, contributing to a more connected and enjoyable driving experience.

4.2.4.1. Dashboard

The dashboard (GUI) used in this project is done using Tkinter. Before diving in the details of the GUI, let's list some of the available choices and clarify why Tkinter chosen to do this project.

To do a GUI for our program that enhance the user experience we have 3 choices, QML with PyQt, Kivy, and Tkinter. The following section gives a glimpse of QML and Kivy, we have talked about Tkinter previously

QML: it is a powerful and flexible declarative language primarily used for designing user interfaces as part of the Qt framework. It supports a wide range of visual elements and effects, making it suitable for developing modern applications with sophisticated UI requirements. Additionally, QML's integration with the Qt framework means it can be used across multiple platforms, including desktop, mobile, and embedded systems, ensuring broad applicability and consistency in user experience.

Kivy: it is an open-source Python library for rapid development of applications that require innovative multi-touch user interfaces. Designed to work across a wide range of platforms, including Windows, macOS, Linux, Android, and iOS. It comes with a rich set of UI controls, ranging from buttons and text inputs to more complex widgets like scroll views and gestures. Kivy's architecture is based on an event-driven framework, which is particularly well-suited for handling touch and gesture inputs, making it ideal for mobile.

After having an idea about the available choices let's know why Tkinter is chosen instead of Kivy and QML.

As our project mainly focuses on running two AI models locally on the raspberry pi without needing to go online and use any cloud service for processing the data, and achieve the reliability, real time communication and stability of the connection between the ECUs and the parent computer (raspberry pi), so we were needing a choice that is not complex and can be used to meet our needs as simple as possible,

although other choices can be more powerful but we will not use its power features, so we used the simplest and fastest tool to implement the needed requirements.

Implementation:

One of the main disadvantages of Tkinter is it looks old and don't provide flexibility to customize the widgets, so the result GUI made with Tkinter will not look modern. Because we need to make a modern UI with Tkinter we used some work around practices to make our dashboard looks modern. In the following we will illustrate how this is done.

Background:

In any GUI there are some static parts in the window that will not change through the running of the program, so instead of making this background using the widgets of Tkinter will shorten the time of developing and draw the static part of the ground on any drawing software, we used Figma which is the recommended solution to design the user interface.

After drawing the static part on the drawing software export your drawing into an image with any extension supported by Tkinter, we used .png extension.

When you have the background as an image, you can set it as a background to the GUI by the following method:

```
resized_image = Image.open("background.png").resize((800, 480))

background_image = ImageTk.PhotoImage(resized_image)

background = tk.Label(root, image=background_image)

background.pack(fill=tk.BOTH, expand=True)
```

Line 1: Opens and resizes an Image. It uses the `Image.open()` function from the PIL library to open an image file named "background.png". Imagine this function as a special tool that can unlock and retrieve the contents of the image file. It uses the `resize()` method on the image, This method takes a tuple containing two values, (800, 480). These values represent the new width and height of the image in pixels which are the same dimensions of the window.

Line 2: uses the `ImageTk.PhotoImage()` function to convert the resized image (`resized_image`) into a format that Tkinter can understand and display. Tkinter, the graphical user interface library, can't directly work with standard Python image objects. This function acts like a translator, taking the image data and converting it into a special Tkinter image format called a PhotoImage. The resulting `resized_image` variable now holds the image data in a Tkinter-compatible format, ready to be displayed on the screen.

Lines 3,4: This line creates a label widget named "background" and places it on the main window ("root") of Tkinter application. The magic happens in two parts:

Make a label act like a picture frame. We tell the label to display the image we prepared earlier (`background_image`) by setting the image property.

Positioning and filling: `.pack(fill=tk.BOTH, expand=True)`positions and stretches the label. Here, `fill=tk.BOTH` tells the label to fill both the width and height of the window. `expand=True` ensures the label expands if the window is resized, keeping the background image filling the available space.

Labels:

The normal shape of labels in Tkinter is old like the following picture, so if you need to make it more modern you can use the following method:

Figure 23 4.2.4.1. Classic shape of label in Tkinter

```
date_label = tk.Label(root,
text=gui_date(),fg="white",bg="#00D1FF", borderwidth=0,
font=("Helvetica", 15,"bold")).place(x=500,y=30)
```

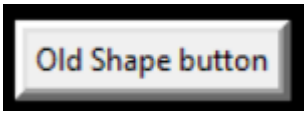
this code makes a label and prints in it the return of a function returns the date, to make it more modern, set the color of background of the label as same as the background behind the label, after adjusting the background of the label make the border of the label = 0, to give the effect of the label is floating in the window, then you can adjust the characteristics of the font like its size and type, then place the label at the needed positions.

Figure 25 4.2.4.1. Modern shape of label

Figure 24 4.2.4.1. Classic shape of button in Tkinter

Buttons:

The normal shape of labels in Tkinter is old like the following picture, so if you need to make it more



modern and put a picture inside this button and make this picture is the button, you can use the following method :

```
resized_image = Image.open("unlock.png").resize((50, 50))
unlock_image = ImageTk.PhotoImage(resized_image)
unlock_button = tk.Button(root, image=unlock_image, command=unlock,
borderwidth=0,bg="#1E1E1E",activebackground="#1E1E1E").place(x=531,y=340)
```

We need to make a picture acts like a button, to make this we follow some steps. First, bring the needed picture and adjust its size to the needed size of the button. Second, convert the resized image (`resized_image`) into a format that Tkinter can understand and display. Third, set the image of the button to the resized image, link the button with the function that will be run when the button is pressed, set the border of the width to 0 to make the button like it is floating, set the background of the button like the background behind the button, and set the active background, which is the color of the background when the button is clicked, to the same color of the background so the background of the button not changes when the button is clicked. Fourth, place the button at the needed position

Figure 26 4.2.4.1. Modern shape of button

4.2.4.2. Back-end program

Let's talk about the logic of the program of the dashboard:
The services that the program of the dashboard provide are reading the output of the Ai models and update the dashboard according to these outputs, receive UART messages from an ECU (which works as a gateway) and update the dashboard according to the received messages, and send UART messages to the ECU (gateway) to redirect it to the wanted ECU.

Read the output of the Ai models:

The two Ai models which monitor the driver and detect if there is any forgotten child will write their output in a text file, this text file will be in the middle of the program of the dashboard and the two Ai models. So, there will be a predefined codes that will define the state of the driver and the existence of the child. Using the following code the program can read the file of the output of the Ai models

```
file = open('Ai_output.txt', 'r')
```

```
Ai_data = file.read()
```

After reading the data from the file, this read date can be used to decide which action should be done. For example, if the driver slept, the raspberry pi would produce an alert on the dashboard and alarm from a sound source to notify the driver to wake up. Another example, if the driver fainted, the raspberry pi would send a UART message to the ECU that is responsible for sending SMS, we will discuss the part of sending UART messages in the following section. Another last example, if there is a forgotten child in the car, the raspberry pi would send a UART message to the ECU that is responsible for sending SMS, and make the car turn on the klaxon so anyone near can save the child.

Receive UART messages:

UART is the communication protocol between the raspberry and other ECUs, so any data the ECUs want to send to it to the raspberry pi for presentation or processing is sent through the UART

In raspberry pi the receive of UART messages is done through the following code:

```
import serial
ser = serial.Serial('/dev/ttyS0', 9600, timeout=1)
data = ser.read(10)
# some code will be done based on the received data
ser.close()
```

The provided Python code snippet demonstrates how to establish a UART communication channel on a Raspberry Pi using the `pyserial` library. Initially, the `serial` module is imported to enable serial communication functionality. A `Serial` object named `ser` is then created, specifying the UART port '/dev/ttyS0' (which can be adjusted to a specific UART port) and the baud rate (`9600`), along with a timeout of 1 second. This setup initializes the UART connection, allowing the Raspberry Pi to communicate with other devices via the specified port. The `ser.read(10)` function reads up to 10 bytes of data from the serial port, storing the received data in the `data` variable. After the data is read any action can be done according to the read data. Finally, the `ser.close()` method is called to close the serial connection, ensuring that resources are properly released, and the communication channel is cleanly terminated.

The ECUs can send data like the temperature of the car, speed of the car, battery percentage, doors status (locked, unlocked).

Send UART messages:

In raspberry pi the send of UART messages is done through the following code:

```
import serial

ser = serial.Serial('/dev/ttyS0', 9600, timeout=1)

ser.write(b'Hello, UART!')

ser.close()
```

The provided Python code snippet demonstrates how the send of UART messages is done, we can notice it looks like the previous code, which is for receiving UART messages, the difference is we replaced the sentence of read from the serial with write function, which takes the bytes of the message wanted to be sent to write it on the serial.

The raspberry pi can send data like asking the ECUs to send SMS, or generate any sounds or notifications, or send data to tell the ECU the temperature that the atmosphere of the car should be set on.

The messages that will be transmitted between the raspberry and the ECUs are encoded and any of them receive any code, it knows what to do at this case.

4.2.4.3. AutoStart the infotainment system

We'll explore how to configure your Raspberry Pi to automatically run the main Python program at startup. This ensures that your application begins operating as soon as the Raspberry Pi boots, creating a seamless and user-friendly experience. There are several methods to achieve this, but we will focus on one approach, which is adding the program to the system's crontab.

The cron daemon (crontab) is a time-based job scheduler in Unix-like operating systems, including Raspbian. By adding a job to the crontab, you can specify tasks to run at specific times or events, such as system boot.

Steps to Auto-Run Using Crontab

Open the Crontab Editor:

Open a terminal on your Raspberry Pi and type the following command to edit the crontab file for the current user: `crontab -e`

Add the Python Program to the Crontab:

In the crontab editor, add the following line at the end of the file:
`@reboot /usr/bin/python3 /path/to/your/script.py`

Replace */path/to/your/script.py* with the absolute path to your Python script. Ensure that you use the correct path to the Python interpreter, which is typically */usr/bin/python3* for Python 3.

Save and Exit:

Save the changes and exit the editor. The method to save and exit depends on the editor you're using (e.g., nano, vim). For nano, press CTRL+X, then Y, and finally ENTER.

Reboot to Test:

Reboot your Raspberry Pi to test if the script runs automatically:
`sudo reboot`

4.3. Embedded BareMetal Implementation

4.3.1. Tools for developing

We utilized two essential IDEs for generating and debugging code, as well as for uploading or programming code into the STM "Blue Pill."

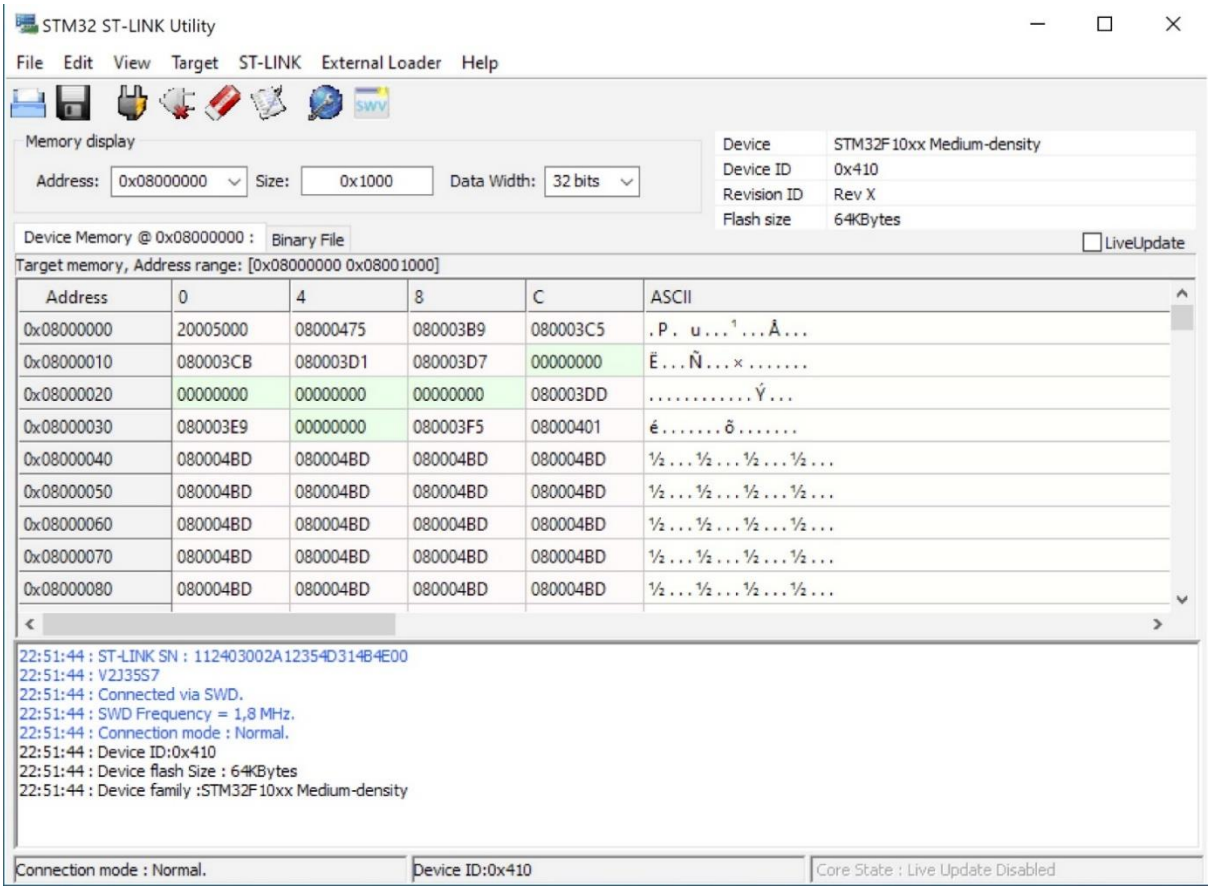
First STM32CubeIDE:

is an integrated development environment (IDE) designed for STM32 microcontrollers. Developed by STMicroelectronics, it combines comprehensive project management, code editing, and debugging tools within a single platform. STM32CubeIDE supports various programming languages, including C and C++, and integrates seamlessly with STM32CubeMX for graphical configuration of MCU peripherals. It offers advanced debugging capabilities, including real-time

variable monitoring, and supports multiple compilers, making it a powerful tool for embedded system development.

And STM32 ST-LINK Utility:

is a software tool for programming and debugging STM32 microcontrollers. It interfaces with the ST-LINK/V2 or ST-LINK/V2-1 in-circuit debugger/programmer. The utility provides features for downloading program code to the MCU's flash memory, performing memory checks, and verifying and erasing flash content. It also includes functionalities for viewing and modifying memory contents, making it a valuable tool for low-level programming and debugging tasks.



And STM32 ST-LINK/V2:

is an in-circuit debugger and programmer for STM32 microcontrollers. It interfaces with the MCU through JTAG or SWD (Serial Wire Debug) protocols, enabling flash programming, debugging, and real-time monitoring. The device supports connection to both STMicroelectronics software tools and third-party IDEs, providing a versatile and efficient solution for embedded system development. The ST-LINK/V2 enhances development efficiency by facilitating seamless code download, execution control, and comprehensive debugging.

4.3.2. GPIO

General-Purpose Input/Output (GPIO) is a crucial component in the realm of microcontroller systems, serving as the primary interface for digital signal interaction between the microcontroller and external components. The versatility of GPIO pins allows them to function in various modes, including input, output, alternate function, and analog, making them indispensable in a wide range of applications. In embedded systems, GPIO pins are used to read digital inputs from external devices, such as switches, sensors, and buttons, and to control digital outputs, such as LEDs, motors, and relays. The ability to configure each GPIO pin individually provides significant flexibility in design, enabling the microcontroller to adapt to diverse peripheral devices and operational requirements. The configuration of GPIO pins is a fundamental task in embedded system development. Each pin can be set to function as either an input or an output, depending on the desired operation. Input pins are used to sense digital signals, allowing the microcontroller to detect the state of external devices. Output pins, on the other hand, are used to send digital signals to control external devices. Additionally, GPIO pins can be configured with pull-up or pull-down resistors to ensure stable signal levels, preventing floating states that can lead to unreliable behavior.

4.3.2.1. GPIO Pin Configuration

Proper configuration ensures that each GPIO pin operates correctly according to the specific needs of the application, whether it involves reading digital inputs, driving outputs, or handling more complex tasks such as interrupt-driven events. This section details the steps and considerations involved in configuring GPIO pins to ensure reliable and efficient operation.

4.3.2.1.1. Understanding GPIO Modes

The STM32F103C6 microcontroller offers several modes for GPIO pins, each serving different purposes:

Input Mode: In this mode, a GPIO pin is used to read digital signals from external devices such as sensors or switches. The microcontroller can detect high or low logic levels on the pin.

Output Mode: Here, the GPIO pin drives digital signals to control external devices like LEDs, relays, or motors. The pin can be set to high or low states to turn devices on or off.

Alternate Function Mode: This mode allows a GPIO pin to perform special functions beyond simple input or output tasks. For instance, it can be used for communication interfaces such as UART, SPI, or I2C.

Analog Mode: In this mode, the GPIO pin can interface with analog devices, such as sensors providing variable voltage levels, which are then converted to digital values by the ADC.

4.3.2.1.2. Steps for GPIO Pin Configuration

Pin Selection: The first step is to identify which GPIO pin(s) will be used for the desired operation. The STM32F103C6 has several GPIO ports (A, B, C, etc.), each with multiple pins. The selection depends on the pin's availability and its alternate functions if needed.

Mode Setting: Using the microcontroller's configuration registers, each pin must be set to the appropriate mode (input, output, alternate function, or analog). This is typically done by configuring the GPIO Mode Register (GPIOx_MODER).

Speed Configuration: For output pins, the speed at which the pin can toggle must be set. This can range from low to very high speed,

depending on the application's requirements. The speed setting is adjusted via the GPIO Speed Register (GPIOx_OSPEEDR).

Pull-up/Pull-down Configuration: Input pins may need pull-up or pull-down resistors to ensure stable signal levels. This prevents the pin from floating, which can lead to erratic behavior. These resistors are configured using the GPIO Pull-up/Pull-down Register (GPIOx_PUPDR).

Output Type Configuration: For output pins, the type (push-pull or open-drain) must be specified. Push-pull configuration allows the pin to drive both high and low states, while open-drain configuration can only drive low, requiring an external pull-up resistor to achieve a high state.

Alternate Function Configuration: If a pin is used for an alternate function, the specific function must be selected by configuring the Alternate Function Register (GPIOx_AFR). This register maps the pin to the desired peripheral function.

4.3.2.1.3. Example: Configuring GPIO Pins in STM32F103C6

Consider a scenario where we need to configure a GPIO pin on the STM32F103C6 for an input with a pull-up resistor, and another pin for output at high speed. The following steps outline the process:

Select Pins: Assume we choose pin PA0 for input and pin PB0 for output.

Mode Setting:

For PA0: Set the mode to input by configuring the GPIOA_MODER register.

For PB0: Set the mode to output by configuring the GPIOB_MODER register.

Speed Configuration:

For PB0: Set the speed to high by configuring the GPIOB_OSPEEDR register.

Pull-up Configuration:

For PA0: Enable the pull-up resistor by configuring the GPIOA_PUPDR register.

Output Type Configuration:

For PB0: Set the output type to push-pull by configuring the GPIOB_OTYPER register.

4.3.2.2. GPIO in STM32F103C6

General-Purpose Input/Output (GPIO) pins are fundamental to embedded systems, providing a flexible interface for interacting with a variety of peripherals and external components. The STM32F103C6 microcontroller, widely used in both academic and industrial settings, offers robust GPIO functionality. This section delves into the specifics of GPIO in the STM32F103C6, covering its configuration, capabilities, and practical applications, ensuring a comprehensive understanding for learners and professionals alike.

4.3.2.2.1. Introduction to GPIO in STM32F103C6

The STM32F103C6 microcontroller features several GPIO ports (A, B, C, etc.), each with multiple pins that can be configured as input or output, and can be used for digital signaling or interfacing with sensors, actuators, and other peripherals. Each GPIO pin can be individually controlled, making it possible to customize the microcontroller's interface according to the needs of the application.

4.3.2.2.2. GPIO Pin Configuration

Configuring GPIO pins in the STM32F103C6 involves several steps, starting with enabling the clock for the relevant GPIO port, and then setting the pin mode, configuration, and speed. The microcontroller's register-based approach allows for precise control over each pin's behavior.

Enabling the Clock: Before configuring any GPIO pin, the clock for the respective GPIO port must be enabled `.RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;` // Enable clock for GPIO Port A

Setting Pin Mode and Configuration: Each pin can be configured for input, output, alternate function, or analog mode, with additional settings for pull-up/pull-down resistors.

Example: Configuring PA0 as a digital input with a pull-up resistor:

```
GPIOA->CRL &= ~GPIO_CRL_MODE0;      // Input mode
GPIOA->CRL |= GPIO_CRL_CNF0_1;      // Input with pull-up/pull-down
```

```
GPIOA->ODR |= GPIO_ODR_ODR0;          // Enable pull-up
```

Example: Configuring PB0 as a digital output with a push-pull configuration and a maximum speed of 50 MHz:`GPIOB->CRL &= ~GPIO_CRL_MODE0;` // Clear mode bits

```
GPIOB->CRL |= GPIO_CRL_MODE0_1;      // Output mode, 50 MHz
GPIOB->CRL &= ~GPIO_CRL_CNF0;        // Push-pull configuration
```

4.3.2.2.3. *GPIO Operations*

Once configured, GPIO pins can be used for various operations, such as reading input values or setting output states. These operations are crucial for interacting with other components in an embedded system.

Reading Input Value: The state of an input pin can be read from the IDR (Input Data Register).

uint8_t `pin_state = (GPIOA->IDR & GPIO_IDR_IDR0) ? 1 : 0;` // Read state of PA0

Setting Output Value: The state of an output pin can be set using the ODR (Output Data Register) or the BSRR (Bit Set/Reset Register) for atomic operations.

```
GPIOB->ODR |= GPIO_ODR_ODR0;  // Set PB0 high
GPIOB->ODR &= ~GPIO_ODR_ODR0; // Set PB0 low


// Alternatively, using BSRR for atomic operations
GPIOB->BSRR = GPIO_BSRR_BS0;  // Set PB0 high
GPIOB->BSRR = GPIO_BSRR_BR0;  // Set PB0 low
```

4.3.2.2.4. *Practical Applications of GPIO in Our Project*

In the context of our Advanced Driver Assistance System (ADAS) project, GPIO pins play an integral role in interfacing with various sensors and actuators. For instance, GPIO pins are used to read the state of buttons and switches, control indicator LEDs, and interface with communication modules.

Driver Monitoring: GPIO pins can be connected to infrared sensors or cameras used for detecting driver alertness and presence.

The sensors provide digital signals that can be read through GPIO inputs.

Child Safety: Sensors placed in car seats to detect the presence of a child can be connected to GPIO pins. The system can then trigger alerts if a child is left unattended in the vehicle.

Environmental Monitoring: Temperature and humidity sensors connected via GPIO pins provide critical data for ensuring the safety and comfort of the vehicle's occupants.

4.3.2.2.5. Communication with Peripheral Devices

GPIO pins also facilitate communication with various peripheral devices, such as indicator LEDs and buzzers, which provide visual and auditory alerts. For instance, an LED connected to a GPIO pin can be turned on or off to signal different system states, such as alerts for driver drowsiness or child safety warnings.// Set PC13 (LED) high to turn on the indicator LED

```
GPIOC->BSRR = GPIO_BSRR_BS13;
```

// Set PC13 (LED) low to turn off the indicator LED

```
GPIOC->BSRR = GPIO_BSRR_BR13;
```

4.3.2.2.6. Integration with Communication Protocols

In addition to direct interfacing, GPIO pins are integral in supporting communication protocols like I2C, SPI, and UART. These protocols are used to interface with more complex sensors and modules, expanding the system's functionality.

For example, the I2C protocol can be used to communicate with a digital temperature sensor module, where GPIO pins are configured for I2C communication lines (SDA and SCL).

// Configure GPIO pins for I2C communication

```
GPIOB->CRL |= (GPIO_CRL_MODE6 | GPIO_CRL_CNF6);
```
// Configure PB6 (SCL)

```
GPIOB->CRL |= (GPIO_CRL_MODE7 | GPIO_CRL_CNF7);
```
// Configure PB7 (SDA)

4.3.3. RCC

Reset and Clock Control (RCC) is a fundamental component in microcontroller systems. The RCC subsystem is responsible for

managing the clock signals and reset mechanisms that orchestrate the operation of the entire microcontroller. It ensures that various peripherals and modules within the microcontroller are synchronized and operate at optimal frequencies, thereby guaranteeing the stability and efficiency of the system.

At its core, the RCC module is tasked with generating and distributing clock signals to different parts of the microcontroller. These clock signals are derived from various sources, including internal and external oscillators. Internal oscillators offer the convenience of an integrated clock source, eliminating the need for additional external components. External oscillators, on the other hand, provide higher accuracy and stability, which are essential for time-sensitive applications. The ability to select and switch between these clock sources adds flexibility to the system, allowing it to adapt to different performance and power consumption requirements.

The clock control aspect of RCC is pivotal in determining the operating frequency of the microcontroller's core and its peripherals. Through the use of prescalers and multipliers, the RCC can adjust the clock frequency to match the specific needs of various modules. This dynamic control not only enhances performance but also contributes to energy efficiency by reducing the clock speed during low-power operations. Equally important is the reset control functionality provided by the RCC. Resets are essential for initializing the microcontroller and ensuring that it starts in a known state. The RCC supports various types of resets, including power-on reset, which occurs when the microcontroller is first powered up; software reset, which can be triggered programmatically; and external reset, initiated by an external signal. These reset mechanisms are critical for recovering from errors, maintaining system integrity, and enabling safe reboots.

In subsequent sections, we will delve deeper into the architecture of RCC, exploring its components, clock sources, clock distribution mechanisms, and reset strategies. This comprehensive examination will provide a solid foundation for configuring and utilizing RCC effectively in the development of advanced embedded systems.

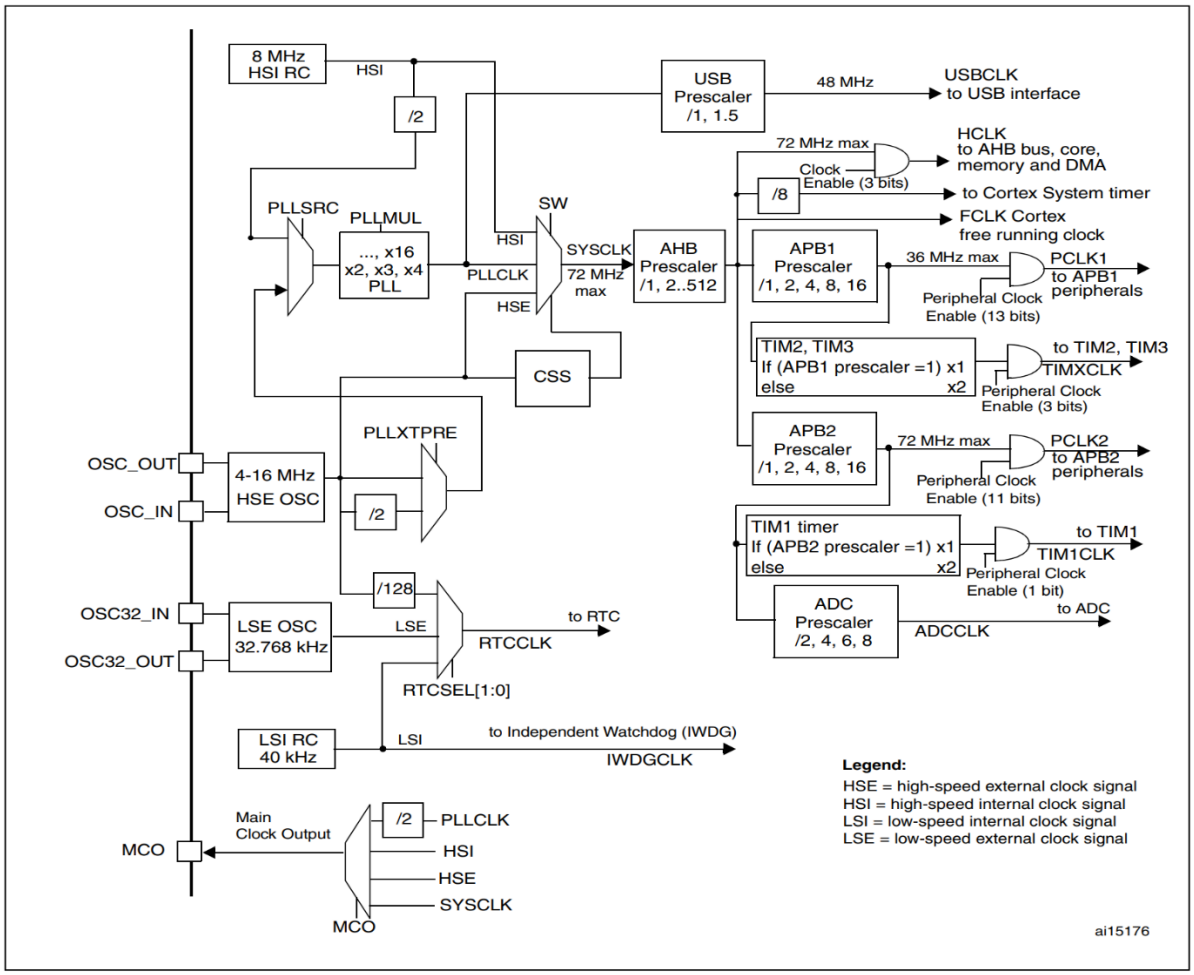


Figure 27 4.3.3. A detailed diagram illustrating the clock sources, prescalers, and distribution paths for various subsystems and peripherals in the STM32F103C6.

4.3.3.1. Clock Sources

Clock sources are fundamental components in embedded systems, providing the necessary timing signals that drive the operation of the entire microcontroller. Understanding the different types of clock sources and their respective functionalities is crucial for ensuring precise and reliable system performance.

In the context of the STM32 microcontroller series, the Reset and Clock Control (RCC) unit offers multiple clock sources, each serving distinct purposes and offering unique characteristics. The primary clock sources include the High-Speed External (HSE) clock, the High-Speed Internal (HSI) clock, the Low-Speed External (LSE) clock, the Low-Speed Internal (LSI) clock, and the Phase-Locked Loop (PLL) clock.

The **High-Speed External (HSE)** clock is typically an external crystal or resonator connected to the microcontroller. It provides a

high-frequency and stable clock signal, making it ideal for applications requiring precise timing and synchronization.

Conversely, the **High-Speed Internal (HSI)** clock is an internal oscillator embedded within the microcontroller. It offers a convenient and cost-effective alternative to the HSE clock, eliminating the need for external components. While the HSI clock may not match the accuracy of an HSE clock, it is sufficient for many general-purpose applications and provides a quick startup time, which is beneficial in power-sensitive designs.

The **Low-Speed External (LSE)** clock, similar to the HSE clock, is an external crystal or resonator but operates at a lower frequency. It is commonly used for low-power applications and real-time clock (RTC) functionality, where maintaining accurate timekeeping with minimal power consumption is essential.

The **Low-Speed Internal (LSI)** clock, an internal oscillator, provides a low-frequency clock signal suitable for low-power and low-frequency applications. It is typically used as a backup clock source for the RTC and watchdog timer, ensuring continued operation even if other clock sources fail.

The **Phase-Locked Loop (PLL)** clock is a sophisticated clock source that can generate higher frequencies by multiplying the frequency of a lower-frequency input clock. The PLL clock offers flexibility in achieving various clock frequencies required by different parts of the microcontroller.

Each clock source can be selected and configured through the RCC registers, providing designers with the ability to tailor the microcontroller's clocking scheme to their application's requirements. The choice of clock source impacts the system's performance, power consumption, and timing accuracy, making it a critical consideration during the design phase.

4.3.3.2. Detailed Configuration of RCC

The detailed configuration of the Reset and Clock Control (RCC) in the STM32F103C6 microcontroller involves precise adjustments to various registers to achieve the desired clock settings for optimal performance and power efficiency. In this section, we will delve into the step-by-step process of configuring the RCC, providing a comprehensive understanding of how to set up the system clock,

configure the PLL, manage peripheral clocks, and implement low-power modes.

4.3.3.2.1. System Clock Configuration

The system clock (SYSCLK) is the primary clock source that drives the CPU and the majority of the peripherals. Configuring the SYSCLK involves selecting the appropriate clock source and setting the necessary prescalers to achieve the target frequency. In our project, we aim to use the High-Speed External (HSE) oscillator as the main clock source, with the Phase-Locked Loop (PLL) to multiply the frequency for higher performance.

To begin, we enable the HSE oscillator and wait for it to stabilize. Next, we configure the PLL to use the HSE as its input source and set the multiplication factor to achieve the desired SYSCLK frequency. Finally, we select the PLL as the system clock source and configure the prescalers for the Advanced High-Performance Bus (AHB), Advanced Peripheral Bus 1 (APB1), and Advanced Peripheral Bus 2 (APB2).

Here is an example of the code to configure the system clock to 72 MHz using the HSE and PLL:

// Enable HSE oscillator

```
RCC->CR |= RCC_CR_HSEON;
```

while `(!(RCC->CR & RCC_CR_HSERDY));`

// Configure the PLL

```
RCC->CFGR |= RCC_CFGR_PLLSRC_HSE | RCC_CFGR_PLLMULL9;
```

// Enable the PLL

```
RCC->CR |= RCC_CR_PLLON;
```

while `(!(RCC->CR & RCC_CR_PLLRDY));`

// Configure prescalers

```
RCC->CFGR |= RCC_CFGR_HPRE_DIV1; // AHB prescaler
RCC->CFGR |= RCC_CFGR_PPRE1_DIV2; // APB1 prescaler
RCC->CFGR |= RCC_CFGR_PPRE2_DIV1; // APB2 prescaler
```

// Select PLL as system clock source

```
RCC->CFGR |= RCC_CFGR_SW_PLL;
```

while `((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);`

4.3.3.2.2. Peripheral Clock Management

Efficient management of peripheral clocks is crucial for optimizing power consumption and ensuring that each peripheral operates at the appropriate frequency. The RCC allows for granular control over the clock sources and prescalers for individual peripherals.

In our project, the peripherals such as ADC, UART, and timers have specific clock requirements. For example, the ADC requires a lower clock frequency for accurate analog-to-digital conversion, while the UART needs a stable clock for reliable data communication.

Here is how we can configure the clocks for the ADC and UART:
// Enable clock for ADC1

```
RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
ADC1->CR2 |= ADC_CR2_ADON;
```

// Enable clock for USART1

```
RCC->APB2ENR |= RCC_APB2ENR_USART1EN;
USART1->CR1 |= USART_CR1_UE;
```

4.3.3.3. Practical Applications of RCC in Our Project

The practical applications of the RCC in our project are multifaceted, addressing various aspects of system performance, power efficiency, and reliability. By leveraging the RCC's capabilities, we can optimize the operation of the STM32F103C6 microcontroller for tasks such as driver monitoring, child safety, and communication.

Table: RCC Configuration for STM32F103C6 in Our Project

Configuration Item	Setting	Purpose
SYSCLK Source	HSE + PLL	High-performance system clock
SYSCLK Frequency	72 MHz	Optimized for processing-intensive tasks

Configuration Item	Setting	Purpose
AHB Prescaler	DIV1	Full speed for AHB peripherals
APB1 Prescaler	DIV2	Suitable speed for low-power peripherals
APB2 Prescaler	DIV1	Full speed for high-speed peripherals
CSS	Enabled	Ensures continuous operation on clock failure
RTC Clock Source	LSE	Accurate timing for real-time applications

4.3.4. ADC

Analog-to-Digital Converters (ADCs) are fundamental components in modern electronics, bridging the gap between the analog world and digital systems.

The core function of an ADC is to sample an analog signal and convert it into a digital representation. This process involves several steps: sampling the input signal at discrete intervals, quantizing the sampled values to a finite number of levels, and encoding these levels into a binary format. The quality of this conversion is determined by several key parameters, including resolution, sampling rate, and accuracy.

In instrumentation, ADCs are used to digitize signals from sensors measuring physical quantities like temperature, pressure, and light intensity. The digital data is then processed to monitor and control various parameters in real time. The STM32F103C6 microcontroller, widely used in embedded applications, features an integrated ADC. This ADC supports multiple input channels, allowing it to interface with various sensors simultaneously. It offers different modes of operation, such as single and continuous conversion, providing flexibility in how data is sampled and processed.

The Analog-to-Digital Converter (ADC) operates based on several fundamental principles and theories that govern the conversion of continuous analog signals into discrete digital values. Understanding these principles is crucial for designing and implementing effective ADC applications in embedded systems.

4.3.4.1. STM32F103C6 ADC Configuration

The STM32F103C6 microcontroller features a 12-bit ADC with up to 16 external channels, allowing for flexible and efficient sampling of multiple signals. The ADC supports various modes, including single, continuous, scan, and discontinuous mode, catering to different application requirements. Additionally, the microcontroller includes features like an analog watchdog for monitoring thresholds, automatic injection for prioritized channels, and temperature sensor measurement, enhancing its utility in embedded applications.

Table: ADC characteristics

Symbol	**Parameter**	**Conditions**	**Min**	**Typ**	**Max**	**Unit**
VDDA	Power supply	-	2.4	-	3.6	V
VREF+(3)	Positive reference voltage	-	2.4	-	VDDA	V
IVREF(3)	Current on the VREF input pin	-	-	160	220	µA
fADC	ADC clock frequency	-	0.6	-	14	MHz
fS(2)	Sampling rate	-	0.05	-	1	MHz
fTRIG(2)	External trigger frequency	fADC = 14 MHz	-	-	823	kHz
			-	-	17	1/fADC
VAIN(3)	Conversion voltage range	-	0	-	VREF+	V
RAIN(2)	External input impedance		-	-	50	kΩ
RADC(2)	Sampling switch resistance	-	-	-	1	kΩ

CADC(2)	Internal sample and hold capacitor	-	-	-	8	pF
tCAL(2)	Calibration time	fADC = 14 MHz	5.9	-	-	µs
			-	83	-	1/fADC
tlat(2)	Injection trigger conversion latency	fADC = 14 MHz	-	-	0.214	µs
			-	-	3	1/fADC
tlatr(2)	Regular trigger conversion latency	fADC = 14 MHz	-	-	0.143	µs
			-	-	2	1/fADC
tS(2)	Sampling time	fADC = 14 MHz	0.107	-	17.1	µs
			-	1.5	239.5	1/fADC
tSTAB(2)	Power-up time	-	0	0	1	µs
tCONV(2)	Total conversion time (including sampling time)	fADC = 14 MHz	1	-	18	µs
			-	14	252	1/fADC

4.3.4.2. Practical Applications of ADC in Our Project

The Analog-to-Digital Converter (ADC) in the STM32F103C6 microcontroller plays a pivotal role in our Advanced Driver Assistance System (ADAS) project. The ADC's ability to accurately convert analog signals into digital data enables a variety of critical monitoring and control functions, enhancing the safety and efficiency of the system. This section details the practical applications of the ADC in our project, providing a comprehensive understanding of its integration and utility

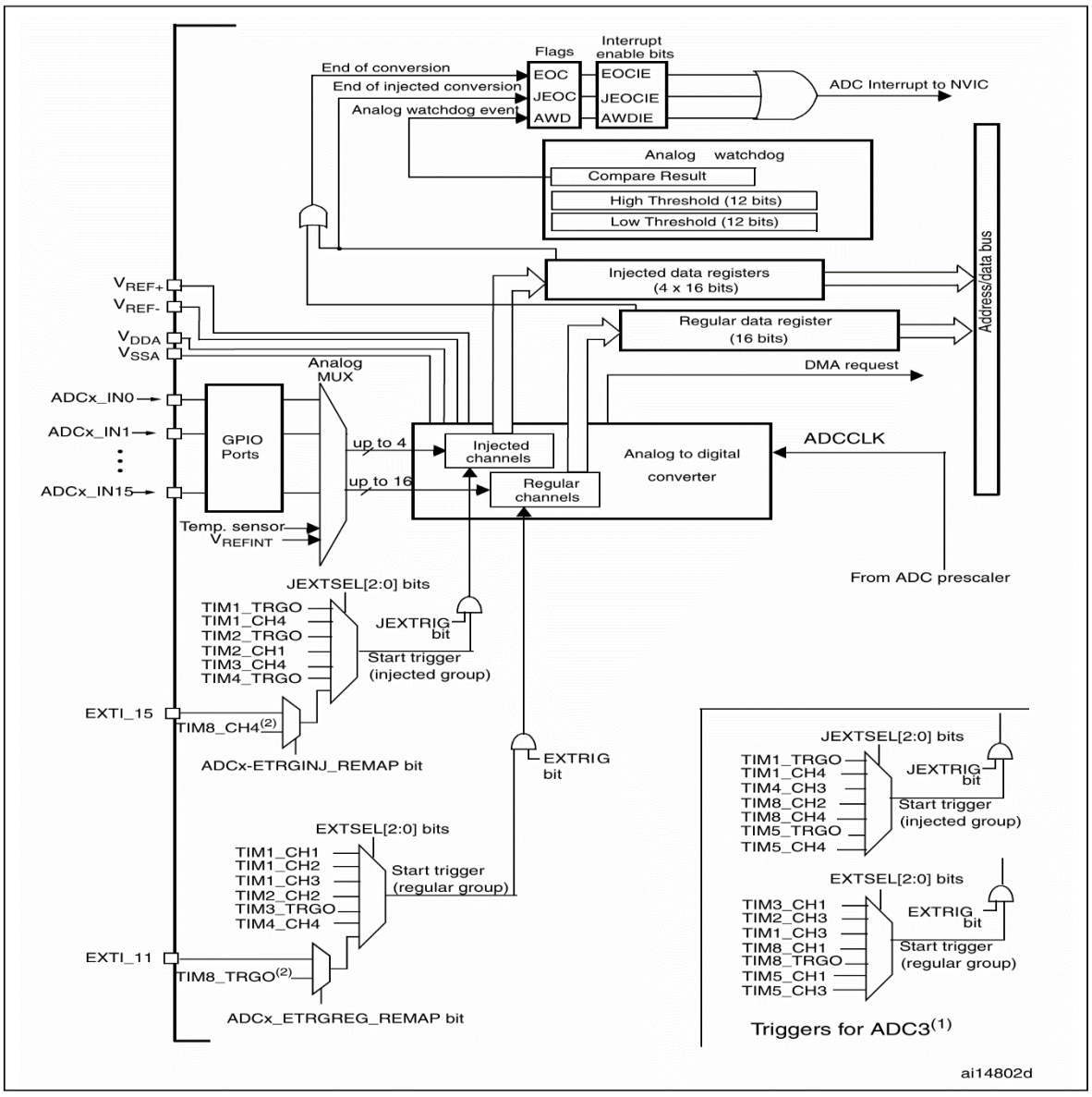


Figure 28 4.3.4.Single ADC block diagram

1. Temperature Monitoring

One of the primary applications of the ADC in our ADAS project is monitoring the cabin temperature. A temperature sensor, such as an NTC thermistor, is connected to the ADC input. The sensor produces an analog voltage proportional to the ambient temperature, which the ADC converts into a digital value for processing. This temperature data is crucial for maintaining a comfortable environment within the vehicle and for triggering alerts if the temperature exceeds safe limits.

Example Configuration:

Sensor	ADC Channel	Function
Temperature Sensor	ADC_CHANNEL_1	Monitor cabin temperature

The following code snippet demonstrates the configuration for reading temperature data:

```
ADC_HandleTypeDef hadc1;

ADC_ChannelConfTypeDef sConfig;

void ADC_Init(void) {
    __HAL_RCC_ADC1_CLK_ENABLE();
    hadc1.Instance = ADC1;
    hadc1.Init.ScanConvMode = ADC_SCAN_ENABLE;
    hadc1.Init.ContinuousConvMode = DISABLE;
    hadc1.Init.DiscontinuousConvMode = DISABLE;
    hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc1.Init.NbrOfConversion = 1;
    HAL_ADC_Init(&hadc1);
    sConfig.Channel = ADC_CHANNEL_1;
    sConfig.Rank = ADC_REGULAR_RANK_1;
    sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
    HAL_ADC_ConfigChannel(&hadc1, &sConfig);
}

void Read_Temperature(void) {
    HAL_ADC_Start(&hadc1);
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
    uint32_t tempValue = HAL_ADC_GetValue(&hadc1);
    // Convert tempValue to temperature in Celsius
}
```

2. Humidity Monitoring

Similar to temperature monitoring, the ADC is used to read humidity levels within the vehicle. A humidity sensor, which generates an analog voltage corresponding to the relative humidity, is connected to another ADC channel.

The ADC digitizes this signal, enabling the system to track humidity levels and maintain optimal cabin conditions.

Example Configuration:

Sensor	ADC Channel	Function
Humidity Sensor	ADC_CHANNEL_2	Monitor cabin humidity levels

3. Battery Voltage Monitoring

The ADC is instrumental in monitoring the battery voltage of the vehicle. A voltage divider circuit is used to scale the battery voltage to a range suitable for the ADC input. The ADC then converts this analog voltage into a digital value, which is used to assess the battery's health and status. Monitoring battery voltage is essential for ensuring that the vehicle operates reliably and for providing early warnings of potential battery failures.

Example Configuration:

Function	ADC Channel	Description
Battery Voltage	ADC_CHANNEL_3	Monitor battery health and status

The following code snippet demonstrates the configuration for reading battery voltage:

```
void ADC_Init(void) {
    // (Initialization code as before)
    sConfig.Channel = ADC_CHANNEL_3;
    sConfig.Rank = ADC_REGULAR_RANK_2;
    sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
    HAL_ADC_ConfigChannel(&hadc1, &sConfig);
}
void Read_Battery_Voltage(void) {
    HAL_ADC_Start(&hadc1);
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
    uint32_t batteryValue = HAL_ADC_GetValue(&hadc1);
    // Convert batteryValue to actual voltage
}
```

4. Light Intensity Measurement

The ADC can also be used to measure ambient light intensity using sensors like photodiodes or light-dependent resistors (LDRs). This data can be utilized to adjust the brightness of the vehicle's display or control automatic lighting systems. The ADC's high resolution ensures that even subtle changes in light intensity are detected accurately.

Example Configuration:

Function	ADC Channel	Description
Light Intensity	ADC_CHANNEL_4	Adjust display brightness

5. Sensor Fusion

By combining data from multiple sensors, the ADC facilitates sensor fusion, enhancing the system's overall functionality. For example, temperature and humidity data can be analyzed together to provide a comprehensive view of the cabin's environmental conditions. Similarly, integrating battery voltage and light intensity measurements can optimize power management and lighting control.

Integration and Processing

The ADC values are typically processed by the main microcontroller unit (MCU), the STM32F103C6 in our case. The MCU runs algorithms that interpret these digital values and take appropriate actions, such as adjusting the HVAC system based on temperature and humidity readings or triggering alerts if the battery voltage drops below a critical threshold. The processed data is also displayed on the user interface, providing real-time feedback to the driver.

Example Flow for Temperature Monitoring:

Sensor Reading: The temperature sensor generates an analog voltage.

Analog-to-Digital Conversion: The ADC converts the analog voltage to a digital value.

Data Processing: The MCU processes the digital value to derive the actual temperature.

Action: Based on the temperature, the system may adjust the HVAC settings or trigger an alert.

4.3.5. CAN

The Controller Area Network (CAN) is a robust vehicle bus standard designed to facilitate communication among microcontrollers and devices without a host computer. Developed by Bosch in the mid-1980s, CAN has become a critical technology in automotive and industrial applications due to its high reliability and efficiency in data transmission. CAN operates by broadcasting messages to all nodes on the network. Each message contains an identifier that denotes its priority, ensuring that critical messages are transmitted with minimal delay. This arbitration process, based on the identifier, allows for efficient and collision-free communication, even in networks with high traffic loads. One of the key features of CAN is its error detection and handling capabilities. The protocol includes several mechanisms to detect errors, such as cyclic redundancy checks (CRC), bit stuffing, and acknowledgment checks. When an error is detected, the network can automatically retransmit the affected message, ensuring data integrity. This robust error management makes CAN highly reliable in noisy environments, which is a common scenario in automotive and industrial settings. The physical layer of CAN is designed to be highly resilient. It supports differential signaling, which helps to minimize the impact of electromagnetic interference (EMI). Additionally, the use of termination resistors at each end of the bus helps to prevent signal reflections, further enhancing signal integrity.

In modern automotive systems, CAN is used to connect various electronic control units (ECUs), such as the engine control unit, transmission control unit, antilock braking system (ABS), airbags, and infotainment systems. This interconnected network allows for coordinated control and data sharing among the different subsystems, improving overall vehicle performance and safety.The evolution of CAN technology has also led to the development of new standards, such as CAN FD (Flexible Data-rate), which offers higher data rates and larger data payloads compared to classical CAN. This allows for more efficient communication, particularly in systems that require the transmission of large amounts of data.

4.3.5.1. CAN Protocol Basic

At its core, the CAN protocol is based on message-oriented communication. Unlike point-to-point communication, where devices communicate directly with each other, CAN uses a broadcast method. Every node on the network receives all messages, but each node

decides independently whether to process a particular message based on its identifier.

A CAN message consists of several fields, each serving a specific purpose:

Start of Frame (SOF): This is a single dominant bit that marks the beginning of a frame, ensuring synchronization of all nodes on the network.

Identifier: This field uniquely identifies the message and determines its priority. Lower values have higher priority, allowing critical messages to be transmitted first.

Control Field: This includes the Data Length Code (DLC), which specifies the number of bytes of data contained in the message.

Data Field: This field can contain up to 8 bytes of data. In CAN FD (Flexible Data-rate), the data field can be extended to accommodate up to 64 bytes.

CRC Field: The Cyclic Redundancy Check field is used for error detection. It ensures that the data has not been corrupted during transmission.

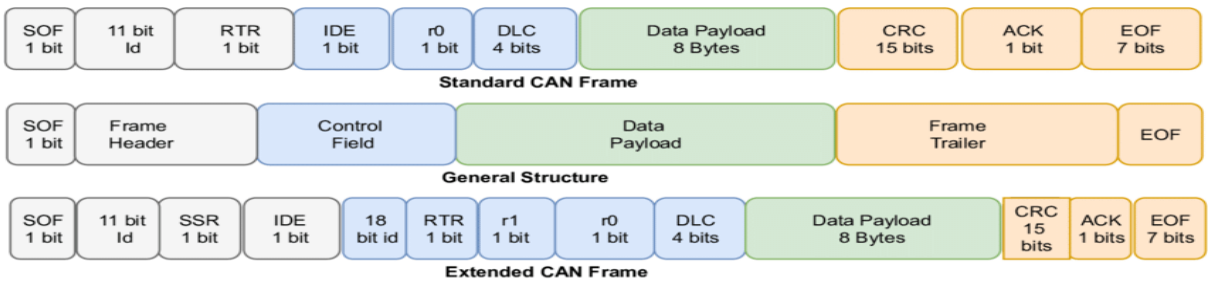
ACK Field: After receiving a message, a receiving node sends an acknowledgment bit to indicate successful reception.

End of Frame (EOF): This marks the end of the message.

Interframe Space: A short period of bus idle time that separates consecutive frames.

One of the defining features of CAN is its use of non-destructive bitwise arbitration. During arbitration, if two nodes start transmitting simultaneously, the node with the higher priority message (lower identifier) continues to transmit, while the other node stops. This ensures that the highest priority message is always transmitted first, without any collisions.

Figure 29 4.3.5.1. CAN Message Frame Structure



Error detection and handling are critical components of the CAN protocol. The protocol employs several methods to detect errors:

Bit Monitoring: Each node monitors the bits on the bus and compares them with the bits it sent. A discrepancy indicates an error.

Bit Stuffing: To ensure synchronization, if a sender detects five consecutive bits of the same polarity, it automatically inserts a complementary bit. The receiver removes this bit to recover the original message.

Frame Check: The structure of a CAN frame is predefined. Any deviation from this structure indicates an error.

CRC Check: The receiver calculates a CRC value from the received message and compares it with the transmitted CRC value. A mismatch indicates an error.

Acknowledgment Check: The transmitter expects an acknowledgment from the receiver. Absence of this acknowledgment signals an error.

When an error is detected, the protocol allows for several corrective actions. The most common response is for the node that detects the error to transmit an error frame, causing all nodes to discard the current message and prepare for a retransmission. This robust error-handling mechanism ensures data integrity and reliability

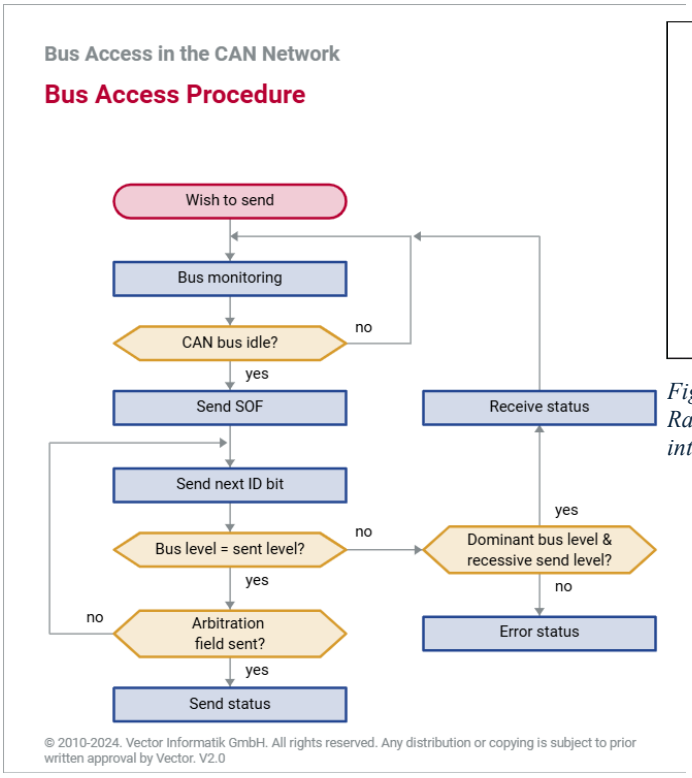


Figure 31 4.3.5.1. bus acess procedure

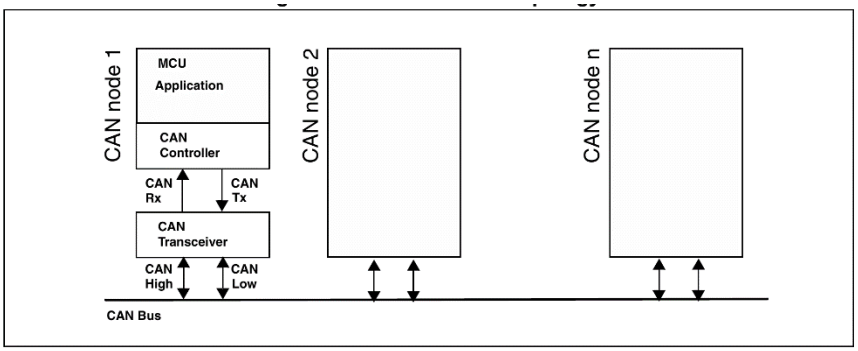
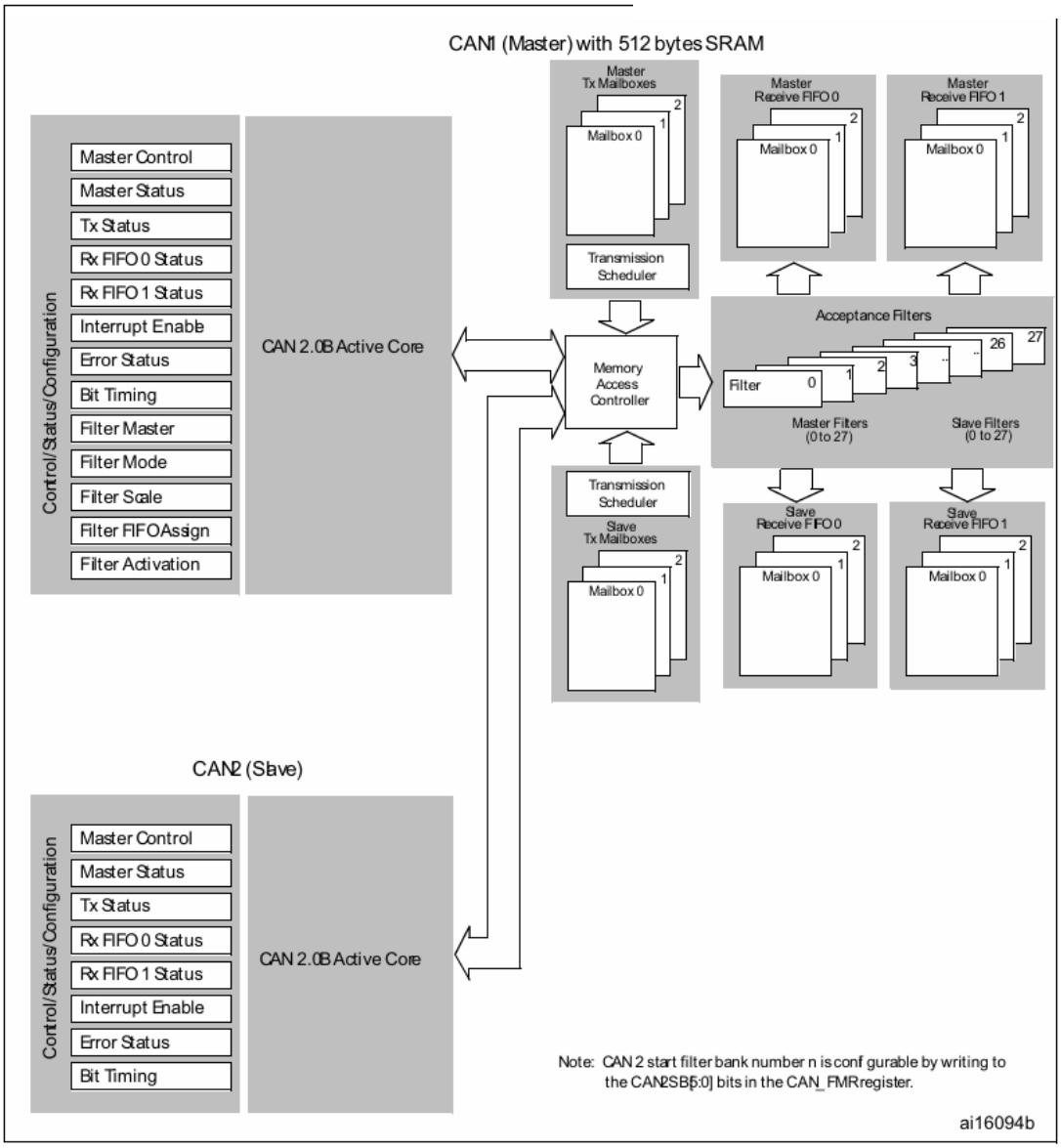


Figure 30 4.3.5.1. CAN Network Topology – Illustrates the connection layout between the Raspberry Pi and STM32F103C6 microcontrollers, highlighting the CAN bus interconnections

4.3.5.2. Implementing CAN in STM32F103C6

The STM32F103C6 is well-suited for this task due to its integrated CAN peripheral, which simplifies the process of setting up and managing CAN communication. This section provides a detailed guide on configuring and utilizing CAN in the STM32F103C6, ensuring that both academic and industrial learners can effectively apply these concepts

Figure 32 4.3.5.2. Dual CAN block diagram



4.3.5.2.1. Configuration of CAN in STM32F103C6

The process of configuring CAN in the STM32F103C6 involves several steps, including initializing the CAN peripheral, setting up the necessary parameters, and configuring the CAN filters. Below is a detailed explanation of these steps:

1. Initializing the CAN Peripheral:

To initialize the CAN peripheral, we need to enable the CAN clock and configure the CAN registers. This involves configuring the CAN timing parameters, setting up the CAN mode (normal mode, loopback mode, or silent mode), and initializing the CAN hardware.

// Enable CAN clock

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
```

// CAN initialization

```
CAN_InitTypeDef CAN_InitStructure;
CAN_StructInit(&CAN_InitStructure);
CAN_InitStructure.CAN_TTCM = DISABLE;
CAN_InitStructure.CAN_ABOM = ENABLE;
CAN_InitStructure.CAN_AWUM = ENABLE;
CAN_InitStructure.CAN_NART = DISABLE;
CAN_InitStructure.CAN_RFLM = DISABLE;
CAN_InitStructure.CAN_TXFP = ENABLE;
CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;
CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;
CAN_InitStructure.CAN_BS1 = CAN_BS1_6tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq;
CAN_InitStructure.CAN_Prescaler = 9;  // Adjust based on the desired baud rate
CAN_Init(CAN1, &CAN_InitStructure);
```

2. Configuring CAN Filters:

CAN filters are used to control which messages are received by the microcontroller. This is crucial for ensuring that the microcontroller only processes relevant messages, reducing unnecessary load.

// CAN filter initialization

```
CAN_FilterInitTypeDef CAN_FilterInitStructure;
CAN_FilterInitStructure.CAN_FilterNumber = 0;
CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdMask;
CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_32bit;
CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
```

```
CAN_FilterInitStructure.CAN_FilterMaskIdHigh = 0x0000;
CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
CAN_FilterInitStructure.CAN_FilterFIFOAssignment = CAN_Filter_FIFO0;
CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;
CAN_FilterInit(&CAN_FilterInitStructure);
```

3. Transmitting CAN Messages:

To transmit messages over the CAN bus, you need to configure the CAN Tx mailbox and send the message using the CAN_Transmit function.

// CAN transmission

```
CanTxMsg TxMessage;
TxMessage.StdId = 0x321;
TxMessage.ExtId = 0x01;
TxMessage.RTR = CAN_RTR_DATA;
TxMessage.IDE = CAN_ID_STD;
TxMessage.DLC = 2;
TxMessage.Data[0] = 0xDE;
TxMessage.Data[1] = 0xAD;
CAN_Transmit(CAN1, &TxMessage);
```

4. Receiving CAN Messages:

Receiving messages involves configuring the CAN Rx FIFO and handling the received messages using the CAN_Receive function.

// CAN reception

```
CanRxMsg RxMessage;
CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
```

// Process the received message

```
if (RxMessage.StdId == 0x321) {
    // Handle the message
}
```

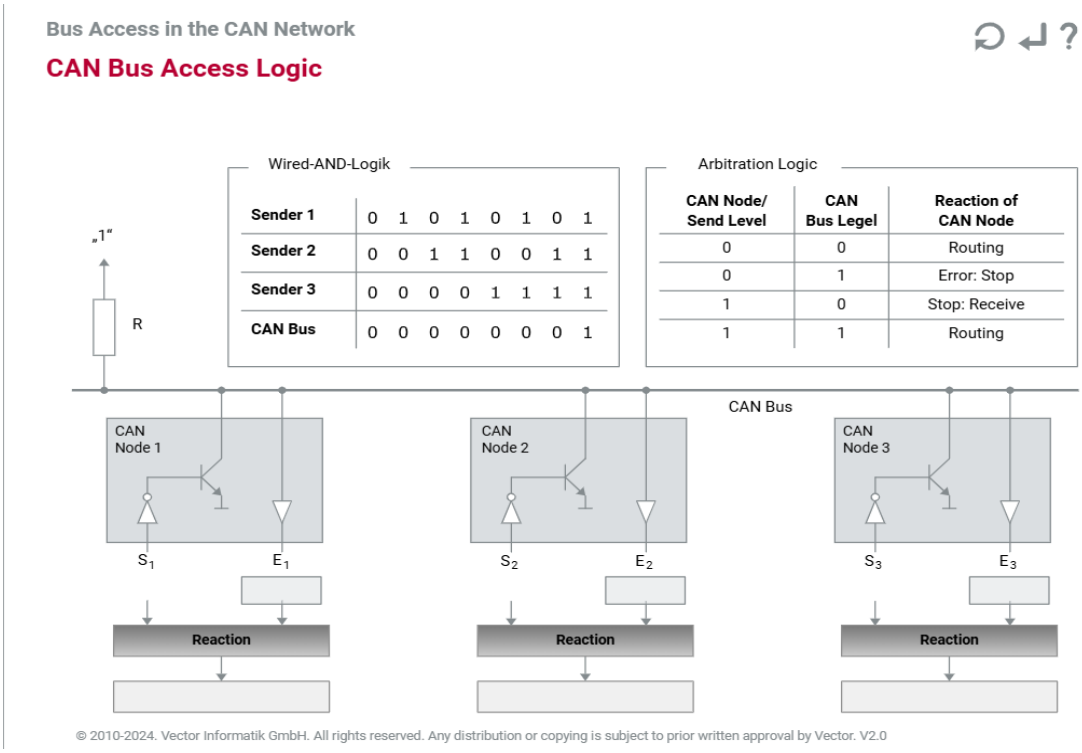


Figure 33 4.3.5.2.1. Arbitration Process – Depicts how nodes on the CAN bus prioritize message transmission based on identifiers, ensuring collision-free communication

4.3.5.3. Practical Applications of CAN in Our Project

This section explores the practical applications of CAN within our system, detailing how it facilitates seamless data exchange and control across the different components.

4.3.5.3.1. Coordinating Multiple Microcontrollers

In our ADAS, the Raspberry Pi acts as the central processing unit, managing high-level functions such as AI-based driver and child detection, user interface, and overall system coordination. Three STM32F103C6 microcontrollers handle specific tasks:

MCU1: Manages GSM communication for sending alerts and notifications.

MCU2: Handles GPS data for location tracking and navigation.

MCU3: Monitors environmental conditions using temperature and humidity sensors.

The CAN bus is the backbone of communication between these microcontrollers and the Raspberry Pi, enabling real-time data exchange and control commands.

4.3.5.3.2. Example Scenario: Child Safety Monitoring

Initial Setup:

The Raspberry Pi sends a configuration message to each MCU over the CAN bus during system initialization.

Each MCU acknowledges the configuration and sets up its respective sensors and communication modules.

Ongoing Monitoring:

MCU3 continuously monitors the temperature and humidity inside the vehicle. It periodically sends sensor data to the Raspberry Pi using CAN messages with a specific identifier (e.g., 0x300).

The Raspberry Pi processes this data to ensure that the environmental conditions are within safe limits for the child.

Alert Generation:

If the Raspberry Pi detects a temperature or humidity level outside the safe range, it sends an alert message to **MCU1** using CAN.

MCU1 uses the GSM module to send an alert notification to the designated contacts, including parents or emergency services.

4.3.5.3.3. Implementation Details

CAN Message Structure:

Each CAN message consists of an identifier, control field, data field, CRC, acknowledgment, and end of frame.

Message identifiers are used to prioritize and route messages appropriately across the network.

CAN Filters and Masks:

CAN filters are configured on each MCU to accept relevant messages and ignore others, reducing processing load and increasing efficiency.

For example, **MCU1** might be configured to accept messages with identifiers in the range of 0x100 to 0x1FF for GSM-related commands and data.

Error Handling and Retransmission:

The CAN protocol includes mechanisms for error detection and automatic retransmission, ensuring reliable communication even in the presence of transient faults or electromagnetic interference.

Our implementation leverages these features to maintain robust communication between the Raspberry Pi and microcontrollers.

To illustrate the CAN bus configuration in our project, consider the following example:

Table : CAN implemenation

Component	Function	CAN ID
Raspberry Pi	Central processing unit	0x100
STM32F103C6 MCU1	GSM communication	0x200
STM32F103C6 MCU2	GPS data management	0x300
STM32F103C6 MCU3	Environmental monitoring (temp/humidity)	0x400

In this setup, the Raspberry Pi sends a command with CAN ID 0x100 to MCU1 (GSM) to initiate a communication sequence. MCU1 responds with its status, using CAN ID 0x200. Similarly, MCU2 and MCU3 communicate their respective data using their designated CAN IDs. This structured approach ensures organized and efficient communication across the network.

4.3.6. UART

UART (Universal Asynchronous Receiver-Transmitter):

UART, or Universal Asynchronous Receiver-Transmitter, is a widely used communication protocol in embedded systems. It facilitates serial communication between two devices, typically a microcontroller and a peripheral, without requiring a clock signal for synchronization.

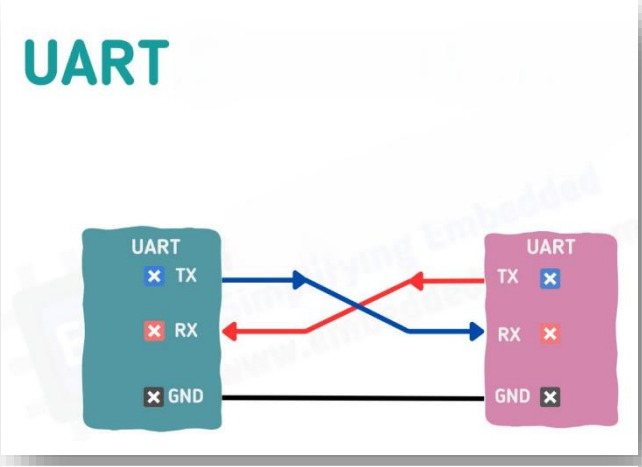


Figure 34 4.3.6. UART Protocol

Instead, UART relies on a shared baud rate, ensuring that both devices send and receive data at the same speed.

A UART system consists of two main components: the transmitter, which converts parallel data from the host device into serial form, and the receiver, which reassembles the serial data back into parallel form for the receiving device. Data transmission in UART occurs over two wires, usually labelled TX (transmit) and RX (receive). Each data packet includes a start bit, data bits, an optional parity bit for error checking, and one or more stop bits. And data transmitted in a specific baud rate is the number of bits transmitted in one second .

One of the key advantages of UART is its simplicity and ease of implementation. It requires minimal hardware resources and can operate effectively over short to medium distances. However, UART is limited by its speed and distance .

In automotive applications, UART is often used for diagnostic communication between various subsystems within a vehicle, enabling efficient data transfer without the need for extensive wiring. Its reliability and straightforward nature make UART an integral part of

many embedded systems, including those in advanced driver assistance systems (ADAS).

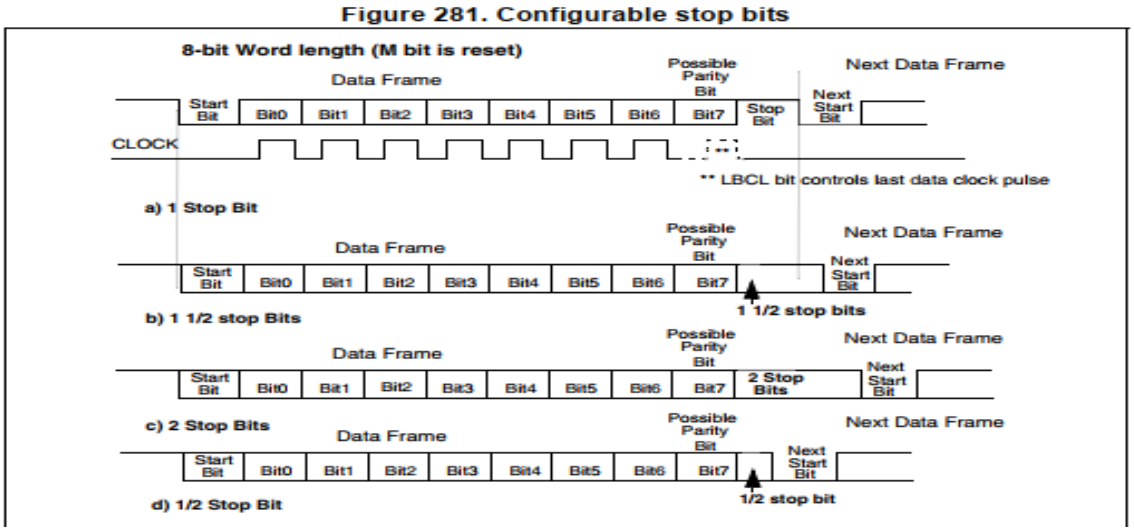


Figure 35 4.3.6. UART Configuration

Alternate function	USART1_REMAP = 0	USART1_REMAP = 1
USART1_TX	PA9	PB6
USART1_RX	PA10	PB7

USART pinout	Configuration	GPIO configuration
USARTx_TX(1)	Full duplex	Alternate function push-pull
	Half duplex synchronous mode	Alternate function push-pull
USARTx_RX	Full duplex	Input floating / Input pull-up
	Half duplex synchronous mode	Not used. Can be used as a general IO
USARTx_CK	Synchronous mode	Alternate function push-pull
USARTx_RTS	Hardware flow control	Alternate function push-pull
USARTx_CTS	Hardware flow control	Input floating/ Input pull-up

Table 52. USART3 remapping

Alternate function	USART3_REMAP[1:0] = "00" (no remap)	USART3_REMAP[1:0] = "01" (partial remap) (1)	USART3_REMAP[1:0] = "11" (full remap) (2)
USART3_TX	PB10	PC10	PD8
USART3_RX	PB11	PC11	PD9
USART3_CK	PB12	PC12	PD10
USART3_CTS	PB13		PD11
USART3_RTS	PB14		PD12

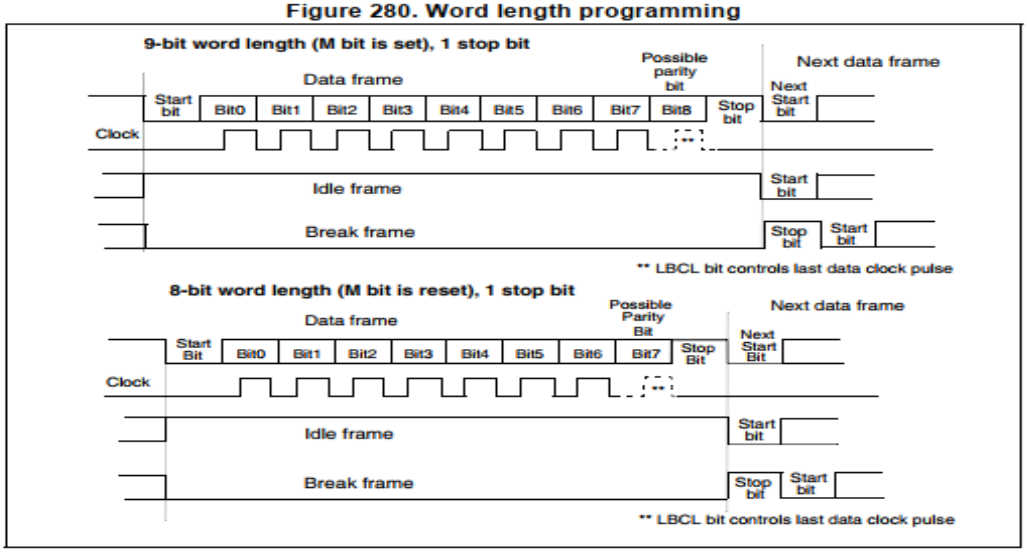
1. Remap available only for 64-pin, 100-pin and 144-pin packages
2. Remap available only for 100-pin and 144-pin packages.

Table 53. USART2 remapping

Alternate functions	USART2_REMAP = 0	USART2_REMAP = 1(1)
USART2_CTS	PA0	PD3
USART2_RTS	PA1	PD4
USART2_TX	PA2	PD5
USART2_RX	PA3	PD6
USART2_CK	PA4	PD7

1. Remap available only for 100-pin and 144-pin packages.

Figure 280. Word length programming



4.3.7. GSM

Figure 36 4.3.7. GSM Module

GSM, or Global System for Mobile Communications, is a standard developed to describe the protocols for second-generation (2G) digital cellular networks used by mobile phones. GSM has become the most widely adopted cellular standard, providing reliable and secure communication over long distances.

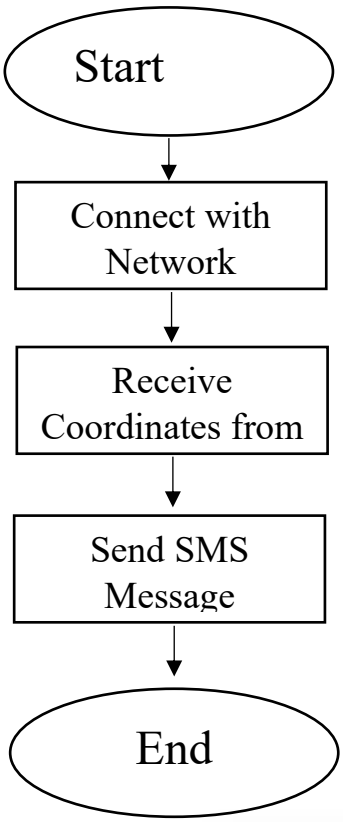
A GSM module in an embedded system, such as an ADAS, enables wireless communication, allowing the system to send and receive data over cellular networks. This capability is crucial for applications requiring remote monitoring and control. For instance, in an ADAS, a GSM module can transmit real-time alerts about driver fatigue or child detection to a remote server or directly to the driver's mobile phone.

GSM operates on various frequency bands, typically 900 MHz and 1800 MHz in Europe and 850 MHz and 1900 MHz in North America. It employs time division multiple access (TDMA) technology to optimize the use of available bandwidth, allowing multiple users to share the same channel.

Security is a significant feature of GSM, incorporating encryption and subscriber identity confidentiality to protect against eavesdropping and fraud. Additionally, GSM supports a variety of services beyond voice calls, including SMS (Short Message Service) for text messaging and GPRS (General Packet Radio Service) for data transmission.

The integration of GSM modules in automotive systems enhances connectivity and safety by enabling features such as emergency calling, real-time traffic updates, and remote diagnostics. As cellular technology evolves, GSM remains a foundational element, ensuring consistent and widespread communication capabilities.

Flow Of Control:



4.3.8. GPS

GPS, or Global Positioning System, is a satellite-based navigation system that provides geolocation and time information to a GPS receiver anywhere on or near the Earth's surface where there is an unobstructed line of sight to four or more GPS satellites. Developed by the United States Department of Defence, GPS has become an essential tool in navigation, timing, and various applications requiring precise location data.

Figure 37 4.3.8. GPS Module

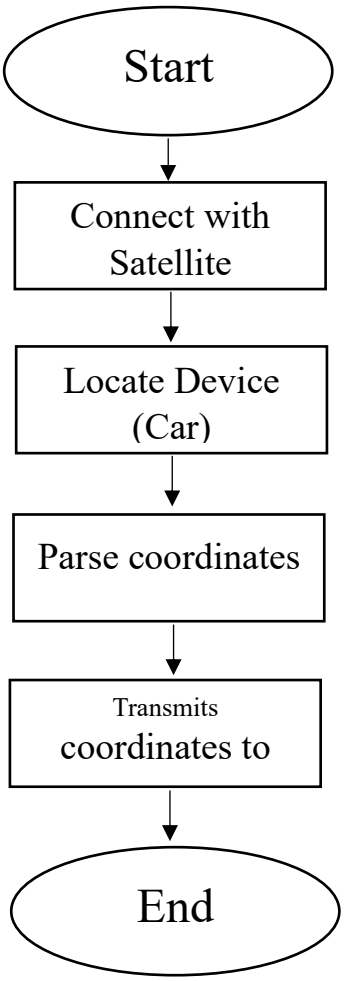
A GPS module in an embedded system receives signals from a network of satellites orbiting the Earth. By calculating the time it takes for the signals from multiple satellites to reach the receiver, the module can determine its exact location in terms of latitude,

longitude, and altitude. This information is crucial for navigation systems, enabling real-time tracking and route planning.

In automotive applications, GPS is integrated into ADAS to enhance driver assistance features. For instance, GPS data can be used for navigation, providing turn-by-turn directions and real-time traffic updates. It also plays a role in advanced safety systems, such as collision avoidance, by informing the system about the vehicle's precise location and movement.

GPS technology operates globally and is unaffected by weather conditions, making it highly reliable. It supports a range of applications from personal navigation devices and smartphone apps to fleet management and autonomous driving systems. The integration of GPS in ADAS not only improves navigation and safety but also enables innovative features such as geofencing, where a vehicle's movements are monitored to ensure it stays within designated boundaries.

Flow Of Control:



4.3.9. NVIC

The Nested Vectored Interrupt Controller (NVIC) is a crucial component in modern microcontroller architectures, particularly those based on ARM Cortex-M processors. The NVIC is designed to handle interrupt management, which is essential for real-time applications where the timely response to external events is critical.

Interrupts are signals that temporarily halt the main program execution to allow the microcontroller to respond to urgent tasks or events. The NVIC manages these interrupts efficiently, prioritizing them and ensuring that high-priority interrupts can pre-empt lower-priority ones. This capability is vital in embedded systems where multiple peripherals may need immediate attention simultaneously.

Prioritization and Nesting:

One of the key features of the NVIC is its ability to handle nested interrupts. Nesting allows a higher-priority interrupt to interrupt a lower-priority one that is currently being serviced. The NVIC supports a configurable priority scheme, enabling developers to assign different priority levels to different interrupts. This ensures that the most critical tasks are addressed first, maintaining the system's responsiveness and stability.

The NVIC uses a vector table to determine the address of the interrupt service routines (ISRs). Each entry in the vector table corresponds to a specific interrupt source, and when an interrupt occurs, the NVIC uses this table to jump to the appropriate ISR. This mechanism ensures that the correct response is executed for each specific interrupt.

The NVIC provides low-latency handling of interrupts. This is particularly important in applications like automotive safety systems, where rapid response to sensor inputs can mean the difference between avoiding a collision and experiencing one. The NVIC's quick interrupt handling ensures that critical events are processed with minimal delay.

The NVIC is an integral component in ARM Cortex-M microcontrollers, providing sophisticated interrupt management capabilities essential for real-time embedded systems. Its ability to prioritize, nest, and handle interrupts with low latency makes it particularly valuable in applications requiring rapid and reliable responses, such as Advanced Driver Assistance Systems. By ensuring that critical events are addressed

promptly and efficiently, the NVIC contributes significantly to the performance and safety of modern automotive systems.

4.3.10. BareMetal Layers Design

Design of MCAL, HAL, APP, and Services

4.3.10.1. MCAL

MCAL provides a standardized interface between the hardware and the application software, abstracting hardware-specific details and facilitating software portability and reusability. Key components include:

ADC :

- Function: Converts analog signals from sensors to digital values for processing.

- MCAL Role: Provides a hardware-independent interface to configure and read ADC values.

DMA (Direct Memory Access):

- Function: Allows peripherals to transfer data directly to/from memory without CPU intervention.

- MCAL Role: Manages DMA configurations and operations to ensure efficient data transfers.

GPIO (General-Purpose Input/Output):

- Function: Controls and monitors digital signals for various input and output operations.

- MCAL Role: Abstracts hardware-specific details, offering a uniform API for GPIO configuration and control.

NVIC (Nested Vectored Interrupt Controller):

- Function: Manages and prioritizes interrupts.

- MCAL Role: Provides interfaces to configure, enable, and handle interrupts, ensuring real-time responsiveness.

RCC (Reset and Clock Control):

- Function: Manages the clock sources and resets peripherals and the microcontroller.

- MCAL Role: Offers APIs to configure clock settings and control system resets.

UART (Universal Asynchronous Receiver-Transmitter):

- Function: Facilitates serial communication between devices.

- MCAL Role: Provides abstraction for UART configuration and data transfer operations.

4.3.10.2. HAL

HAL builds on MCAL, providing higher-level, device-independent APIs for application development, ensuring ease of use and faster development cycles.

- **ADC HAL**: Simplifies ADC setup and data acquisition processes.

-**DMA HAL**: Streamlines DMA channel configuration and data transfer handling.

-**GPIO HAL**: Offers user-friendly functions for setting up and controlling GPIO pins.

- **NVIC HAL**: Provides straightforward APIs for interrupt management and handling.

-**RCC HAL**: Simplifies clock configuration and system reset management.

-**UART HAL**: Eases the implementation of UART communication protocols.

4.3.10.3. APPLICATION

The Application Layer represents the highest level in the software stack, containing the user-defined logic and functionality tailored to specific application requirements.It leverages underlying hardware and abstraction layers to achieve desired outcomes. It contains :

1. User Interface Management:

- Handles user interactions through inputs (buttons, switches, touchscreens) and outputs (displays, LEDs).

2. Data Processing:

- Processes sensor data using filtering and signal processing algorithms.

3. Control Algorithms:

- Implements control logic, including feedback loops and PID controllers.

4. Communication Protocols:

- Facilitates data exchange using protocols like UART, SPI, I2C, Bluetooth, and Wi-Fi.

5. Configuration and Calibration:

- Manages system parameters and sensor calibration.

6. Application-Specific Logic:

- Custom routines tailored to specific application requirements.

4.3.10.3. SERVICES

Services offer essential functionalities and utilities supporting the entire software stack, enhancing system robustness and efficiency.

It contains :

System Initialization:

Manages hardware setup and configuration at startup.

Memory Management:

Handles dynamic memory allocation and buffer management.

Logging and Diagnostics:

Provides tools for performance monitoring and debugging.

Timing and Scheduling:

Manages task execution and prioritization using timers and schedulers.

Event Handling:

Manages events and interrupts with event queues and ISRs.

Power Management:

Optimizes power consumption through control of power modes and peripherals.

Security Services:

Ensures data integrity with encryption and authentication.

Configuration Management:

Manages system settings and updates.

Data Management:

Handles data storage, retrieval, and manipulation.

The Application Layer and Services are crucial for developing robust and scalable embedded systems. The APP layer delivers application-specific functionalities, while Services provide essential utilities supporting the entire stack. This modular architecture promotes reusability, scalability, and reliability in embedded system development.

4.3.10.5. Integration

Integrating MCAL, HAL, APP, and Services ensures a robust, scalable, and maintainable system. MCAL abstracts hardware specifics, HAL simplifies interaction with hardware, APP layer customizes application logic, and Services provide essential utilities, collectively enabling efficient and reliable system development. This layered architecture enhances portability, reusability, and scalability across different projects and platforms.

5. Chapter 5: Testing and Results

5.1. Introduction to Testing

Testing is a critical phase in the development lifecycle of embedded systems, ensuring that the final product meets its design specifications, performs reliably under various conditions, and provides a user-friendly experience. In the context of our Advanced Driver Assistance System (ADAS) project, testing plays a vital role in validating the system's functionality, performance, and safety features, which are crucial for real-world applications.

The primary goals of testing are to identify and rectify defects early in the development process, verify that individual components and integrated modules function correctly, and ensure that the overall system meets the required standards and specifications. Effective testing helps in mitigating risks, improving system reliability, and enhancing user satisfaction.

5.2. Unit Testing

Unit testing is the first level of testing performed on individual components or modules of the system. In our ADAS project, unit testing focuses on verifying the functionality of specific features such as the Analog-to-Digital Converter (ADC), General-Purpose Input/Output (GPIO), and Controller Area Network (CAN) communication. The primary objective of unit testing is to ensure that each module functions correctly in isolation. This helps to identify and fix defects at an early stage, reducing the risk of more complex issues arising during later stages of integration and system testing.

Example - Unit Testing of ADC: For the ADC module in our STM32F103C6 microcontroller, unit testing might involve the following steps:

Configure the ADC module for different resolution settings (e.g., 8-bit, 10-bit, 12-bit).

Provide a known analog input signal (e.g., 1.5V) and measure the digital output.

Compare the digital output with the expected value, based on the resolution setting.

Verify the linearity and accuracy of the ADC across its entire input range.

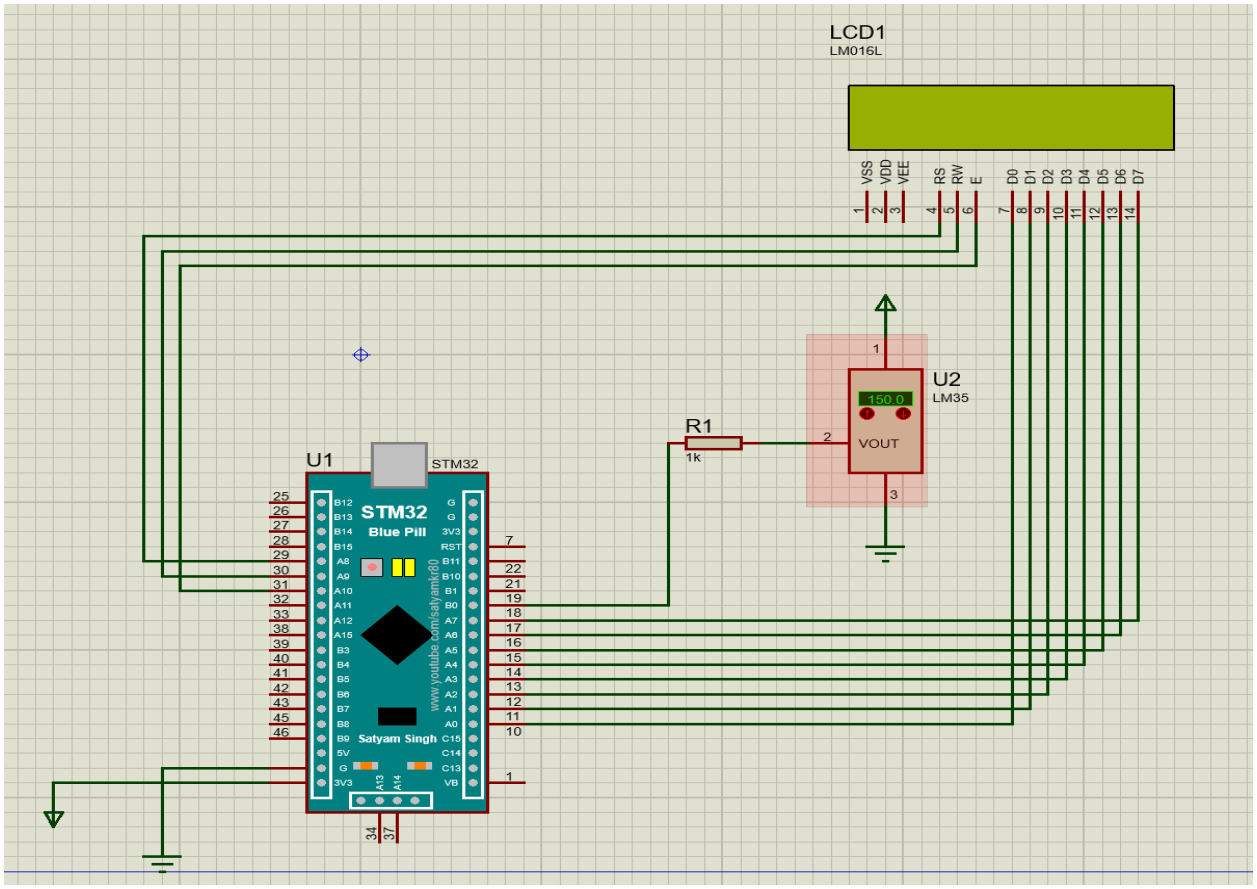


Figure 38 5.2. ADC testing simulator using protous:

5.3. Integration Testing

Integration testing is the phase where individual modules, which have been tested in isolation during unit testing, are combined and tested as a group. The objective of integration testing is to identify issues that arise when different modules interact with each other. This is especially important in complex systems like our Advanced Driver Assistance System (ADAS), where multiple components need to work seamlessly together. Integration testing typically follows two main approaches: top-down and bottom-up testing. In top-down integration testing, higher-level modules are tested first, followed by lower-level modules. This approach is beneficial for verifying the system's architecture early on. Bottom-up integration testing, on the other hand, starts with lower-level modules and progresses to higher-level ones, which is useful for validating the functionality of foundational components before moving to more complex interactions.

Example - Integration Testing of CAN Communication: For our ADAS project, integration testing of the CAN communication involves verifying the interaction between the Raspberry Pi and the STM32F103C6 microcontrollers. Test cases might include:

Sending a message from the Raspberry Pi to a microcontroller and verifying that the message is received and processed correctly.

Simulating a sensor input on one microcontroller, transmitting the data over the CAN bus, and verifying that the Raspberry Pi receives and logs the data correctly.

Inducing communication errors or data corruption and verifying that the system handles these situations gracefully, perhaps by retrying the communication or logging an error.

5.4. System Testing

System testing is the final phase of testing where the fully integrated ADAS system is tested as a complete unit. The objective of system testing is to ensure that the entire system functions correctly and meets all specified requirements under real-world conditions. System testing involves a comprehensive evaluation of the system's functionality, performance, security, and user experience. This phase typically includes a variety of test types, such as functional testing, performance testing, stress testing, and user acceptance testing (UAT).

Functional testing verifies that the system performs all its intended functions correctly. For the ADAS project, this includes testing features like driver and child detection, communication over the CAN bus, and interaction with the GSM and GPS modules. Test cases are derived from the system's functional requirements and cover all possible use cases and scenarios.

Performance testing assesses how the system performs under various conditions. This includes measuring response times, throughput, and resource utilization. For instance, performance testing of the ADAS might involve simulating high traffic on the CAN bus and measuring how quickly and accurately the system processes and responds to messages.

Stress testing evaluates the system's behavior under extreme conditions, such as high load, limited resources, or adverse environmental conditions. For our project, this might involve subjecting the system to high temperatures or simulating communication failures to see how the system copes and recovers.

User acceptance testing involves real users testing the system to ensure it meets their needs and expectations. This phase is crucial for validating the system's usability and ensuring that it delivers a positive user experience. For the ADAS, UAT might involve having drivers and vehicle owners test the

system in real driving conditions and providing feedback on its performance and usability.

Example - System Testing of Driver and Child Detection: For our ADAS project, system testing of the driver and child detection features involves:

Simulating the presence of a driver and a child in various positions and verifying that the system accurately detects and alerts for each scenario.

Testing the system's response to different lighting conditions, ensuring that it performs well both during the day and at night.

Measuring the system's response time from detecting the presence of a child to triggering an alert, ensuring that it meets the required performance standards.

5.5. Testing Tools and Environments

Testing tools and environments are essential for ensuring the robustness, reliability, and performance of our Advanced Driver Assistance System (ADAS). They facilitate the execution of comprehensive test cases, simulate various scenarios, and provide insights into the system's behavior under different conditions.

Objectives of Using Testing Tools and Environments: The primary goal of employing testing tools and environments is to streamline the testing process, enhance test coverage, and ensure that the ADAS system meets all functional and non-functional requirements. These tools help automate repetitive tasks, simulate real-world conditions, and provide detailed reports on system performance and reliability.

Key Testing Tools: Several testing tools are utilized in the development and testing of our ADAS project. These tools range from those that automate unit and integration tests to those that simulate complex driving environments and monitor system performance.

Unit Testing Tools:

Ceedling: A unit testing framework for C, commonly used for testing embedded systems. Ceedling helps automate the execution of unit tests, making it easier to verify individual functions and modules in isolation.

Unity: Another popular unit testing framework for C, often used in conjunction with Ceedling. Unity provides a simple interface for writing and running unit tests.

Integration Testing Tools:

CANoe: A comprehensive tool for developing, testing, and analyzing entire automotive networks and individual ECUs. CANoe is particularly useful for simulating CAN bus communication and testing the interaction between different components in the ADAS.

VectorCAST: A tool for automated integration and system testing of embedded systems. VectorCAST supports various testing methodologies and integrates with continuous integration (CI) systems to ensure regular and thorough testing.

System Testing Tools:

Jenkins: A CI tool that automates the process of building, testing, and deploying code changes. Jenkins helps ensure that tests are run consistently and that any issues are identified and addressed promptly.

Simulink: A simulation tool used to model and simulate dynamic systems. Simulink can be used to create virtual driving environments and test the ADAS system under various conditions.

Testing Environments: Creating an effective testing environment is crucial for ensuring that tests accurately reflect real-world conditions. The testing environment includes both hardware and software components necessary to execute and monitor tests.

Hardware Testing Environment:

Development Boards: STM32F103C6 microcontroller development boards are used to test the ADAS system's hardware components. These boards allow developers to run tests on the actual hardware, ensuring that the system performs as expected.

Sensors and Actuators: Various sensors and actuators are integrated into the testing environment to simulate real-world conditions. For example, cameras, temperature sensors, and GPS modules are used to test the ADAS system's functionality.

Software Testing Environment:

Virtual Machines: Virtual machines are used to create isolated testing environments that replicate the production environment. This helps ensure that tests are executed consistently and that results are reliable.

Simulation Software: Tools like Simulink and MATLAB are used to create virtual driving environments and simulate various scenarios. This allows developers to test the ADAS system's behavior under different conditions without needing access to a physical vehicle.

Example - Testing CAN Communication: For our ADAS project, testing the CAN communication involves setting up a testing environment that includes both hardware and software components. Using CANoe, we can simulate CAN messages and monitor the interaction between the Raspberry Pi and STM32F103C6 microcontrollers. This helps ensure that the system can accurately transmit and receive data over the CAN bus, even under different conditions.

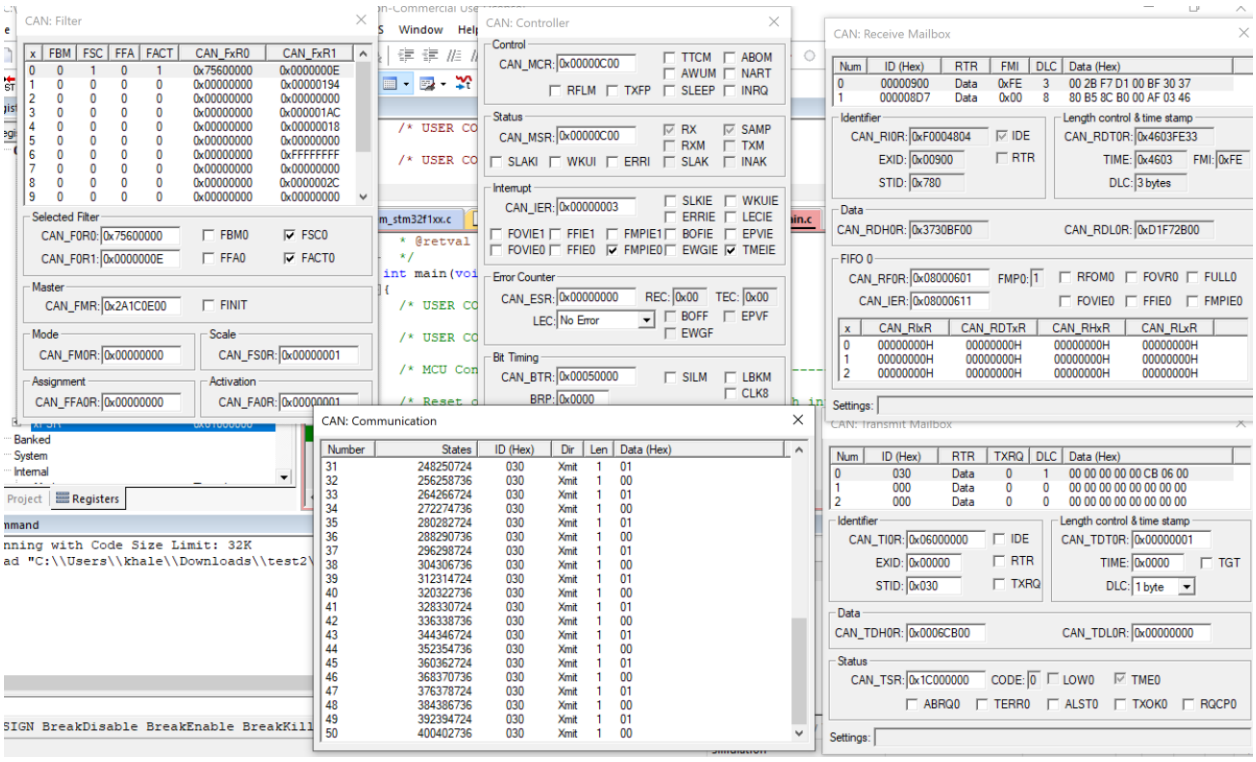


Figure 39 5.5. CAN testing simulator using KEIL

5.6. Performance Testing

Performance testing is a critical phase in the testing process, where the ADAS system's responsiveness, stability, scalability, and resource usage are evaluated under various conditions. The primary objective of performance testing is to ensure that the system meets the required performance criteria and can handle real-world scenarios effectivel. Performance testing involves several types of tests, each designed to evaluate different aspects of the

system's performance. These tests include load testing, stress testing, endurance testing, and spike testing.

Load Testing: Load testing assesses the system's performance under normal and peak load conditions. For the ADAS project, this involves simulating various driving scenarios with different numbers of sensors and actuators active simultaneously. The goal is to ensure that the system can handle expected traffic levels without performance degradation.

Stress Testing: Stress testing evaluates the system's behavior under extreme conditions, such as high load or limited resources. This helps identify the system's breaking point and ensures that it can recover gracefully from failures. In the ADAS project, stress testing might involve simulating a high number of CAN messages or artificially limiting system resources to test its robustness.

Endurance Testing: Endurance testing, also known as soak testing, assesses the system's performance over an extended period. This helps identify issues related to resource leaks, memory consumption, and long-term stability. For the ADAS project, endurance testing involves running the system continuously for several days to ensure it remains stable and performs well over time.

Spike Testing: Spike testing evaluates the system's response to sudden increases in load. This helps ensure that the system can handle unexpected spikes in traffic without crashing or becoming unresponsive. In the ADAS project, spike testing might involve simulating a sudden influx of CAN messages or rapidly changing sensor data.

Tools and Techniques: Several tools and techniques are used to facilitate performance testing. These tools help simulate load, monitor system performance, and analyze test results.

Load Testing Tools:

Apache JMeter: An open-source load testing tool that can simulate multiple users and monitor system performance. JMeter can be used to test the ADAS system's response to various load conditions.

LoadRunner: A comprehensive performance testing tool that supports load, stress, and endurance testing. LoadRunner provides detailed performance metrics and helps identify bottlenecks in the system.

Monitoring Tools:

Prometheus: An open-source monitoring tool that collects and analyzes performance metrics. Prometheus can be used to monitor the ADAS system's resource utilization, response times, and stability during performance testing.

Grafana: A visualization tool that works with Prometheus to create detailed dashboards and reports. Grafana helps visualize performance data and identify trends and issues.

Example - Performance Testing of Driver and Child Detection: For our ADAS project, performance testing of the driver and child detection features involves:

Load Testing: Simulating different numbers of sensors and actuators to ensure the system can handle the expected load without performance degradation.

Stress Testing: Simulating a high number of CAN messages or limiting system resources to test the system's robustness.

Endurance Testing: Running the system continuously for several days to ensure long-term stability and performance.

Spike Testing: Simulating sudden increases in sensor data to test the system's response to unexpected load spikes.

5.7. Evaluation of Driver Monitoring Model

Throughout the training, we validated and tested the model using various metrics such as accuracy, precision, recall, and F1-score. We iteratively adjusted hyperparameters and augmented the training dataset to improve performance

5.7.1. Validation Metrics

The model achieved a high mean average precision (mAP) of 97.1%, indicating excellent performance in correctly identifying the classes within the dataset.

The precision of the model was 95.1%, meaning that out of all the positive predictions made by the model, 95.1% were correct.

The recall was also 95.1%, indicating that the model was able to identify 95.1% of all actual positive instances in the dataset.

These high metrics demonstrate that our model is both accurate in its predictions and reliable in identifying true positives.

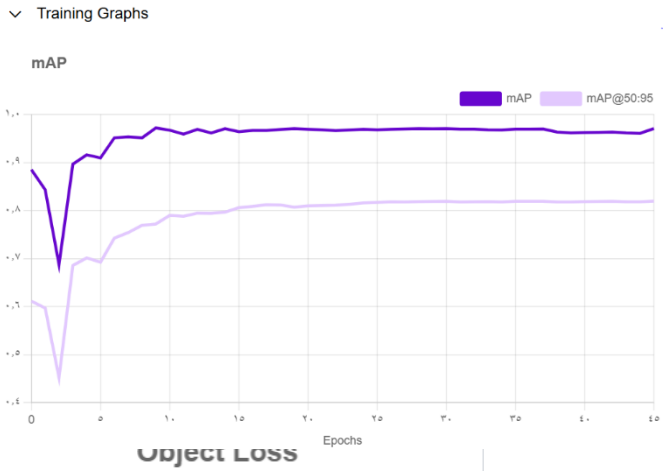


Figure 40 5.7.1.showing the progression of mean average precision (mAP) across epochs.

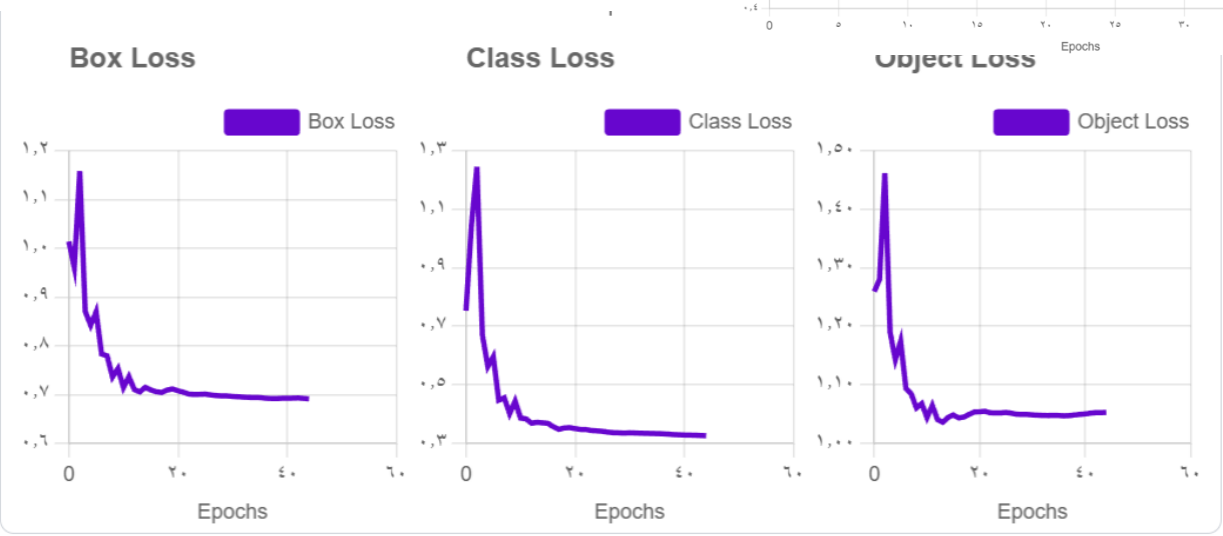


Figure 41 5.7.1. Graphs illustrating the progression of box loss (localization loss), class loss (classification loss), and object loss (confidence loss) across training epochs.4.7 Deployment

5.7.2. Confusion Matrix

The confusion matrix for the test images with a confidence threshold of 0.6 reveals a recall of 96% and a precision of 97%. This indicates that the model effectively identified 96% of all actual positives in the dataset while maintaining a high precision rate of 97%, reflecting its robust capability in accurately detecting and classifying driver behaviors .

Increasing the confidence threshold may lead to fewer false positives, potentially increasing precision further by ensuring that predictions are more confident and accurate. However, this adjustment could also decrease recall slightly, as the model may become more selective in identifying positive instances, potentially missing some lower-confidence predictions."

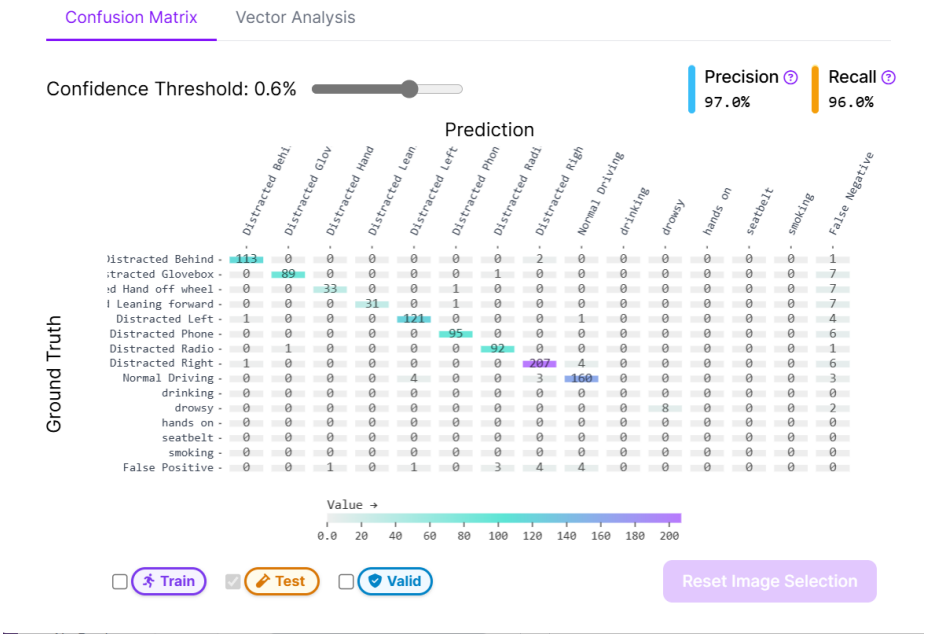


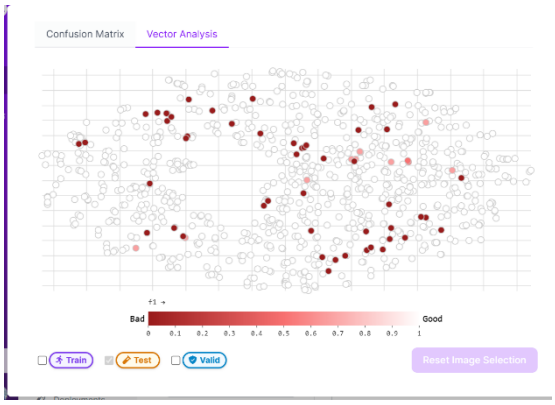
Figure 42 5.7.2. showing Confusion Matrix for the Test Set

5.7.3. F1 Score

This scatter plot shows F1 scores for all the images in this version plotted by 2D projected semantic embeddings

Each dot represents one image in the version. Semantically similar images will be close to each other on the graph. The nodes are colored by their f1 score (a measure taking both accuracy and recall into consideration). This lets you see which images performed well when comparing their ground truth labels to the ones predicted by the model and see if there are certain types of images that may be performing better than others.

Fig 4.6.13 showing Vector Analysis



5.7.4. Average Precision by Class for the Test Set

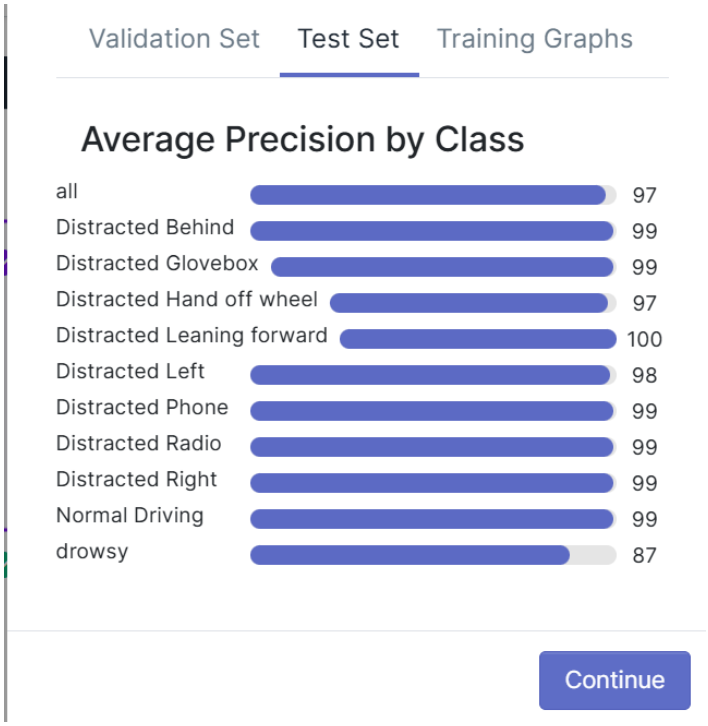
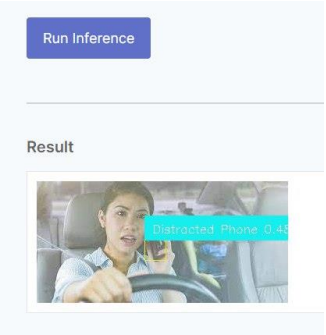


Figure 43 5.7.4. Average Precision by class

5.7.5. Testing on External Images

The following images illustrate the model's ability to accurately detect and classify driver behaviors .



Description: The model correctly identifies a distracted driver looking at a mobile phone, even though the image isn't very clear.

Description: The model identifies a driver using a mobile phone ,and wearing the seat belt .

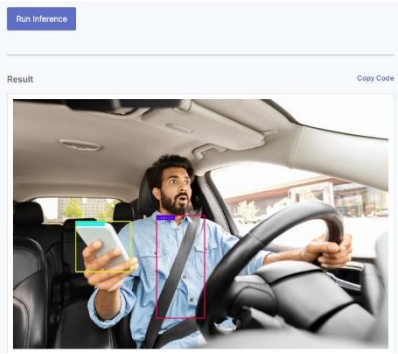
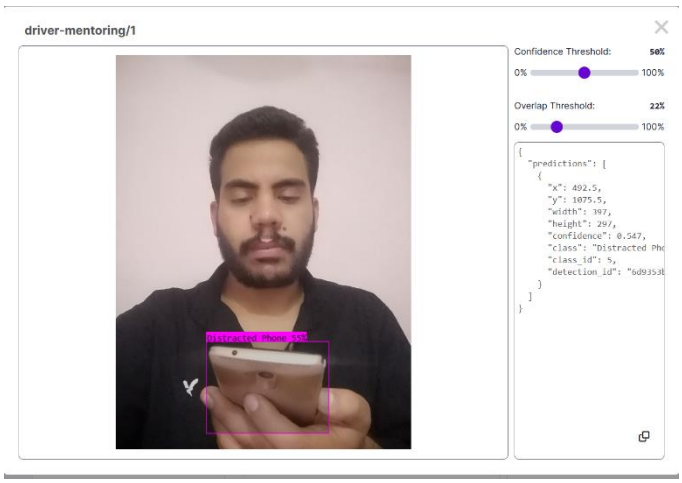
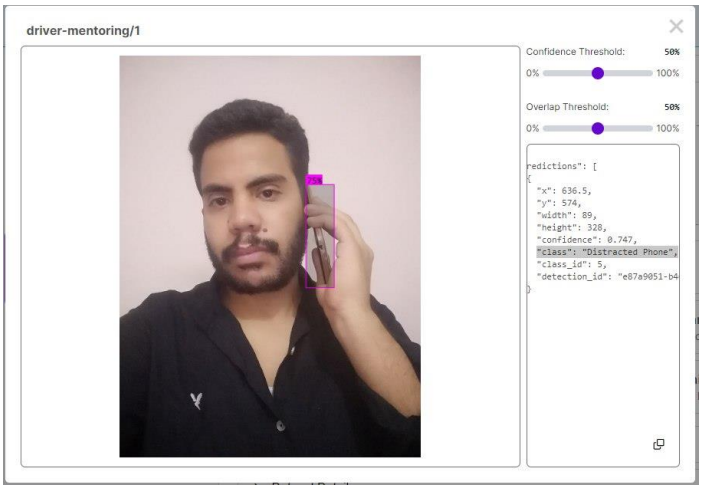


Figure 44 5.7.5. Testing on external image

Description: The model identifies a member of the project using a mobile phone with confidence score 75%.





5.8. Evaluation of the Child Detection Model

The evaluation of the child detection model was carried out using various metrics such as mean Average Precision (mAP), precision, and recall. Below are the insights derived from the evaluation phase:

mAP: The model achieved a mean Average Precision (mAP) of 88.7%. This indicates a high level of accuracy in detecting and localizing the classes within the dataset.

Precision: With a precision rate of 89.2%, the model demonstrated its capability to accurately identify true positives while minimizing false positives.

Recall: The recall rate stood at 83.0%, showing the model's effectiveness in capturing most of the actual positives within the dataset.

5.8.1. Training Graphs Analysis

1. **mAP over Epochs**:

   The top graph shows the progression of the mAP and mAP@50:95 over the training epochs.

   The mAP stabilized at around 88.7%, demonstrating consistent performance throughout the training.

2. **Box Loss vs. Epochs**:

   The first graph below the mAP graph shows the box loss over the epochs.

A consistent decrease in box loss indicates that the model improved in localizing the bounding boxes over time.

3. **Class Loss vs. Epochs**:

   The second graph depicts the class loss over the epochs.

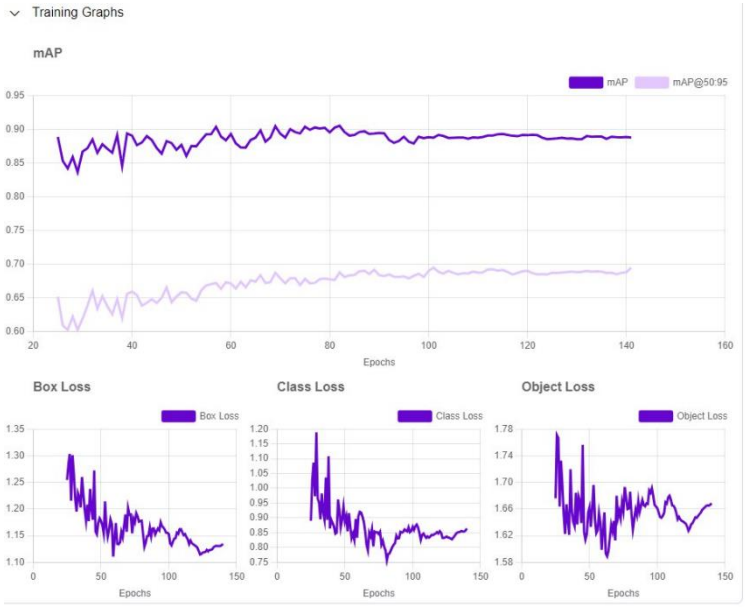
   The decrease in class loss signifies that the model became better at classifying the objects correctly.

4. **Object Loss vs. Epochs**:

   The third graph shows the object loss over the epochs.

   The object loss reduction reflects the model's improved confidence in object detection.

   These graphs and metrics collectively demonstrate the model's robust performance and reliability in detecting child presence accurately under various conditions. The comprehensive evaluation indicates that the model is well-prepared for real-world application in child detection scenarios.



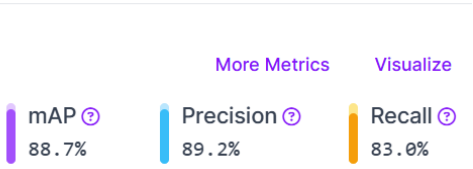


Figure 45 5.8.1. Graphs illustrating the progression of box loss (localization loss), class loss (classification loss), and object loss (confidence loss) across training epochs.

5.8.2. Real-Time Testing of the Child Detection Model

We tested the child detection model in real-time scenarios under different lighting conditions, including daylight and nighttime. The model performed admirably, accurately detecting the presence of a child in both environments.

These results illustrate the robustness and versatility of the child detection model, confirming its effectiveness in diverse real-world settings. The images below showcase the model's performance in these conditions:

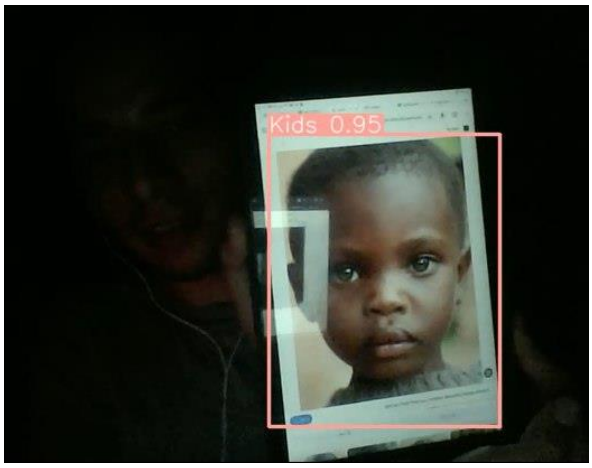
1. **Daylight Detection**
2. **Nighttime Detection**

5.8.2.1. Daylight Testing

Image 1: In daylight, the model successfully identified the child with high accuracy despite varying lighting and background conditions.

5.8.2.2. Nighttime Testing

- **Image 2**: During nighttime, the model continued to perform well, correctly detecting the child even under low-light conditions.



5.9. Troubleshooting and Debugging

Troubleshooting and debugging are critical components of the software development process, particularly for complex systems like our Advanced Driver Assistance System (ADAS). Ensuring the system functions correctly under all conditions requires a systematic approach to identifying, analyzing, and resolving issues that arise during development and testing. Troubleshooting and debugging involve a combination of methods and tools designed to pinpoint and rectify issues. The process typically starts with problem identification, followed by a detailed analysis to understand the underlying cause, and finally, implementing a solution and verifying its effectiveness.

The first step in troubleshooting is accurately identifying the problem. This can involve observing system behavior, reviewing test results, and gathering relevant data. In the case of our ADAS project, problems might manifest as unexpected sensor readings, communication errors between components, or system crashes. Detailed logging and monitoring tools are invaluable at this stage, providing insights into system performance and helping to isolate the problem's source.

Once the problem is identified, the next step is to analyze and diagnose the root cause. This involves examining the system's code, configuration, and interactions to understand what went wrong. Techniques such as code reviews, static analysis, and simulation can help uncover issues. For example, in our project, analyzing the CAN bus communication logs can reveal discrepancies or data transmission errors that might be causing system malfunctions.

After diagnosing the problem, the focus shifts to implementing a solution. This might involve modifying the code, updating configurations, or adjusting system parameters. It's crucial to test the proposed solution thoroughly to ensure it resolves the issue without introducing new problems. In our ADAS project, this could mean refining the algorithms for sensor data processing or enhancing error-handling routines for CAN communication.

The final step is to verify and validate the solution. This involves re-running tests to confirm that the problem has been resolved and that the system performs as expected. Continuous integration and automated testing frameworks can be particularly useful here, enabling repeated testing under consistent conditions. For our ADAS system, this might include re-executing

unit, integration, and system tests to ensure all components function correctly together.

In our project, suppose we encounter an issue where the STM32F103C6 microcontroller intermittently fails to receive messages over the CAN bus. The troubleshooting process would start with logging CAN messages and monitoring communication flow. If logs indicate missing or corrupted messages, the next step would be to analyze the microcontroller's CAN configuration and code handling message reception. Possible solutions could involve correcting timing settings, enhancing error-checking mechanisms, or adjusting interrupt priorities. After implementing changes, rigorous testing would ensure the issue is resolved and that communication is reliable

5.10. System Result

The model's ability to maintain high detection accuracy across different times of the day demonstrates its potential for reliable use in various practical applications, ensuring safety and monitoring in real-world environments.

GitHub Repository Link:

mohamedAhmedSalama/Graduation-Project2024 (github.com)

Project Video Link:

https://drive.google.com/drive/folders/1QoZtAyir4akFETfp_iMzApMb55lHLIyp

6. Chapter 6: Conclusion and future work

6.1. Conclusion

In this graduation project, we successfully integrated various technologies to create a comprehensive in-car monitoring system utilizing a Raspberry Pi, Raspberry Pi touchscreen, camera, and multiple Electronic Control Units (ECUs) communicating via UART. The primary objective was to enhance vehicle safety through real-time data visualization and advanced computer vision techniques.

The implementation involved developing two critical computer vision models: one for detecting a forgotten child in the car and the other for driver monitoring. The forgotten child detection model addresses the pressing issue of child safety by alerting caregivers if a child is left unattended in the vehicle. This feature can potentially save lives by preventing heatstroke incidents. The driver monitoring system aims to improve road safety by continuously assessing the driver's state and providing timely notifications if signs of drowsiness or distraction are detected.

The integration of these models with the Raspberry Pi and touch screen interface allows for intuitive interaction and real-time display of crucial data. The seamless communication between the ECUs and the Raspberry Pi ensures that the system can reliably gather and process information from various sensors and control units within the vehicle.

Moreover, the project showcases the versatility and potential of Raspberry Pi in developing sophisticated, cost-effective solutions for automotive safety. By leveraging the power of computer vision and machine learning, we have demonstrated how technology can be harnessed to address real-world problems and contribute to safer driving environments.

In conclusion, this project represents a significant step forward in vehicular safety technology. It highlights the importance of innovative thinking and interdisciplinary collaboration in solving complex challenges. The successful deployment and testing of this system underline its feasibility and effectiveness, setting the stage for future enhancements and broader applications. As automotive technology continues to evolve, the integration of intelligent systems such as this will play a crucial role in making our roads safer for everyone.

6.2. Future Work

While this project has made significant strides in enhancing vehicular safety, there remains substantial potential for further development and refinement. Future work on this system will focus on integrating more advanced ECUs, expanding computer vision capabilities, and broadening the application scope to address a wider range of safety and security concerns.

One of the primary areas of future enhancement is the incorporation of advanced ECUs such as those used in adaptive cruise control and engine monitoring. Adaptive cruise control will enable the vehicle to automatically adjust its speed to maintain a safe distance from other vehicles, thereby improving road safety and driving comfort. Engine monitoring ECUs will provide real-time diagnostics and performance data, ensuring that the vehicle operates efficiently and alerting drivers to potential issues before they become critical. Additionally, integrating security-focused ECUs will enhance the overall security of the vehicle, protecting it from unauthorized access and tampering.

Another significant avenue for future work involves expanding the computer vision models. Implementing face identification technology will add an additional layer of security by verifying the identity of the driver and passengers. This feature can be particularly useful in scenarios where access control is crucial. Furthermore, the system can be adapted to monitor multiple children in vehicles like school buses. By leveraging advanced computer vision techniques, the system can ensure the safety of all passengers, providing alerts if any child is left behind or if unusual behavior is detected.

Additionally, expanding the system's connectivity features to integrate with smart city infrastructure and other IoT devices will further enhance its functionality and user experience.

In conclusion, the future work on this project will build upon the solid foundation laid by the current system, aiming to create an even more robust, versatile, and comprehensive vehicle safety solution. By embracing new technologies and continuously innovating, we can ensure that our system remains at the forefront of automotive safety advancements, making our roads safer for everyone.

6.3. Tools

1. **Eclipse IDE.**
2. **STM32CubeIDE.**
3. **Visual Studio Code..**
4. **Putty**
5. **VNC viewer**
6. **Proteus**
7. **ARM Keil**
8. **Qemu**
9. **Jupiter**
10. **Raspberry pi imager**
11. **STM32 ST-LINK Utility: used to burn code to microcontroller**

References

1. National Highway Traffic Safety Administration. (2018). "Child Heatstroke Fatalities." Retrieved from [NHTSA](https://www.nhtsa.gov/child-safety/child-heatstroke)
2. TechCrunch. (2021). "The world is waking up to driver monitoring systems." Retrieved from [TechCrunch](https://techcrunch.com/2021/11/15/the-world-is-waking-up-to-driver-monitoring-systems/)
3. U.S. Congress. (2017). "Bill S.1663 - HOT CARS Act of 2017." Retrieved from [Congress.gov](https://www.congress.gov/bill/115th-congress/senate-bill/1663/text)
4. Viola, P., & Jones, M. (2001). "Rapid Object Detection using a Boosted Cascade of Simple Features." In Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2001). Retrieved from [IEEE Xplore](https://ieeexplore.ieee.org/document/990517)
5. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016). "SSD: Single Shot MultiBox Detector." In European Conference on Computer Vision (pp. 21-37). Springer, Cham. Retrieved from [SpringerLink](https://link.springer.com/chapter/10.1007/978-3-319-46448-0_2)
6. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). "You Only Look Once: Unified, Real-Time Object Detection." In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 779-788). Retrieved from [IEEE Xplore](https://ieeexplore.ieee.org/document/7780460)
7. LeCun, Y., Bengio, Y., & Hinton, G. (2015). "Deep learning." Nature, 521(7553), 436-444. Retrieved from [Nature](https://www.nature.com/articles/nature14539)
8. Hyundai. (2019). "Hyundai Introduces Rear Occupant Alert to Prevent Child Heatstroke Fatalities." Retrieved from [Hyundai News](https://www.hyundainews.com/en-us/releases/2827)
9. *The Definitive Guide to the ARM Cortex-M3 and Cortex-M4 Processors* by Joseph Yiu.
10. *STM32F10xxx Reference Manual (RM0008)*, STMicroelectronics.
11. *Programming Embedded Systems in C and C++* by Michael Barr.
12. *Embedded Systems: Real-Time Interfacing to the ARM Cortex-M Microcontroller* by Jonathan W. Valvano.

13. Lin, Z., et al. "Design and Implementation of a Real-time Driver Monitoring System Based on Deep Learning." *IEEE Transactions on Intelligent Transportation Systems*, 2020.
14. Wang, H., et al. "Automotive Embedded Systems Development: A Survey." *IEEE Access*, 2019.
15. Smith, J. "Optimizing ADC Performance in Embedded Systems." *Journal of Embedded Systems*, 2021.:
16. STMicroelectronics. "STM32F103C6 Datasheet."
17. Bosch. "CAN Specification Version 2.0."
18. STMicroelectronics. "STM32F103 Reference Manual."
19. YOLOv10: Real-Time End-to-End Object Detection
    Ao Wang, Hui Chen, Lihao Liu, Kai Chen, Zijia Lin, Jungong Han, Guiguang Ding
20. https://www.raspberrypi.com/documentation/computers/getting-started.html
21. https://www.raspberrypi.com/documentation/accessories/display.html