Mansoura University

Faculty of Engineering

Electronics and Communication Department

Secured Firmware Over The Air (FOTA) For ADAS Systems

Graduation Project (2023 - 2024)

Prepared by

Team Members	ID
1. Mohamed Fadel Mohamed	1000174690
2. Mahmoud Maher Khater	1000188475
3. Mahmoud Ahmed Elawady	1000174852
4. Nour Ibrahim El Morsy	1000174642
5. Mahmoud Ezzat Hussien	1000174282
6. Mahmoud Osama Sallam	1000174724
7. Sara Ali Dawood	1000174725
8. Youmna Yasser Abbas	1000188261

Supervised By

Assoc. prof. Abeer Twakol

Abstract

Nowadays, any automotive vehicle has a very large number of Electronic Control Units (ECUs), it may have around 100 ECUs and approximately 100 million lines for software coding. These numbers are increasing rapidly since connected cars have been introduced.

It’s suggested that by 2020, there will be 300 ECUs in a single car, to manage most of the functions and features within a car. After the vehicle's release, OEMs usually need to fix bugs, add new features, or update the ECUs’ software. This requires the customer to visit the maintenance center periodically. Recalling the vehicles to do that will not be considered as a good option for OEMs as it negatively affects their reputation, user experience, unnecessary cost and time.

We aim to update the software of a car by flashing the code on the ECUs over the air without using any physical connection, FOTA enables cars to be upgraded remotely, without having the user to worry about a software-related recall or update.

Our solution fulfill security, proper timing and also solving real life problems which impact cost reduction for OEMs and user satisfaction and make our life easier.

Acknowledgment

We would like to express our deep gratitude to **Assoc. prof. Abeer Twakol** and **Eng. Mohamed Abdo** our project supervisors, provision of expertise, And technical support in design and implementation. Without their superior knowledge and experience, the Project would not be in perfect quality of outcomes, and thus their support has been essential for their patient guidance, enthusiastic encouragement, Useful critiques of this project work and their advice and assist in keeping our progress on schedule.

Many thanks to **Prof. Dr. Sherif Masoud Al-Badawi**, the principal of the faculty of engineering Mansoura university, **Prof. Dr. Nihal Fayez**, the head of the Electronics and communication department.

Table of Contents

List of figures

List of Abbreviations

FOTA	Firmware Over The Air
OEM	Original Electronic Manufacturer
GUI	Graphical User Interface
TCU	Telematics Control Unit
ECU	Electronic Control Unit
MCU	Microcontroller Unit
BL	Bootloader
SOF	Start Of Frame
EOF	End Of Frame
CRC	Cyclic Redundancy Check
ADAS	Advanced Driver Assistance Systems
ACC	Adaptive Cruise Control
RTOS	Real Time Operating System
UART	Universal Asynchronous Receiver Transmitter
CAN	Controller Area Network

CHAPTER 1

FOTA INTRODUCTION

GRADUATION PROJECT 2024

1 Chapter 1: FOTA Introduction

1.1 History of FOTA

Firmware Over-The-Air (FOTA) is a Mobile Software Management (MSM) technology in which a mobile device's operating firmware is upgraded and updated wirelessly by its manufacturer. FOTA-capable phones directly download the updates from the service provider. Depending on link speed and file size the process typically takes three to 10 minutes.

FOTA took 4 management stages process:

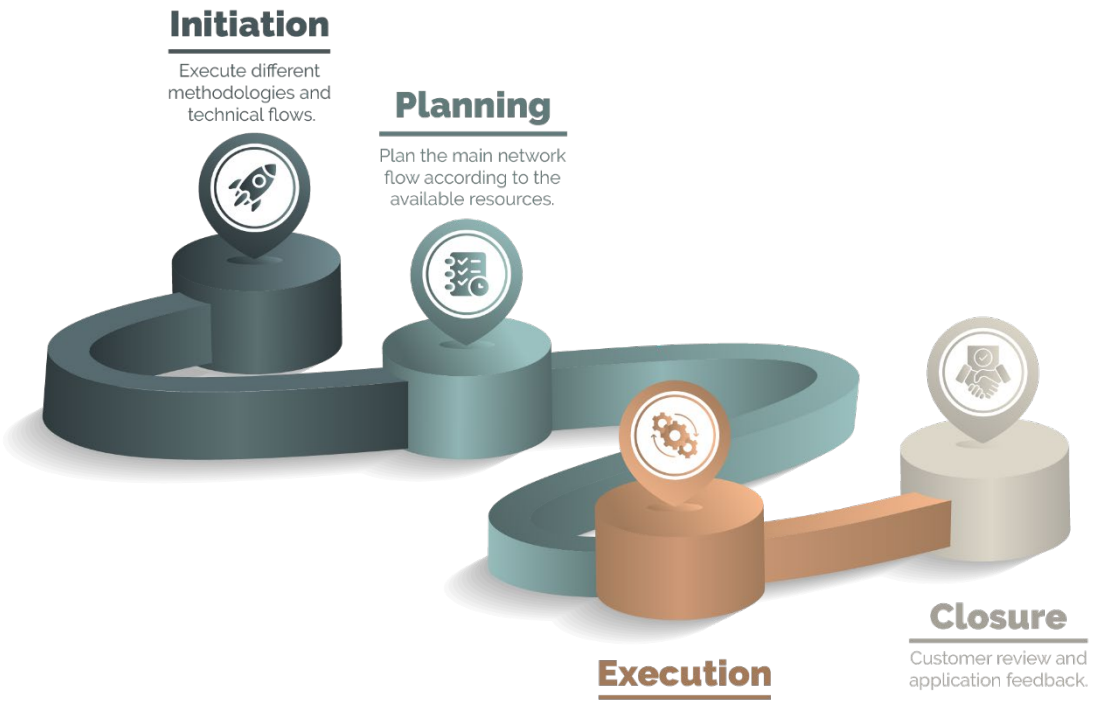


Figure 1-1: FOTA management process

Through these stages FOTA shall facilitate:

- Enables fabricators to patch glitches in new systems.
- Allows OEMs to be able to send and install new software updates and features remotely after customers have bought the cars.

FOTA represent a tremendous opportunity for automotive manufacturers, with a clear route to cost savings, particularly during vehicle warranty periods; revenue gains from the sales of new features and services, and a more engaged relationship with clients.

So, the management process took its time to bring out the whole fully experience to the customer and the OEMs, the imperative to move towards FOTA stems from the autonomous, connected, and electric vehicles trend in the automotive sector. As these vehicles are using increasingly complex hardware and operating systems, vendors would need a mechanism to repair, manage and enhance any aspect of vehicle efficiency.

In order to achieve that it was about the first steps toward any idea a good strategy to overcome the complexity of such application by dividing the process within different structures.

Next, we will summarize the historical keys of development.

1.2 Problem Definition

Today, an automotive vehicle has on average 100 **Electronic Control Units** (ECU) and about 100 million lines of software code. At an average cost of $10 per line, these electronic systems aren't cheap and recalls lead to multi-billion-dollar losses and also leave a dent on manufacturers' reputations. This situation will be worse as this number is growing rapidly such that experts suggest that there will be 300 ECUs in a car in 2020 so that, OEMs must reduce the impact of product recalls by managing the software efficiently over the lifecycle of the vehicle.

So, there are several constrains to update the cars' software:

1. **Time**: upgrading the firmware could take many days.
2. **Complexity**: upgrading the firmware must be completely handled at the application layer.
3. **Cost**: to upgrade the firmware of the car, people must go to maintenance centers.

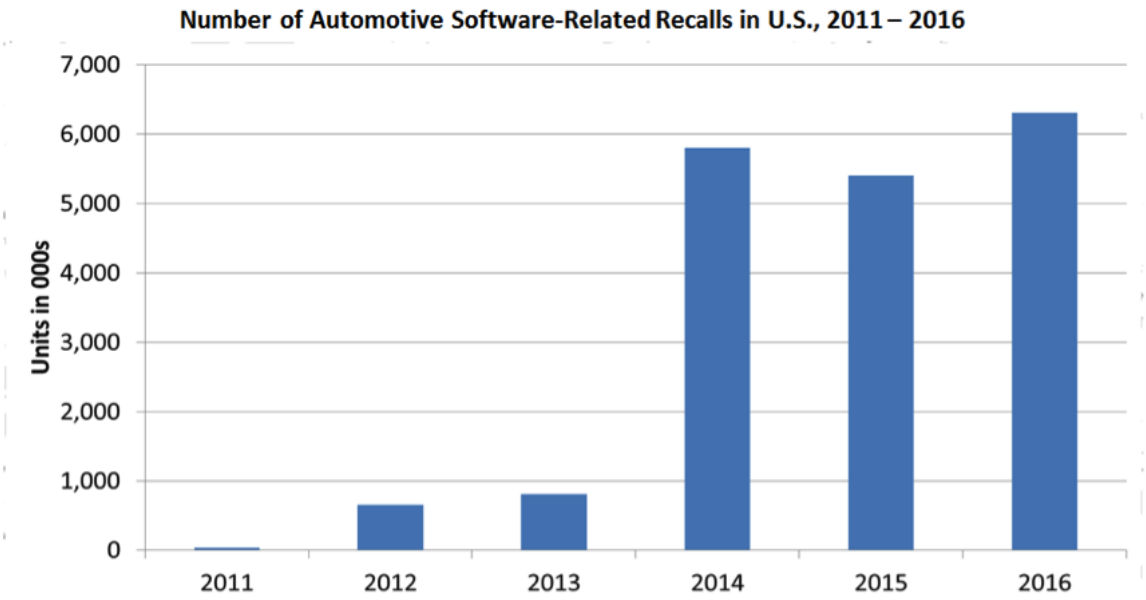


Figure 1-2: Automotive software recalls

1.3 Key developments timeline

In the automobile industry, in September 2012, Tesla delivered the first FOTA software update for their 'Model S' vehicles. The business published an update and use either car's integrated 3 G network link or a Wi-Fi signal from the customer's home network.

Other automotive OEMs such as General Motors, BMW, Volvo, Detroit Diesel Organization and Mercedes-Benz have begun to deliver updates to OTA from 2017-2018. Ford also plans to deliver updates to OTA by 2020. Agricultural machinery manufacturers including John Deere, AGCO Company, and CNH

Industrial have begun providing simple OTA updates for their farm machinery along with automotive OEMs.

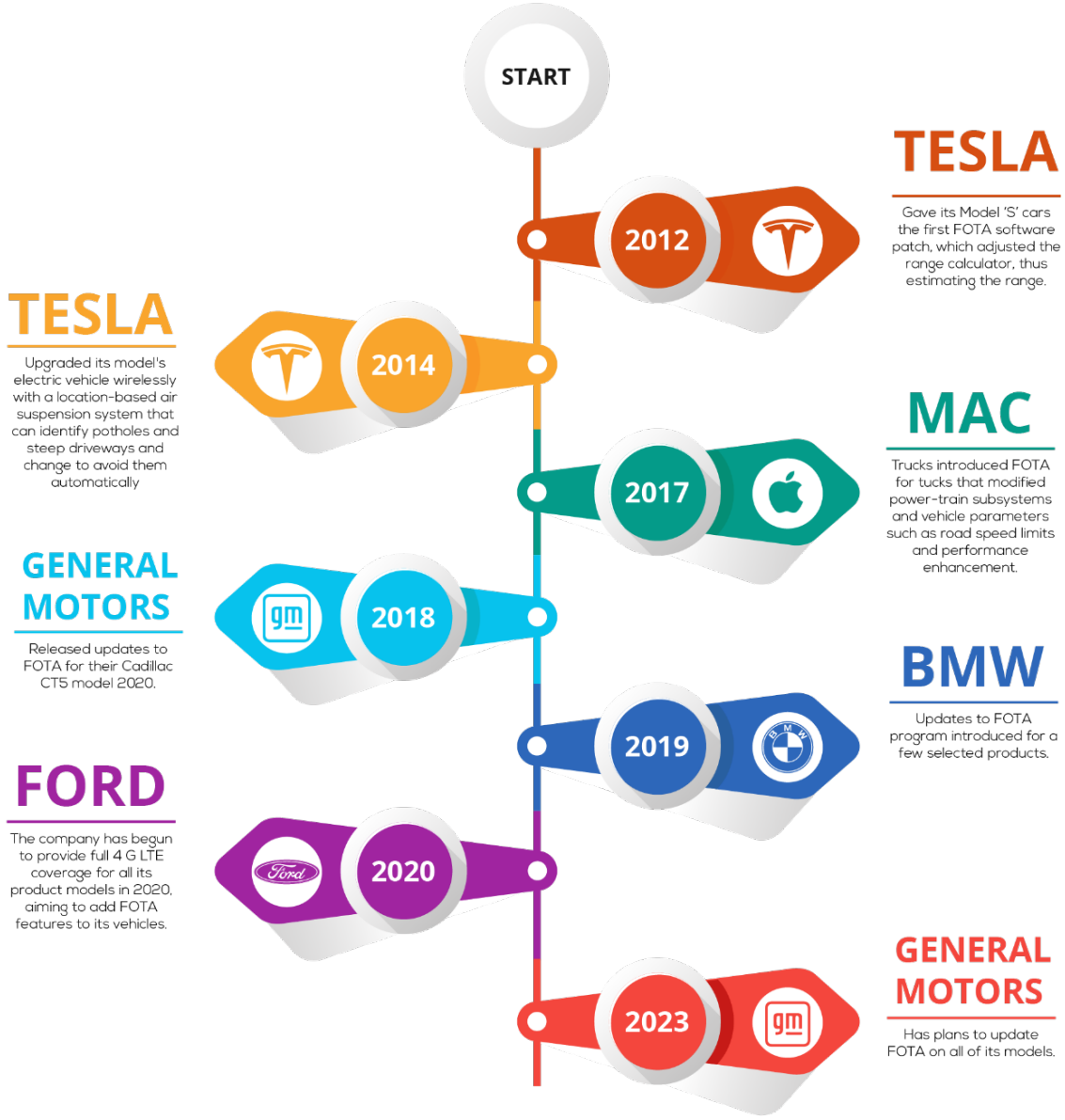


Figure 1-3: Key developments timeline

In the automotive industry, FOTA updates go hand in hand with self-driving and connected vehicles. IOT updates will be an important part of the near future, according to industry analysts, by 2030, 98 percent of cars sold worldwide are projected to be connected vehicles. The value produced by next-generation cars (in terms of the cost of the vehicle) will be totally different than the current ones; the software will be a key selling point in next-generation cars. To sell their

vehicles in the near future, automotive OEMs will deliver robust software along with hardware systems.

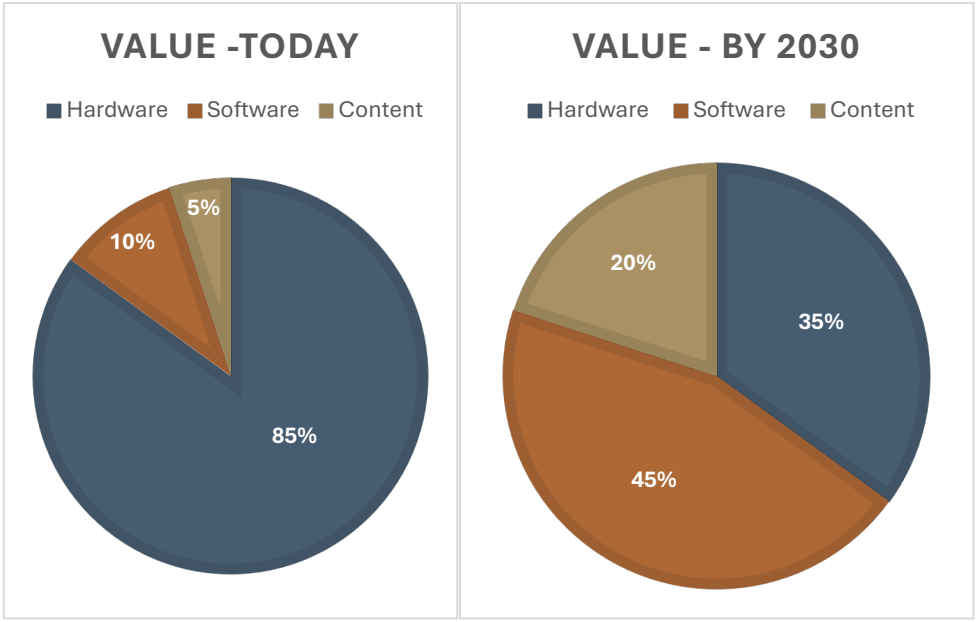


Figure 1-4: Automotive value

Apparently, industry forecasts say that FOTA 's future in automotive is looking promising. This is mainly relevant to the automotive industry which is shifting its gear towards software-driven autonomous systems development.

The Research and Economies reported that the Automotive Over the Air (FOTA) updates market is expected to expand at a CAGR of 58.15 per cent over the period 2018-2022.And many of the largest automobile players are already on the path of making FOTA in automotive systems seamless and stable upgrade, a fact.

1.4 Key players and market

The over-the-air update is a means of delivering new applications, firmware, and other cloud-based upgrading facilities. The FOTA updates were only available for smartphones until recent time; however, with the significant increase in the number of connected cars around the globe and vehicles becoming more software-dependent, the need for FOTA updates has increased significantly. Software upgrades have long been achieved by local software upgrade procedures, whereby the car is sent to a vendor / mechanic who upgrades the software afterwards.

CHAPTER 2

METHODOLOGY

GRADUATION PROJECT 2024

2 Chapter 2: Methodology

2.1 Methodology Overview

The development of the FOTA system will be guided by Agile methodologies, specifically using an iterative and incremental approach. Agile methodologies offer flexibility, continuous feedback, and adaptive planning, which are essential for the dynamic nature of FOTA systems.

2.1.1 Significance for FOTA

1) **Operational Efficiency**: Streamline firmware updates, reducing downtime and operational disruptions.

2) **Cost Savings**: Minimize physical interventions, contributing to significant cost savings.

3) **Customer Satisfaction**: Improve the overall customer experience with hassle-free firmware updates

2.2 Agile Methodology

Agile development divides the project into small, manageable units called sprints, typically lasting two to three weeks. Each sprint involves planning, development, testing, and review phases, ensuring that a potentially shippable product increment is delivered at the end of each cycle. This iterative process allows for continuous feedback and adjustments, aligning the development closely with customer requirements.

2.3 Development Tools and Technologies Development Environment

The project will utilize a range of tools and technologies to ensure efficient development and high-quality output. Key components include:

- Integrated Development Environment (IDE):
- Tools like STM32Cube IDE and Kiel for coding and debugging.
- **Version Control**: Git for version control and collaboration.
- Continuous Integration (CI)
- Testing Frameworks

2.4 Firmware and Server Development

- **Programming Languages**: C/C++ for firmware development, Python/Java for server-side components.
- **Communication Protocols**: Implementation of secure protocols for communication between the server and vehicle ECUs.

2.5 User Interface

- **In-Car Application:** Embedded development tools and frameworks to ensure compatibility with automotive standards.

2.6 Testing and Validation

Testing is a critical component of the Agile development process, ensuring that each increment of the FOTA system is functional and reliable. The testing process will include:

2.6.1 Unit Testing

- **Objective:** Verify that individual components of the firmware and server are workingcorrectly.
- **Tools:** Google Test, JUnit, or similar frameworks.

2.6.2 Integration Testing

- **Objective:** Ensure that the firmware and server components work together seamlessly.
- **Tools:** Custom scripts and tools to simulate real-world scenarios.

2.7 Security Considerations

Security is paramount in a FOTA system to protect against unauthorized access andpotential cyber threats. Key security measures include:

- **Encryption:** Using strong encryption algorithms to secure communication andfirmware data.
- **Authentication:** Implementing robust authentication mechanisms to verify the identity of the update source.
- **Integrity Checks:** Ensuring firmware integrity with checksums or digital signaturesto prevent tampering.

2.8 Documentation and Reporting

Comprehensive documentation will be maintained throughout the project to ensuretransparency and facilitate knowledge transfer. Key documents include:

- **Requirement Specifications:** Detailed documentation of project requirements.
- **Design Documents:** Architectural and design specifications of the FOTA system.
- **Test Plans:** Detailed test plans and results.

- **User Manuals:** Guides for end-users on how to use the FOTA system.

2.9 TimeLine

Our development process consists of:-

- Project delivery sprints: Six sprints.
- Sprints duration: One Week.

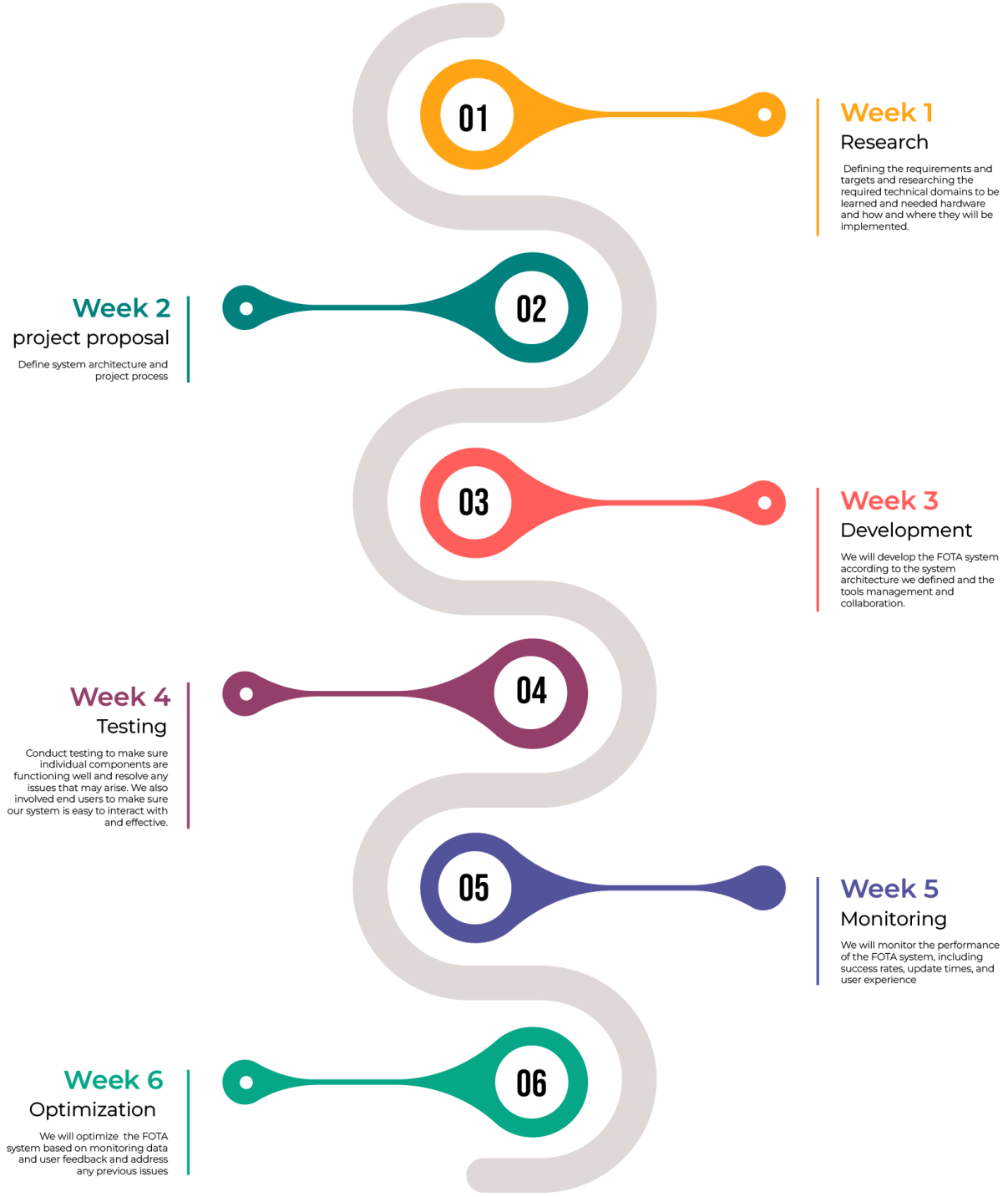


Figure 2-1: TimeLine of Project

2.10 Conclusion

The Agile methodology provides a structured yet flexible framework for developing a robustFOTA system for automotive applications. By dividing the project into manageable sprints, the development process remains focused, adaptive, and aligned with customer needs.

This approach ensures that the final product is not only functional and secure but also meets the high standards of modern automotive technology. The successful implementation of this project will demonstrate the effectiveness of Agile methodologies in embedded systems development and contribute to advancements in automotive technology.

CHAPTER 3

PROJECT OVERVIEW

GRADUATION PROJECT 2024

3 Chapter 3: Project Overview

3.1 Introduction:

In this chapter, a quick overview of the project features and flow will be explained in detail, in addition to the block diagram of the system. The solution consists of two main features: Software Updates and Live Diagnostics. The flow of each of them will be discussed in detail

3.2 About FOTA:

As IoT devices proliferate, the need for efficient firmware updates becomes crucial. Firmware Over The Air (FOTA) emerges as a solution, allowing remote updates without physical access. Our project addresses the challenges in firmware management for diverse embedded systems within the IoT ecosystem.

3.3 Overview:

Traditional firmware update methods are often impractical for widespread IoT deployments. FOTA enables wireless updates, streamlining the transition to new firmware versions. Our project aims to design a robust and scalable FOTA system, emphasizing security and compatibility for various embedded system architectures.

3.4 Block Diagram:

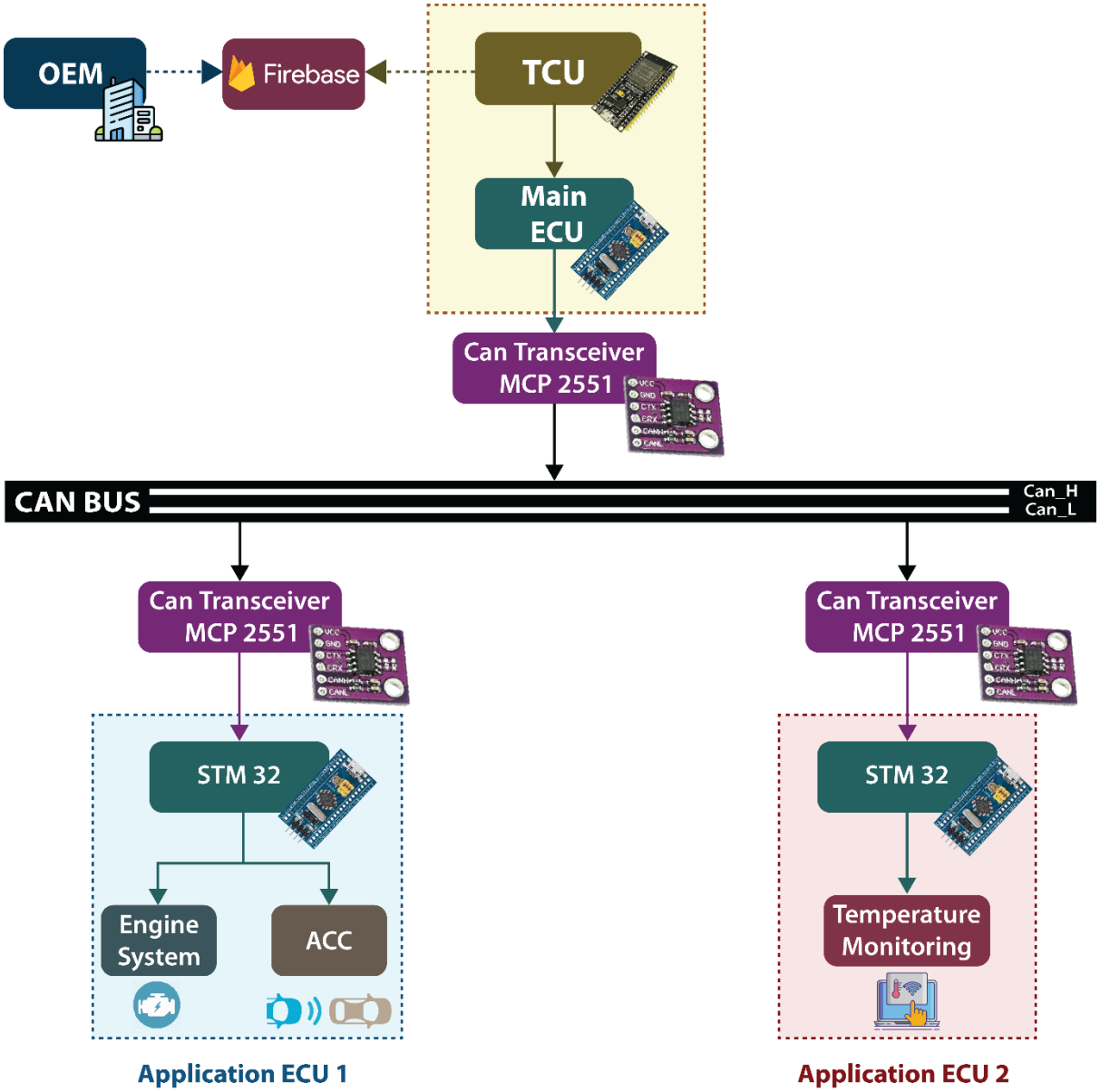


Figure 3-1: Block Diagram

As shown the system consists of 2 main controllers:

1. **Main ECU**: It is connected to a TCU and acts as the gateway to the server.
2. **Application ECU**: It acts as a faulty node whose software needs to updates or have a feature added to it.

3.5 Software Flowchart:

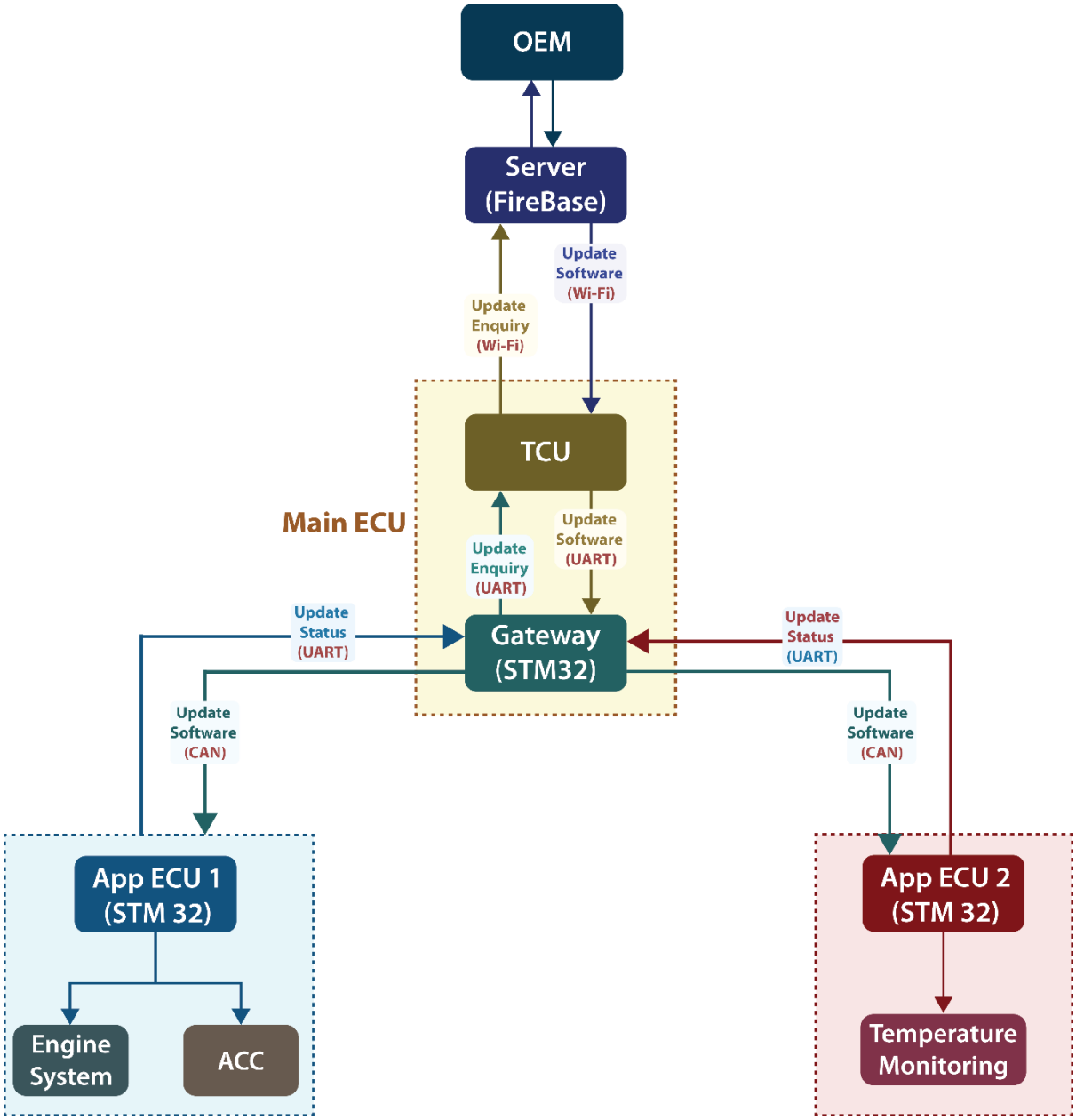


Figure 3-2: Software Flowchart

This diagram represents a system architecture for a vehicle's electronic control unit (ECU) network, which is designed to handle software updates and diagnostics for various subsystems. Here's an explanation of the different components and their interactions:

In this diagram, the main feature is the "Gateway (STM32)". This component serves as the central hub that manages communication and data flow between various parts of the system. Here's why the Gateway (STM32) is the main feature:

3.5.1 Central Communication Hub:

- The Gateway connects to the Telematics Unit, CAN Controller, App ECU 1, and App ECU 2.
- It receives update software data from the TCU via UART.
- It distributes update software to App ECU 1 (via CAN) and App ECU 2 (via CAN).

3.5.2 Interface with CAN Controller:

- The Gateway communicates with the CAN Controller to handle update enquiries and permissions.
- It sends update requests and receives update permissions via CAN.

3.5.3 Coordination with User Interface:

- the Gateway (Main ECU) is the main feature because it coordinates all the data and update processes, acting as the pivotal point that ensures smooth communication and operation of the entire ECU network in the vehicle.

3.6 Components:

3.6.1 OEM (Original Equipment Manufacturer)

- The source of updates and diagnostic data.
- Sends update software and diagnostic data to the server (Firebase).

3.6.2 Server (Firebase)

- Acts as a central repository for update software data.
- Communicates with TCU module over Wi-Fi.

3.6.3 Telematics Unit

- A microcontroller with Wi-Fi capabilities.
- Receives updates data from the server via Wi-Fi.
- Sends update software data to the Gateway via UART.

3.6.4 Gateway (STM32)

- Central hub for communication within the vehicle's ECU network.
- Receives update software data from Telematics Unit.
- Distributes update software to various ECUs over CAN (Controller Area Network) and UART.

3.6.5 App ECU 1 (STM32)

- Handles specific application functions such as the Engine System and Collision System.
- Receives update software data via CAN.

3.6.6 App ECU 2 (STM32)

- Handles specific application functions such as Temperature Monitoring.
- Receives update software via CAN and via UART.

3.7 Data Flow:

3.7.1 Update Software and Diagnostics Data Flow

- OEM to Server: Update software data are uploaded from the OEM to the server.
- Server to Telematics Unit: The server sends this data to the TCU over Wi-Fi.
- TCU to Gateway: TCU transfers the data to the Gateway via UART.
- Gateway to App ECUs: The Gateway distributes the update software to App ECU 1 and App ECU 2 via CAN (for App ECU 2) and UART (for App ECU 1).

3.8 Communication Protocols:

- Wi-Fi: Used for communication between the Server and TCU.
- UART: Used for communication between the TCU and Gateway, as well as between the Gateway and App ECUs.
- CAN: Used for communication within the vehicle's ECU network, including update approval and permissions, and between the Gateway and App ECU 2.

This architecture allows for centralized control and monitoring of the vehicle's various subsystems, enabling efficient software updates data collection.

CHAPTER 4

BOOTLOADER

GRADUATION PROJECT 2024

4 Chapter 4: Bootloader

4.1 History of Bootloaders:

In the early days of computing, bootstrapping was a manual process. Operators used front-panel switches to input initial instructions, which loaded a more complex program from secondary storage. This method was labor-intensive and error-prone.

4.1.1 First Generation Bootloaders

During the 1970s, automated booting processes emerged. IBM mainframes, for example, utilized Basic Input/Output System (BIOS) stored in ROM to initialize hardware and load the operating system from external storage. Minicomputers, like DEC's PDP-11, employed similar techniques, marking the transition from manual to automated bootstrapping

4.1.2 Personal Computers and Standardization

The advent of personal computers in the 1980s brought standardization to the boot process. IBM PCs introduced BIOS, which became a de facto standard for PC-compatible systems. BIOS managed the initial hardware checks (POST) and loaded the boot sector from the bootable device, typically a floppy disk or hard drive.

4.1.3 Modern Bootloaders

in the 2000s, the introduction of the Unified Extensible Firmware Interface (UEFI) replaced the aging BIOS. UEFI offers a more flexible and secure pre-boot environment, supporting features like secure boot and faster boot times. Modern bootloaders, such as GRUB 2, have become essential in handling sophisticated boot processes and security requirements.

4.2 Bootloader definition

A bootloader is a critical piece of software embedded within a device that executes upon powering on or resetting the system. It serves as the initial program that initializes the hardware and loads the main operating system or firmware into the device's memory. Essentially, the bootloader acts as a bridge between the hardware and the higher-level software, ensuring that the system starts up correctly and securely.

4.3 The primary purpose of a bootloader in FOTA:

is to allow systems software to be updated without the use of specialized hardware such as JTAG programmer.

4.4 Data Frames in Bootloader Communication

A **data frame** is a structured packet of data used in communication protocols to transmit commands, data, and responses between devices. In bootloader communication, data frames typically consist of:

1. **Length**: The length of the entire data frame, including the header and payload. This helps the receiver know how many bytes to expect.
2. **Command/ID**: An identifier that specifies the type of command or data being sent. This tells the bootloader what action to perform.
3. **Payload/Data**: The actual data or parameters associated with the command. This could be firmware data, configuration settings, or other information.
4. **Checksum/CRC**: An error-checking value used to verify the integrity of the data frame. This ensures that the data has not been corrupted during transmission.

4.4.1 Host command format:

⇨ Command length

(1 byte = N) + command code (1 byte) + details (N bytes) +CRC (4 byte)

```
    ⇨  #define CBL_GET_VER_CMD                          0x10
This command used to read bootloader version from MCU
    ⇨  #define CBL_GET_HELP_CMD                         0x11
This command used to know commands supported by bootloader
    ⇨  #define CBL_GET_CID_CMD                          0x12
This command used to read MCU identification number
    ⇨  #define CBL_GET_RDP_STATUS_CMD                   0x13
This command used to read flash read protection level
    ⇨  #define CBL_GO_TO_ADDR_CMD                       0x14
This command used to jump bootloader to specific address
    ⇨  #define CBL_FLASH_ERASE_CMD                      0x15
This command used to mass erase or sector erase of user flash
    ⇨  #define CBL_MEM_WRITE_CMD                        0x16
This command used to write data in different memories in MCU
    ⇨  #define CBL_ED_W_PROTECT_CMD                     0x17
This command used to enable read/write protect on different sectors
    ⇨  #define CBL_MEM_READ_CMD                         0x18
This command used to read data from different memories
    ⇨  #define CBL_READ_SECTOR_STATUS_CMD               0x19
This command used to read all sectors protection status
    ⇨  #define CBL_OTP_READ_CMD                         0x20
This command used to read protection level
    ⇨  #define CBL_CHANGE_ROP_Level_CMD                 0x21
This command used to change read protection level
/* CRC_VERIFICATION */
      #define CRC_TYPE_SIZE_BYTE                       4
Size of CRC in Bytes
      #define CRC_VERIFICATION_FAILED                  0x00
Send 0 if CRC verification failed
      #define CRC_VERIFICATION_PASSED                  0x01
Send 1 if CRC verification succeeded
      #define CBL_SEND_NACK                            0xAB
      Not Acknowledge
      #define CBL_SEND_ACK                             0xCD
      Acknowledge
```

4.5 Bootloader CRC Verification

CRC (Cyclic Redundancy Check) verification in a bootloader context refers to the process of using CRC algorithms to ensure the integrity of data, typically firmware images, during the bootloader's operation. It is a crucial step in preventing the loading of corrupted or tampered firmware, which could compromise the functionality and security of the device.

4.5.1 Purpose of CRC Verification

The primary purpose of CRC verification in a bootloader includes:

1. **Data Integrity**: CRC ensures that the firmware image or data being loaded into the device's memory is not corrupted or altered during transmission or storage. This is critical to prevent system failures or vulnerabilities caused by faulty firmware.

2. **Security**: By verifying the integrity of the firmware image using CRC, bootloaders can detect unauthorized modifications or tampering attempts. This protects the device from potential security threats, such as malware injection or unauthorized access.

3. **Reliability**: Bootloaders rely on CRC verification to ensure that only valid and verified firmware updates are applied to the device. This helps maintain the overall reliability and performance of the system.

4.5.2 Verification of CRC

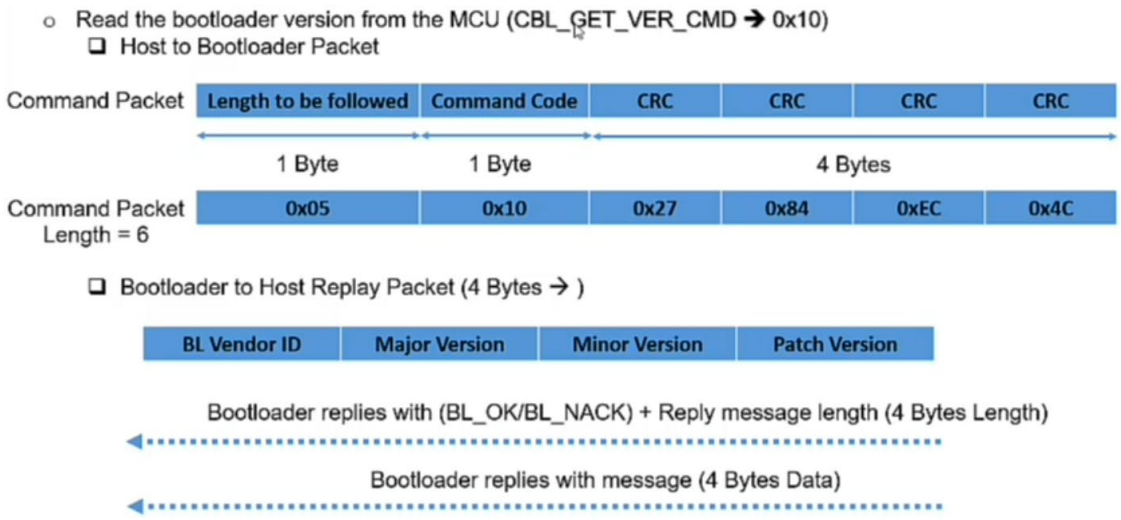
- ✓ During the firmware loading process, the bootloader recalculates the CRC value over the received firmware data.
- ✓ It compares the recalculated CRC value with the stored CRC value (from the firmware metadata).
- ✓ If the recalculated CRC matches the stored CRC, the bootloader considers the firmware image valid and proceeds with the loading process.
- ✓ If there is a mismatch, indicating data corruption or tampering, the bootloader can reject the firmware image and may initiate error handling or recovery procedures.

4.6 Host to bootloader communication

At first host send command to bootloader , then bootloader replies with Acknowledgement(Ack) if command verified or (NACK) if command not verified + message length.

4.7 Main Functions used on bootloader driver:

4.7.1 Get version function



This Function Is Used TO Get Version Of The Bootloader:

```
static void Bootloader_Get_Version(uint8_t *Host_Command_Buffer)
	{
		uint8_t Bootloader_Version[4] = {CBL_VENDOR_ID , CBL_SW_MAJOR_VERSION
, CBL_SW_MINOR_VERSION , CBL_SW_PATCH_VERSION};
		uint16_t Host_Bootloader_Command_Packet_length = 0 ;
		uint32_t Host_Bootloader_CRC = 0 ;
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
	Bootloader_Print_message("Read the bootloader version from the MCU \r\n");
#endif
		Host_Bootloader_Command_Packet_length                                    =
Bootloader_Host_Command_Buffer[0] + 1 ;
		Host_Bootloader_CRC = *((uint32_t *)((Bootloader_Host_Command_Buffer +
Host_Bootloader_Command_Packet_length) - CRC_TYPE_SIZE_BYTE));
		if(CRC_VERIFICATION_PASSED    ==    Bootloader_CRC_Verification((uint8_t
*)&Host_Command_Buffer[0]      ,      Host_Bootloader_Command_Packet_length-4      ,
Host_Bootloader_CRC))
		{
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
		Bootloader_Print_message("CRC Verification Passed \r\n");
#endif
		Bootloader_Send_ACK(4);
			Bootloader_Send_Data_To_Host((uint8_t
*)(&Bootloader_Version[0]) , 4);
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
```

```
    Bootloader_Print_message("Bootloader Ver. %d.%d.%d \r\n", Bootloader_Version[1],
Bootloader_Version[2], Bootloader_Version[3]);
#endif
                }
                else
                {
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
       Bootloader_Print_message("CRC Verification Failed \r\n");
#endif
                      Bootloader_Send_NACK();
                }

       }
```

4.7.1.1 Function explanation

Bootloader_Get_Version Function Overview

The **`Bootloader_Get_Version`** function is designed to handle a request from a host device to retrieve the bootloader version. It performs CRC verification to ensure the integrity of the received command and sends the version information back to the host if the verification passes.

Key Steps

1. Initialize Bootloader Version:
   - The bootloader version information (vendor ID, major, minor, and patch versions) is stored in an array.

2. Extract Command Packet Length and CRC:
   - The length of the command packet is determined from the first byte of the received buffer.
   - The CRC value is extracted from the end of the command packet.

3. CRC Verification:
   - The function verifies the integrity of the command data using CRC.
   - If the CRC check passes, it proceeds to send the version information; otherwise, it sends a negative acknowledgment (NACK).

4. Send Response to Host:
   - If CRC verification is successful:

     Sends an acknowledgment (ACK) to the host.

     Sends the bootloader version information to the host.
   - If CRC verification fails:

     Sends a negative acknowledgment (NACK) to the host.

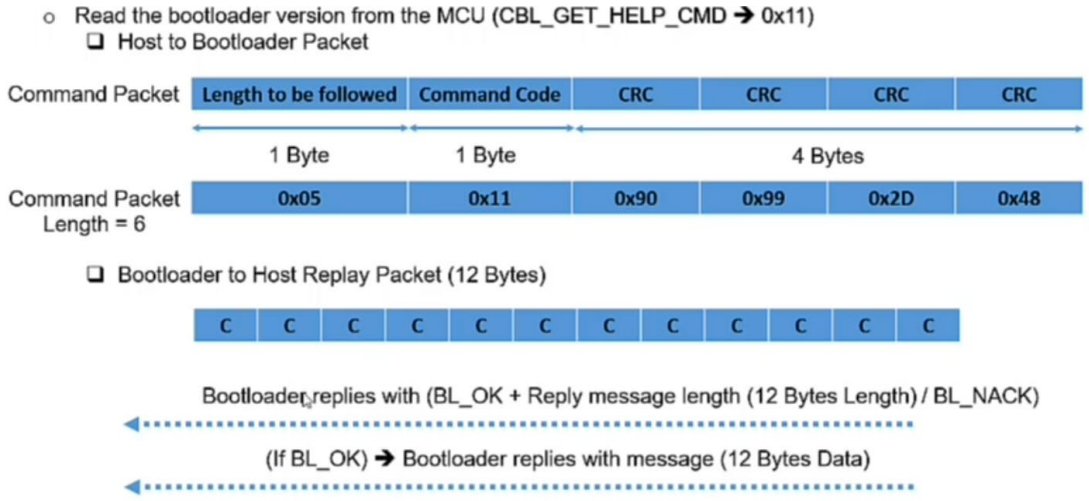
5. Debug Messages:
   - Optional debug messages are printed to provide feedback on the process (if debugging is enabled).

4.7.1.2 Summary

The function ensures secure and reliable communication between the host and the bootloader by verifying the integrity of the received command before responding with the bootloader version information. If the command is valid, the bootloader sends back the version details; otherwise, it notifies the host of the failure.

4.7.2 Get help function

This Function Is Used TO To Give Information About Bootloader Commands:



```
static void Bootloader_Get_Help(uint8_t *Host_Command_Buffer)
	{
		uint16_t Host_Bootloader_Command_Packet_length = 0 ;
		uint32_t Host_Bootloader_CRC = 0 ;
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
	Bootloader_Print_message("Read the commands supported by the bootloader \r\n");
#endif
		Host_Bootloader_Command_Packet_length = Bootloader_Host_Command_Buffer[0] + 1 ;
		Host_Bootloader_CRC = *((uint32_t *)((Bootloader_Host_Command_Buffer + Host_Bootloader_Command_Packet_length) - CRC_TYPE_SIZE_BYTE));

		if(CRC_VERIFICATION_PASSED == Bootloader_CRC_Verification((uint8_t *)&Host_Command_Buffer[0] , Host_Bootloader_Command_Packet_length-4 , Host_Bootloader_CRC))
		{
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
		Bootloader_Print_message("CRC Verification Passed \r\n");
#endif
		Bootloader_Send_ACK(12);
			Bootloader_Send_Data_To_Host((uint8_t *)Bootloader_Supported_CMDs , 4);
		}
		else
		{
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
		Bootloader_Print_message("CRC Verification Failed \r\n");
#endif
			Bootloader_Send_NACK();
		}
	}
```

4.7.2.1 Bootloader_Get_Help Function Overview

The **`Bootloader_Get_Help`** function is designed to respond to a host device's request for a list of commands supported by the bootloader. It ensures the integrity of the received command using CRC verification and sends the supported commands back to the host if the verification passes.

Key Steps

1. Local Variables:
   - **`Host_Bootloader_Command_Packet_length`**: Stores the length of the received command packet.
   - **`Host_Bootloader_CRC`**: Stores the CRC value extracted from the received command packet.

2. Debug Message:
   - If debugging is enabled, it prints a message indicating that the bootloader will read the supported commands.

3. Extract Command Packet Length and CRC:
   - The length of the command packet is determined from the first byte of the buffer plus one.
   - The CRC value is extracted from the end of the command packet.

4. CRC Verification:
   - The function verifies the integrity of the command data using the Bootloader_CRC_Verification function.
   - If the CRC check passes, it proceeds to send the list of supported commands; otherwise, it sends a negative acknowledgment (NACK).

5. Send Response to Host:
   - If CRC verification is successful:

     Sends an acknowledgment (ACK) to the host indicating success.

     Sends the list of supported commands to the host.

     If debugging is enabled, prints a message indicating successful CRC verification.
   - If CRC verification fails:

     Sends a negative acknowledgment (NACK) to the host.

     If debugging is enabled, prints a message indicating failed CRC verification.

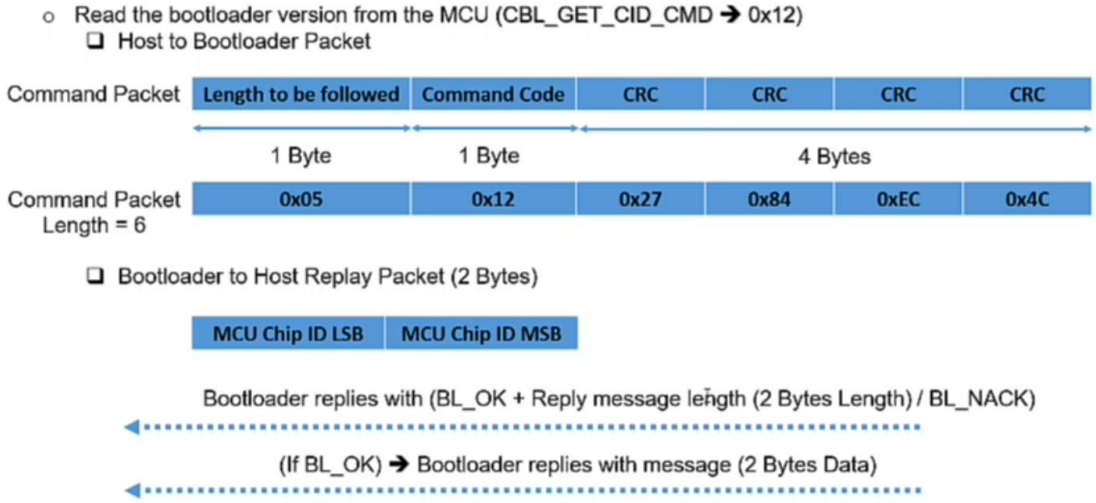
4.7.2.2 Summary

1. Extracts the command packet length and CRC value from the received buffer.
2. Performs CRC verification to ensure data integrity.
3. If the CRC check passes, it sends an acknowledgment and the list of supported commands to the host.
4. If the CRC check fails, it sends a negative acknowledgment (NACK) to the host.

This function ensures secure and reliable communication between the host and the bootloader, providing the host with necessary information about the bootloader's capabilities.

4.7.3 Get chip identification number

This Function Is Used TO Get The Id Of The Chip On MCU:




```
static void Bootloader_Get_Chip_Identification_Number(uint8_t *Host_Command_Buffer)
	{
		uint16_t Host_Bootloader_Command_Packet_length = 0 ;
		uint32_t Host_Bootloader_CRC = 0 ;
		uint16_t MCU_Identification_Number = 0;
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
	Bootloader_Print_message("Read the MCU chip identification number \r\n");
#endif
		Host_Bootloader_Command_Packet_length 
Bootloader_Host_Command_Buffer[0] + 1 ;
		Host_Bootloader_CRC = *((uint32_t *)((Bootloader_Host_Command_Buffer +
Host_Bootloader_Command_Packet_length) - CRC_TYPE_SIZE_BYTE));

		if(CRC_VERIFICATION_PASSED == Bootloader_CRC_Verification((uint8_t
*)&Host_Command_Buffer[0] , Host_Bootloader_Command_Packet_length-4 ,
Host_Bootloader_CRC))
		{
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
		Bootloader_Print_message("CRC Verification Passed \r\n");
#endif
			MCU_Identification_Number = (uint16_t)((DBGMCU->IDCODE) &
0X00000FFF);
		Bootloader_Send_ACK(2);
			Bootloader_Send_Data_To_Host((uint8_t
*)&MCU_Identification_Number , 2);
		}
		else
		{
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
		Bootloader_Print_message("CRC Verification Failed \r\n");
#endif
			Bootloader_Send_NACK();
		}
	}
```

4.7.3.1 Bootloader_Get_Chip_Identification_Number Function Overview

The **`Bootloader_Get_Chip_Identification_Number`** function is designed to handle requests from a host device to retrieve the MCU (Microcontroller Unit) chip identification number. It ensures the integrity of the received command using CRC verification and sends the chip identification number back to the host if the verification passes.

Key Steps

1. Local Variables:
   - **`Host_Bootloader_Command_Packet_length`**: Stores the length of the received command packet.
   - **`Host_Bootloader_CRC`**: Stores the CRC value extracted from the received command packet.
   - **`MCU_Identification_Number`**: Stores the MCU identification number.
2. Debug Message:
   - If debugging is enabled, it prints a message indicating that the bootloader will read the MCU chip identification number.
3. Extract Command Packet Length and CRC:
   - The length of the command packet is determined from the first byte of the buffer plus one.
   - The CRC value is extracted from the end of the command packet.
4. CRC Verification:
   - The function verifies the integrity of the command data using the **`Bootloader_CRC_Verification`** function.
   - If the CRC check passes, it proceeds to read the MCU identification number; otherwise, it sends a negative acknowledgment (NACK).

5. Send Response to Host:
   - If CRC verification is successful:

     Reads the MCU identification number from the appropriate register.

     Sends an acknowledgment (ACK) to the host indicating success.

     Sends the MCU identification number to the host.

     If debugging is enabled, prints a message indicating successful CRC verification.
   - If CRC verification fails:

     Sends a negative acknowledgment (NACK) to the host.

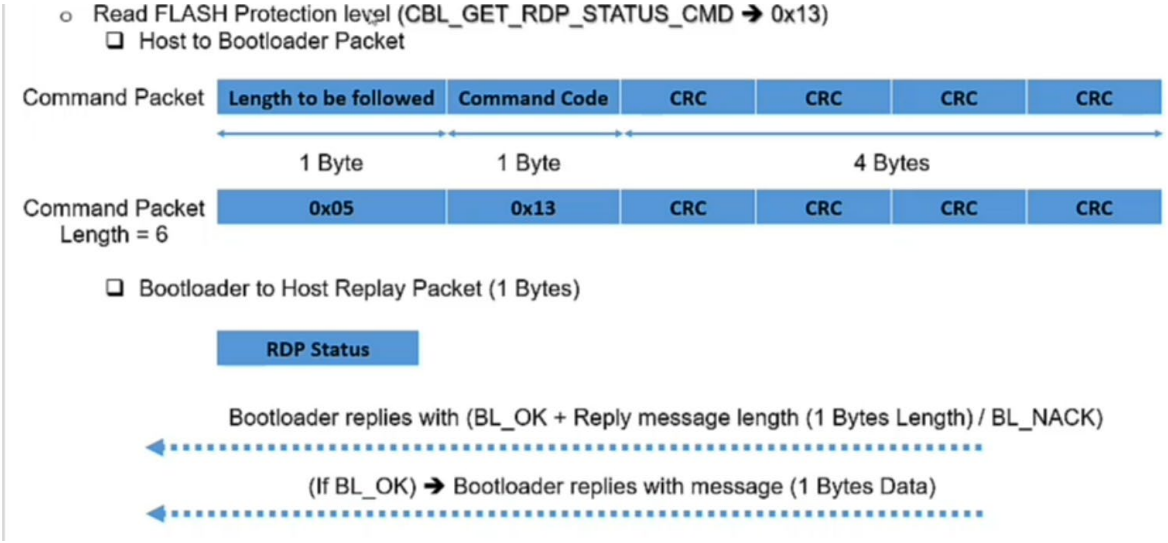
     If debugging is enabled, prints a message indicating failed CRC verification.

4.7.3.2 Summary

This function ensures secure and reliable communication between the host and the bootloader, providing the host with necessary information about the MCU chip identification number while verifying the integrity of the received command.

4.7.4 Read protection level

This Function Is Used TO Read The Protection Level Of The Memory (Flash Memory)




```
static void Bootloader_Read_Protection_Level(uint8_t *Host_Command_Buffer)
	{
		uint16_t Host_Bootloader_Command_Packet_length = 0 ;
		uint32_t Host_Bootloader_CRC = 0 ;
		uint8_t Protection_level = 0;

		Host_Bootloader_Command_Packet_length                                    =
Bootloader_Host_Command_Buffer[0] + 1 ;
		Host_Bootloader_CRC = *((uint32_t *)((Bootloader_Host_Command_Buffer +
Host_Bootloader_Command_Packet_length) - CRC_TYPE_SIZE_BYTE));
		if(CRC_VERIFICATION_PASSED    ==    Bootloader_CRC_Verification((uint8_t
*)&Host_Command_Buffer[0]      ,      Host_Bootloader_Command_Packet_length-4      ,
Host_Bootloader_CRC))
		{
		Bootloader_Send_ACK(1);
			Protection_level = CBL_STM32F401RCT6_Get_Protection_Level();
			Bootloader_Send_Data_To_Host((uint8_t *)&Protection_level , 2);
		}
		else
		{
			Bootloader_Send_NACK();
		}
	}
```

This Helper Function Is Used TO Read The Protection Level Of The Memory (Flash Memory)

```
static uint8_t CBL_STM32F401RCT6_Get_Protection_Level(void)
	{
		HAL_StatusTypeDef Hal_GPL_Status = HAL_ERROR;
		FLASH_OBProgramInitTypeDef Flash_Protection_Level ;
		uint8_t Protection_Level = 0;
		HAL_FLASHEx_OBProgram( &Flash_Protection_Level );
		Protection_Level= ((uint8_t)Flash_Protection_Level.RDPLevel);
		return Protection_Level;
	}
```

Function: `Bootloader_Read_Protection_Level`

This function handles a request from a host device to read the protection level of the flash memory. It verifies the integrity of the received command using CRC and then retrieves the protection level if the verification passes.

4.7.4.1 Explanation of Function

This function processes a request from a host device to read the flash memory protection level of the microcontroller. It follows these steps:

1. **Extract Command Data**: Determines the length of the command packet and extracts the CRC value from the received data.

2. **CRC Verification**: Verifies the integrity of the command using CRC. If the verification fails, it sends a negative acknowledgment (NACK) to the host.

3. **Retrieve and Send Protection Level**: If the CRC check passes, the function retrieves the protection level using a helper function, sends an acknowledgment (ACK), and then sends the protection level back to the host.

`CBL_STM32F401RCT6_Get_Protection_Level`

This helper function is responsible for reading the current protection level of the flash memory in the STM32F401RCT6 microcontroller. It performs the following:

1. **Reads Protection Level**: Utilizes the Hardware Abstraction Layer (HAL) to access the flash memory protection settings.

2. **Returns the Protection Level**: Extracts the protection level from the HAL structure and returns it.

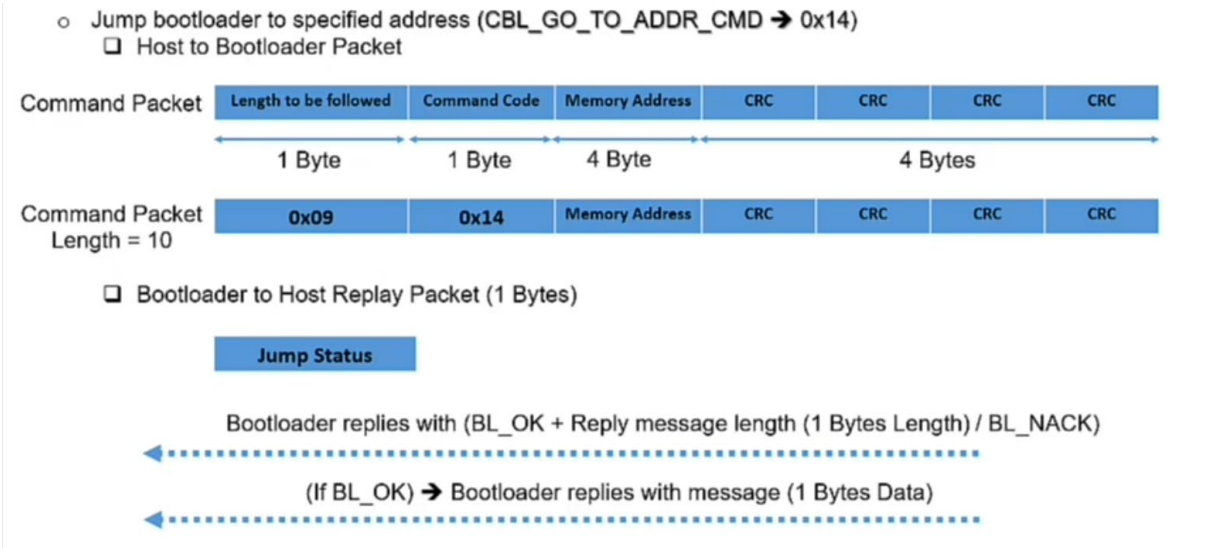
4.7.4.2 Summary

- **`Bootloader_Read_Protection_Level`**: Manages host requests to read the flash memory protection level, verifying the command with CRC and returning the protection level if the check passes.
- **CBL_STM32F401RCT6_Get_Protection_Level**: Reads and returns the flash memory protection level using HAL functions.

These functions work together to securely provide the host with the microcontroller's flash memory protection status, ensuring the integrity of the communication and data.

4.7.5 jump to address

This Function Is Used TO Jump To Specific Adress In The Memory Which Host Want




```
static void Bootloader_Jump_To_Address(uint8_t *Host_Command_Buffer)
	{
		uint16_t Host_Bootloader_Command_Packet_length = 0 ;
		uint32_t Host_Bootloader_CRC = 0 ;
		uint32_t Bootloader_Host_Jump_Address = 0 ;
		uint8_t Address_Check = ADDRESS_IS_INVALID ;

#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
	Bootloader_Print_message("Jump bootloader to specified address \r\n");
#endif
```

```
            Host_Bootloader_Command_Packet_length                                    =
Bootloader_Host_Command_Buffer[0] + 1 ;
            Host_Bootloader_CRC = *((uint32_t *)((Bootloader_Host_Command_Buffer +
Host_Bootloader_Command_Packet_length) - CRC_TYPE_SIZE_BYTE));

            if(CRC_VERIFICATION_PASSED   ==   Bootloader_CRC_Verification((uint8_t
*)&Host_Command_Buffer[0]      ,      Host_Bootloader_Command_Packet_length-4      ,
Host_Bootloader_CRC))
            {
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
            Bootloader_Print_message("CRC Verification Passed \r\n");
#endif
                  Bootloader_Send_ACK(1);
                  Bootloader_Host_Jump_Address           =           *((uint32_t
*)&Host_Command_Buffer[2] );
                  Address_Check                                                    =
Host_Address_Verification(Bootloader_Host_Jump_Address);
                  if(ADDRESS_IS_VALID == Address_Check )
                  {
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                  Bootloader_Print_message("Address verification succeeded \r\n");
#endif
                        Bootloader_Send_Data_To_Host((uint8_t  *)&Address_Check  ,
1);
                        PTR_JUMP                          Jump_To_Choosen_Address
=(PTR_JUMP)(Bootloader_Host_Jump_Address + 1);
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                  Bootloader_Print_message("Jump     to     :     0x%X     \r\n",
Jump_To_Choosen_Address);
#endif
                        Jump_To_Choosen_Address();
                  }
            }
            else
            {
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
            Bootloader_Print_message("CRC Verification Failed \r\n");
#endif
                  Bootloader_Send_Data_To_Host((uint8_t *)&Address_Check , 1);
                  Bootloader_Send_NACK();
            }

      }
```

4.7.5.1 Explanation of Bootloader_Jump_To_Address Function

The **`Bootloader_Jump_To_Address`** function is designed to handle a request from a host device to jump to a specified memory address. This is typically used to transfer control to a new firmware or application that resides at a specific address in memory. The function ensures the integrity of the request using CRC verification and validates the specified address before performing the jump.

Key Steps

1. Local Variables:
   - **`Host_Bootloader_Command_Packet_length`**: Stores the length of the received command packet.
   - **`Host_Bootloader_CRC`**: Stores the CRC value extracted from the received command packet.
   - **`Bootloader_Host_Jump_Address`**: Stores the address to which the bootloader is instructed to jump.
   - **`Address_Check`**: Stores the result of the address validation (valid or invalid).
2. Debug Message:
   - If debugging is enabled, it prints a message indicating that the bootloader received a jump-to-address command.
3. Extract Command Packet Length and CRC:
   - Determines the length of the command packet and extracts the CRC value from the end of the packet.
4. CRC Verification:
   - Verifies the integrity of the command data using CRC. If the CRC check fails, it sends a negative acknowledgment (NACK) and the result of the address check to the host.
   - If the CRC check passes, it proceeds to the next steps.
5. Send Acknowledgment:
   - Sends an acknowledgment (ACK) to the host indicating that the CRC check passed.
6. Extract and Validate Jump Address:
   - Extracts the jump address from the command buffer.
   - Validates the extracted address using the **`Host_Address_Verification`** function.

7. Address Verification:
   - If the address is valid:

     Sends the result of the address check to the host.

     Prepares to jump to the specified address.

     Prints a debug message indicating the jump address.

     Jumps to the specified address, transferring control to the new location.
   - If the address is invalid:

     Sends the result of the address check and a NACK to the host.

4.7.5.2 Summary

- **Purpose**: Handles requests to jump to a specified memory address, ensuring command integrity and address validity.

- **CRC Verification**: Ensures the command has not been corrupted.

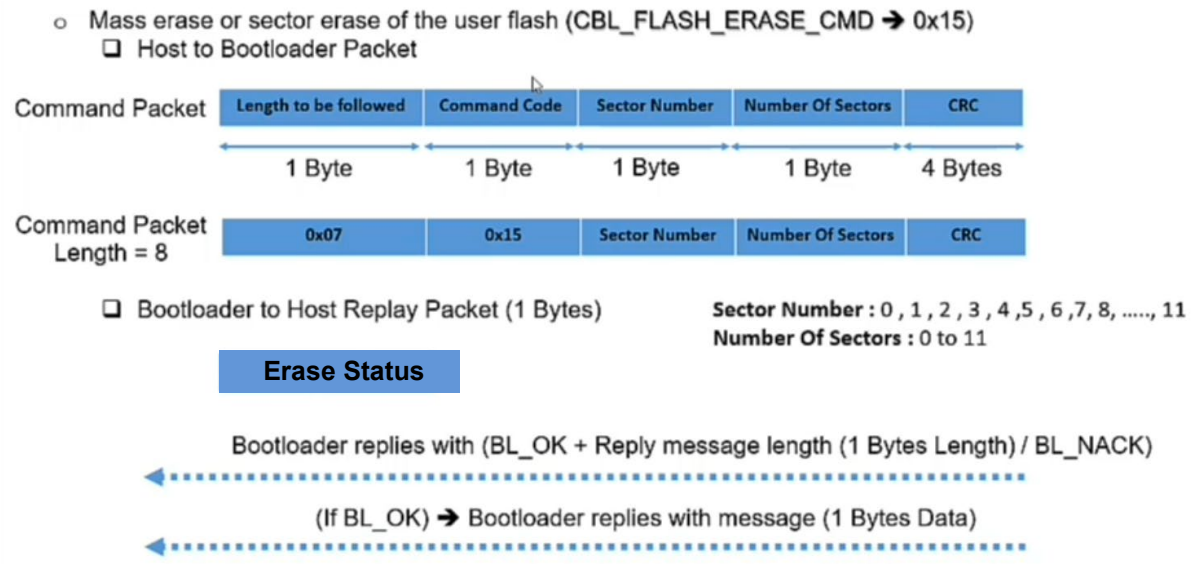
- **Address Validation**: Checks if the specified address is valid.

- **Jump to Address**: If valid, transfers control to the specified address, enabling the execution of new firmware or application code.

This function ensures secure and reliable control transfer within the bootloader, maintaining system integrity during the transition.

4.7.6 Erase Flash

This Function Is Used TO Perform Flash Erase




```
static void Bootloader_Erase_Flash(uint8_t *Host_Command_Buffer)
	{
		uint16_t Host_Bootloader_Command_Packet_length = 0 ;
		uint32_t Host_Bootloader_CRC = 0 ;
    uint8_t ERASE_STATE = 0 ;

#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
	Bootloader_Print_message("Mass erase or sector erase of the user flash \r\n");
#endif

		Host_Bootloader_Command_Packet_length                                    =
Bootloader_Host_Command_Buffer[0] + 1 ;
		Host_Bootloader_CRC = *((uint32_t *)((Bootloader_Host_Command_Buffer +
Host_Bootloader_Command_Packet_length) - CRC_TYPE_SIZE_BYTE));

		if(CRC_VERIFICATION_PASSED    ==    Bootloader_CRC_Verification((uint8_t
*)&Host_Command_Buffer[0]      ,      Host_Bootloader_Command_Packet_length-4      ,
Host_Bootloader_CRC))
		{
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
		Bootloader_Print_message("CRC Verification Passed \r\n");
#endif
			Bootloader_Send_ACK(1);
		  ERASE_STATE =     Perform_Flash_Erase(Host_Command_Buffer[2]       ,
Host_Command_Buffer[3]);
			if(SUCCESSFUL_ERASE == ERASE_STATE)
			{
				Bootloader_Send_Data_To_Host((uint8_t   *)&ERASE_STATE   ,
1);
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
			Bootloader_Print_message("Successful Erase \r\n");
#endif
			}
			else
			{
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
			Bootloader_Print_message("Erase request failed !!\r\n");
#endif
				Bootloader_Send_Data_To_Host((uint8_t   *)&ERASE_STATE   ,
1);
```

```
                    }
                }
                else
                {
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                Bootloader_Print_message("CRC Verification Failed \r\n");
#endif
                    Bootloader_Send_NACK();
                }
        }
```

4.7.6.1 Explanation of Bootloader_Erase_Flash Function

The **`Bootloader_Erase_Flash`** function handles a request from a host device to erase parts or all of the user flash memory. It ensures the integrity of the command using CRC verification and performs the erase operation if the verification passes.

Key Steps

1. Local Variables:
   - **`Host_Bootloader_Command_Packet_length`**: Stores the length of the received command packet.
   - **`Host_Bootloader_CRC`**: Stores the CRC value extracted from the received command packet.
   - **`ERASE_STATE`**: Stores the result of the erase operation (successful or failed).
2. Debug Message:
   - If debugging is enabled, it prints a message indicating that the bootloader received an erase command.
3. Extract Command Packet Length and CRC:
   - Determines the length of the command packet from the first byte of the buffer plus one.
   - Extracts the CRC value from the end of the command packet.
4. CRC Verification:
   - Verifies the integrity of the command data using CRC. If the CRC check fails, it sends a negative acknowledgment (NACK) to the host.
   - If the CRC check passes, it proceeds to the next steps.

5. Send Acknowledgment:
    - Sends an acknowledgment (ACK) to the host indicating that the CRC check passed.
6. Perform Erase Operation:
    - Calls the `Perform_Flash_Erase` function with the parameters extracted from the command buffer (likely indicating which sectors to erase or if a mass erase is required).
    - Stores the result of the erase operation in `ERASE_STATE`.
7. Send Erase Result:
    - If the erase operation is successful, it sends the ERASE_STATE to the host and prints a debug message indicating success.
    - If the erase operation fails, it sends the ERASE_STATE to the host and prints a debug message indicating failure.

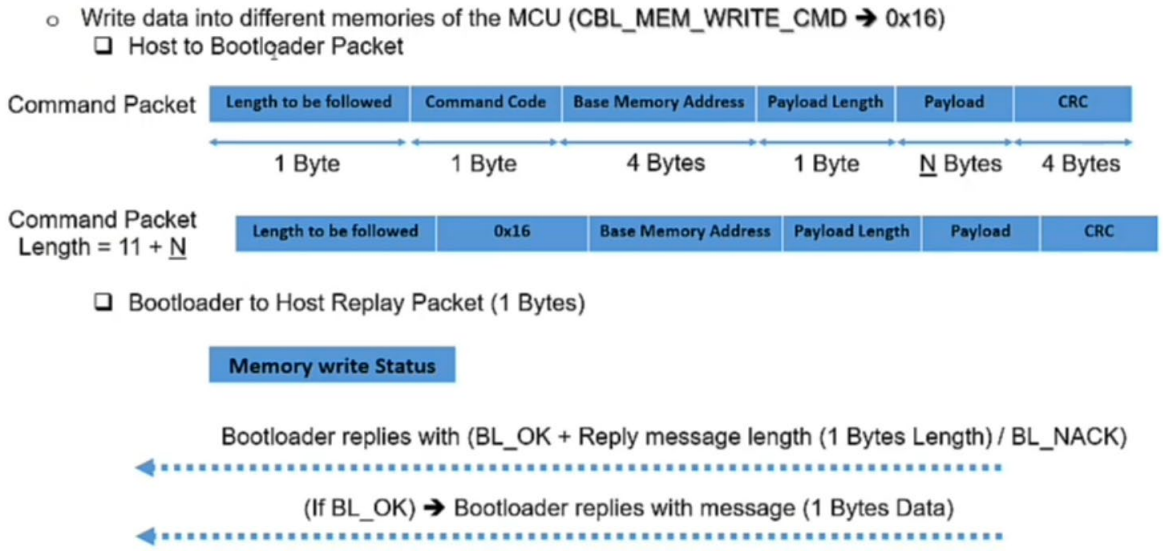
4.7.6.2 Summary

- **Purpose**: Handles requests to erase parts or all of the user flash memory, ensuring command integrity and providing feedback on the success of the operation.
- **CRC Verification**: Ensures the command has not been corrupted.
- **Erase Operation**: Calls a helper function to perform the actual erase, passing parameters from the command.
- **Feedback**: Sends the result of the erase operation (successful or failed) back to the host.

This function ensures secure and reliable execution of flash memory erase operations, maintaining system integrity during the process.

4.7.7 Memory Write Function

This Function Is Used TO Write Data In Different Memories In MCU




```
static void Bootloader_Memory_Write(uint8_t *Host_Command_Buffer)
        {
                uint16_t Host_Bootloader_Command_Packet_length = 0 ;
                uint32_t Host_Bootloader_CRC = 0 ;
                uint32_t Host_Write_Address = 0 ;
                uint32_t Host_Payload_Length = 0 ;
                uint8_t Address_Check = ADDRESS_IS_INVALID ;
                uint8_t Flash_Pay_Load_Write_State = FLASH_PAYLOAD_WRITE_FAILED ;

#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
        Bootloader_Print_message("Write data into different memories of the MCU \r\n");
#endif

                Host_Bootloader_Command_Packet_length = Bootloader_Host_Command_Buffer[0] + 1 ;
                Host_Bootloader_CRC    =    *((uint32_t    *)((Bootloader_Host_Command_Buffer    +
Host_Bootloader_Command_Packet_length) - CRC_TYPE_SIZE_BYTE));

                if(CRC_VERIFICATION_PASSED        ==        Bootloader_CRC_Verification((uint8_t
*)&Host_Command_Buffer[0] , Host_Bootloader_Command_Packet_length-4 , Host_Bootloader_CRC))
                {
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                Bootloader_Print_message("CRC Verification Passed \r\n");
#endif

                  Bootloader_Send_ACK(1);
                  Host_Write_Address = *((uint32_t *)(&Host_Command_Buffer[2])) ;

#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                Bootloader_Print_message("HOST_Address = 0x%X \r\n", Host_Write_Address);
#endif

            Host_Payload_Length = Host_Command_Buffer[6] ;
                        Address_Check = Host_Address_Verification(Host_Write_Address);
                         if(ADDRESS_IS_VALID == Address_Check)
                         {
                                 Flash_Pay_Load_Write_State                                        =
Flash Memory Write Payload((uint8 t    *)&Host Command Buffer[7]    ,    Host Write Address    ,
Host_Payload_Length );
                                 if(Flash_Pay_Load_Write_State == FLASH_PAYLOAD_WRITE_PASSED)
                                 {

Bootloader_Send_Data_To_Host((uint8_t *)&Flash_Pay_Load_Write_State , 1);
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                                Bootloader_Print_message("Payload Valid \r\n");
```

```
#endif

                                }
                                else
                                {

Bootloader_Send_Data_To_Host((uint8_t *)&Flash_Pay_Load_Write_State , 1);
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                                Bootloader_Print_message("Payload InValid \r\n");
#endif

                                }
                        }
                        else
                        {
                                Bootloader_Send_Data_To_Host((uint8_t *)&Address_Check , 1);
                        }
                }
                else
                {
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                Bootloader_Print_message("CRC Verification Failed \r\n");
#endif
                        Bootloader_Send_NACK();
                }
```

4.7.7.1 Explanation of Bootloader_Memory_Write Function

The **`Bootloader_Memory_Write`** function handles a request from a host device to write data into various memory locations of the MCU (Microcontroller Unit). It verifies the integrity of the command using CRC and performs the write operation if the verification and address validation pass.

Key Steps

1. Local Variables:
   - **`Host_Bootloader_Command_Packet_length`**: Stores the length of the received command packet.
   - **`Host_Bootloader_CRC`**: Stores the CRC value extracted from the received command packet.
   - **`Host_Write_Address`**: Stores the memory address where data is to be written.
   - **`Host_Payload_Length`**: Stores the length of the payload to be written.

- **`Address_Check`**: Stores the result of the address validation (valid or invalid).
- **`Flash_Pay_Load_Write_State`**: Stores the result of the flash memory write operation (successful or failed).

2. Debug Message:
   - If debugging is enabled, it prints a message indicating that the bootloader received a memory write command.

3. Extract Command Packet Length and CRC:
   - Determines the length of the command packet from the first byte of the buffer plus one.
   - Extracts the CRC value from the end of the command packet.

4. CRC Verification:
   - Verifies the integrity of the command data using CRC. If the CRC check fails, it sends a negative acknowledgment (NACK) to the host.
   - If the CRC check passes, it proceeds to the next steps.

5. Send Acknowledgment:
   - Sends an acknowledgment (ACK) to the host indicating that the CRC check passed.

6. Extract and Validate Write Address:
   - Extracts the write address from the command buffer.
   - Validates the extracted address using the **`Host_Address_Verification`** function.

7. Write Data to Memory:

- If the address is valid:

  Calls the `Flash_Memory_Write_Payload` function to write the payload data to the specified address.

  Sends the result of the write operation (successful or failed) to the host.

  If debugging is enabled, prints messages indicating whether the payload is valid or invalid.

- If the address is invalid:

  Sends the address validation result to the host.

4.7.7.2 Summary

- **Purpose**: Handles requests to write data into various memory locations of the MCU, ensuring command integrity and providing feedback on the success of the operation.

- **CRC Verification**: Ensures the command has not been corrupted.

- **Address Validation**: Checks if the specified write address is valid.

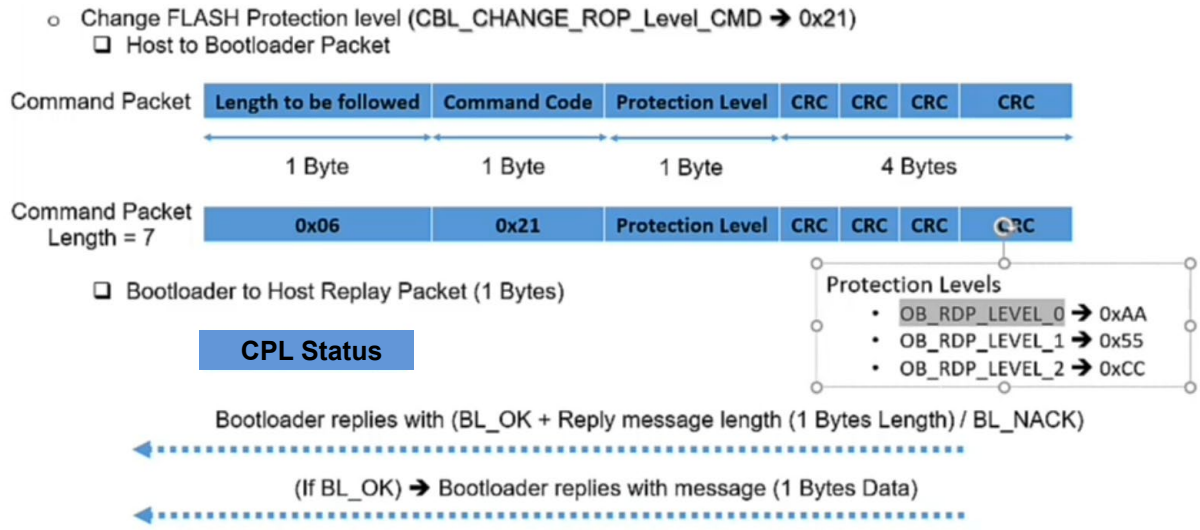
- **Write Operation**: Writes the payload to the specified memory address if the address is valid.

- **Feedback**: Sends the result of the write operation (successful or failed) back to the host.

This function ensures secure and reliable execution of memory write operations, maintaining system integrity during the process.

4.7.8 Change Read Protection level Function

This Function Is Used TO hange Read Protection Level Of MCU




```
static void Bootloader_Change_Read_Protection_Level(uint8_t *Host_Command_Buffer)
        {
                uint16_t Host_Bootloader_Command_Packet_length = 0 ;
                uint32_t Host_Bootloader_CRC = 0 ;
                uint8_t Change_Status = ROP_LEVEL_CHANGE_INVALID;
                uint8_t Host_Rop_Level = 0;

#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
        Bootloader_Print_message("Change read protection level of the user flash \r\n");
#endif

                Host_Bootloader_Command_Packet_length = Bootloader_Host_Command_Buffer[0] + 1 ;
                Host_Bootloader_CRC      =      *((uint32_t      *)((Bootloader_Host_Command_Buffer      +
Host_Bootloader_Command_Packet_length) - CRC_TYPE_SIZE_BYTE));
                if(CRC_VERIFICATION_PASSED             ==             Bootloader_CRC_Verification((uint8_t
*)&Host_Command_Buffer[0] , Host_Bootloader_Command_Packet_length-4 , Host_Bootloader_CRC))
                {
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                Bootloader_Print_message("CRC Verification Passed \r\n");
#endif
                Bootloader_Send_ACK(1);
                        Host_Rop_Level = Host_Command_Buffer[2];
                        if (2 == Host_Rop_Level)
                        {
                                Change_Status = ROP_LEVEL_CHANGE_INVALID;
                        }
                        else
                        {
                                if(0 == Host_Rop_Level)
                                {
                                        Host_Rop_Level = 0XAA;
                                }
                                else if(1 == Host_Rop_Level)
                                {
                                        Host_Rop_Level = 0X55;
                                }
                                        Change_Status = Change_Read_Protection_Level(Host_Rop_Level);

                }
                                  Bootloader_Send_Data_To_Host((uint8_t *)&Change_Status , 1);
         }
        else
                {
#if (BL_DEBUG_ENABLE == DEBUG_INFO_ENABLE)
                Bootloader_Print_message("CRC Verification Failed \r\n");
#endif
                        Bootloader_Send_NACK();
                }
        }
```

4.7.8.1 Explanation of Bootloader_Change_Read_Protection_Level Function

The `Bootloader_Change_Read_Protection_Level` function manages requests from a host device to change the read protection level of the user flash memory on the MCU (Microcontroller Unit). It verifies the command integrity using CRC, interprets the desired read protection level from the command, and performs the necessary operation.

Key Steps

1. Local Variables:
   - **Host_Bootloader_Command_Packet_length**: Stores the length of the received command packet.
   - **Host_Bootloader_CRC**: Stores the CRC value extracted from the received command packet.
   - **Change_Status**: Stores the status of the read protection level change operation.
   - **Host_Rop_Level**: Stores the desired read protection level received from the host.

2. Debug Message:
   - If debugging is enabled, it prints a message indicating that the bootloader received a command to change the read protection level.

3. Extract Command Packet Length and CRC:
   - Determines the length of the command packet from the first byte of the buffer plus one.
   - Extracts the CRC value from the end of the command packet.

4. CRC Verification:
   - Verifies the integrity of the command data using CRC. If the CRC check fails, it sends a negative acknowledgment (NACK) to the host.
   - If the CRC check passes, it proceeds to the next steps.

5. Send Acknowledgment:
   - Sends an acknowledgment (ACK) to the host indicating that the CRC check passed.

6. Extract and Validate Read Protection Level:
   - Extracts the desired read protection level from the command buffer (**`Host_Rop_Level`**).
   - Depending on the value of **`Host_Rop_Level`**, prepares the appropriate value (**`0xAA`** or **`0x55`**) to change the read protection level.

7. Change Read Protection Level:
   - Calls **`Change_Read_Protection_Level`** function with the appropriate value (**`0xAA`** or **`0x55`**) to initiate the read protection level change process.

8. Send Operation Status to Host:
   - Sends the result of the read protection level change operation (**`Change_Status`**) back to the host.

4.7.8.2 Summary

- **Purpose**: Manages requests to change the read protection level of the user flash memory on the MCU.

- **CRC Verification**: Ensures the command integrity before proceeding with the operation.

- **Read Protection Level Interpretation**: Interprets the desired read protection level and prepares the necessary value (`0xAA` or `0x55`) accordingly.

- **Change Operation**: Initiates the read protection level change operation using `Change_Read_Protection_Level` function.

- **Feedback**: Sends the status of the operation (success or failure) back to the host.

This function ensures secure and reliable execution of read protection level changes, maintaining system integrity during the process.

CHAPTER 5

ADAS

GRADUATION PROJECT 2024

5 Chapter 5: ADAS

5.1 Introduction:-

Advanced Driver Assistance Systems (ADAS) are a collection of technologies in cars designed to improve safety and make driving easier. These systems use a combination of sensors, cameras, radar, and other technologies to monitor the environment around the vehicle and assist the driver in various ways.

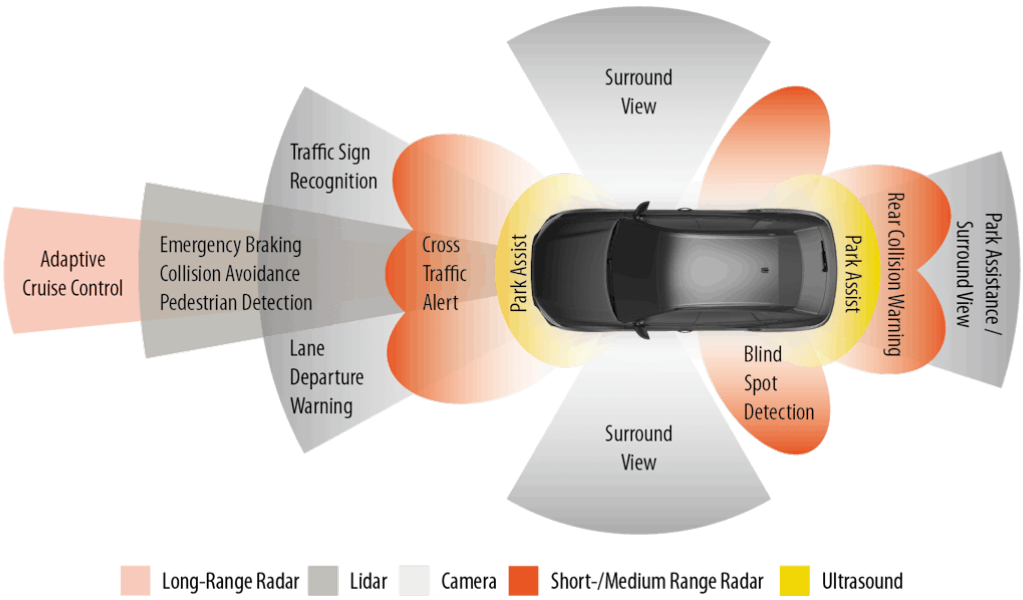


Figure 5-1: Advanced Driver Assistance Systems

5.2 Key Features of ADAS:

1. **Sensors and Cameras**: ADAS uses sensors and cameras to gather information about the vehicle's surroundings. These can detect other vehicles, pedestrians, road signs, and obstacles.

2. **Safety Alerts**: ADAS provides warnings to the driver about potential hazards. For example, if a car is in your blind spot when you're trying to change lanes, ADAS will alert you.

3. **Automatic Actions**: Some ADAS features can take control of the vehicle to avoid accidents. For instance, automatic emergency braking can stop the car if it detects an imminent collision.

4. **Driver Assistance**: ADAS includes features like adaptive cruise control, which maintains a set distance from the car in front of you, and lane-keeping assistance, which helps keep the car in its lane.

5. **Comfort and Convenience**: ADAS also improves driving comfort with features like automated parking, where the car can park itself, and traffic sign recognition, which reads and displays speed limits and other signs.

5.3 How ADAS Works:

1. **Data Collection**: Sensors and cameras collect real-time data about the vehicle's surroundings.

2. **Processing**: This data is processed by embedded systems in the car, which are specialized computers that handle specific functions.

3. **Decision Making**: The system analyzes the data to make decisions. For example, it can determine if there is a risk of collision and decide whether to warn the driver or take action.

4. **Action**: Based on the decision, the system can either alert the driver or control the vehicle to avoid danger.

5.4 Benefits of ADAS:

1. **Increased Safety**: By providing real-time assistance, ADAS helps prevent accidents and reduces the severity of those that do occur.

2. **Reduced Driver Fatigue**: Features like adaptive cruise control and lane-keeping assistance make long drives less tiring.

3. **Enhanced Driving Experience**: Automated parking and traffic sign recognition add convenience and ease to the driving experience.

Overall, ADAS represents a significant step towards the development of fully autonomous vehicles. It not only enhances safety but also improves the overall driving experience by making it more comfortable and convenient.

5.5 First Test Case Acc

Adaptive Cruise Control (ACC) is a crucial feature of Advanced Driver Assistance Systems (ADAS) in modern cars. Here's a simplified explanation of how ACC works within a car's embedded system:

5.5.1 Purpose of ACC:

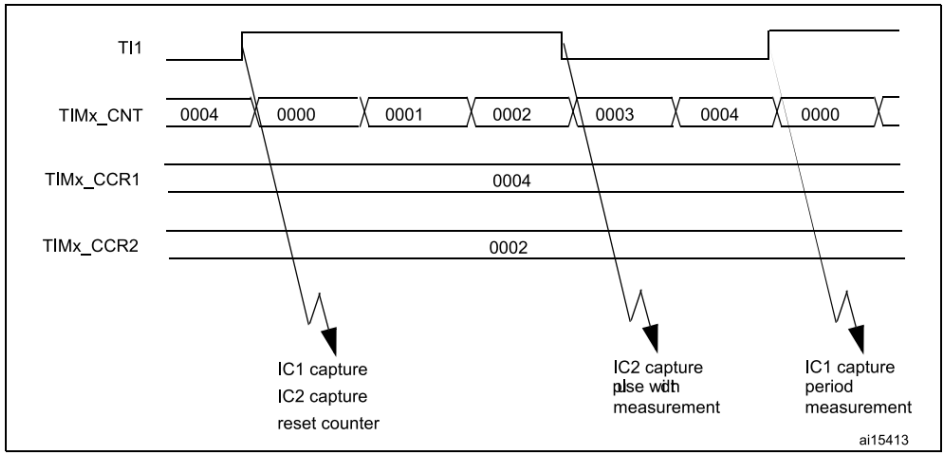
- ACC helps maintain a safe distance between your car and the vehicle in front of you. It adjusts your car's speed automatically, so you don't have to keep accelerating and braking manually.

5.5.2 How It Works:

- **Detection**: Ultrasonic sensors detect the speed and distance of the car in front.

- **Processing**: The ECU processes this information and calculates whether your car needs to slow down or can maintain its current speed.

- **Action**: If the car in front slows down, the ECU signals the throttle to reduce speed or the brakes to activate gently. If the road ahead is clear, the ECU maintains or increases the car's speed to the preset limit.



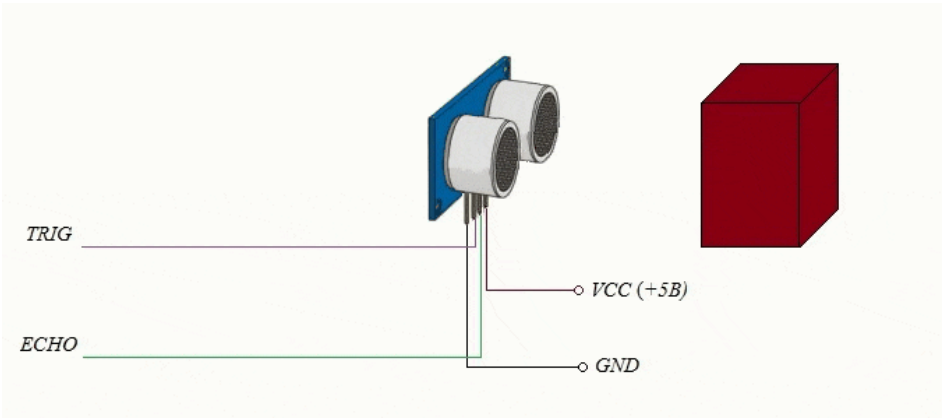


Figure 5-2: Adaptive Cruise Control

5.5.3 Benefits:

- **Safety**: ACC helps prevent rear-end collisions by maintaining a safe following distance.
- **Convenience**: It reduces the need for constant acceleration and braking, making long drives more comfortable.

5.5.4 Operation Modes:

- **Cruise Mode**: Maintains a constant speed set by the driver.
- **Adaptive Mode**: Adjusts speed automatically based on the traffic conditions ahead.

In essence, ACC enhances driving safety and comfort by intelligently managing the car's speed and maintaining a safe distance from other vehicles, all controlled by the embedded system's sensors and processors.

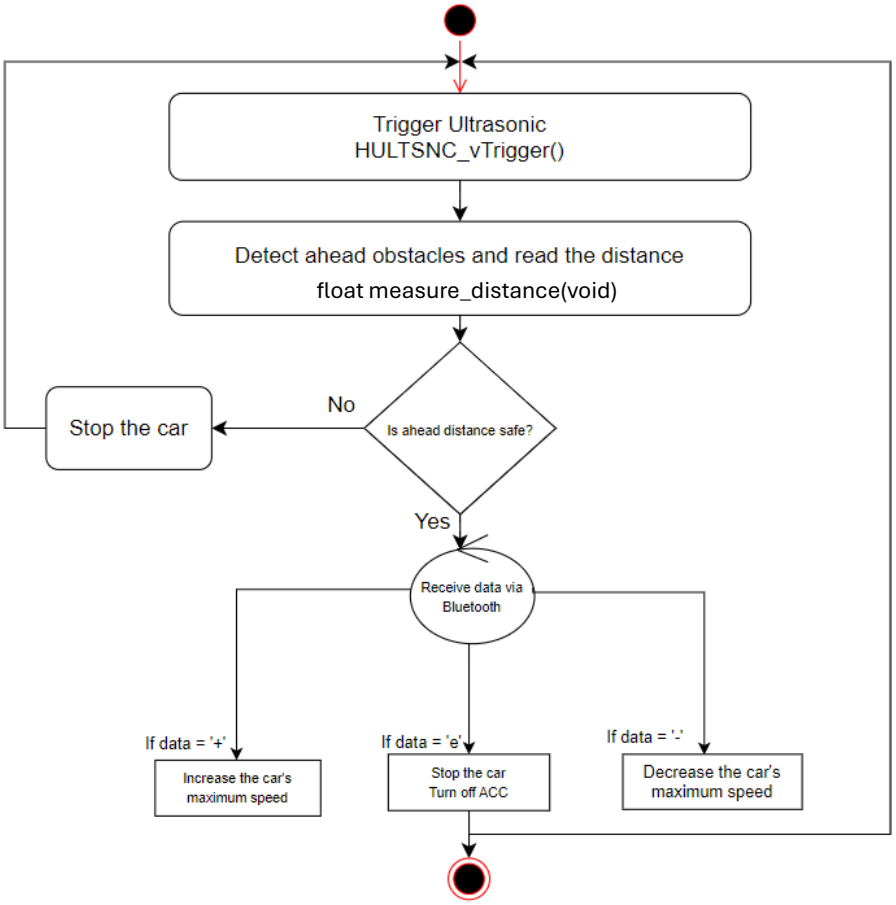


Figure 5-3: Adaptive Cruise Control Flowchart

5.5.5 We will explain ultrasonic sensor code first.

The code in (Figure 5-4) is used to calculate distance using ultrasonic sensor

```
// Function to measure the distance using the ultrasonic sensor
float measure_distance(void) {
    uint32_t pulse_width;
    float distance;

    // Trigger the ultrasonic sensor
    HAL_GPIO_WritePin(GPIO_PORT, TRIG_PIN, GPIO_PIN_SET);
    delay_us(10); // Wait for 10 microseconds
    HAL_GPIO_WritePin(GPIO_PORT, TRIG_PIN, GPIO_PIN_RESET); // Reset the trigger signal

    // Wait for the echo signal to start
    while (HAL_GPIO_ReadPin(GPIO_PORT, ECHO_PIN) == GPIO_PIN_RESET);

    // Measure the pulse width of the echo signal
    __HAL_TIM_SET_COUNTER(&htim4, 0); // Reset the timer counter
    while (HAL_GPIO_ReadPin(GPIO_PORT, ECHO_PIN) == GPIO_PIN_SET); // Wait until the echo signal goes low
    pulse_width = __HAL_TIM_GET_COUNTER(&htim4); // Get the pulse width

    // Calculate distance in centimeters
    distance = (float)pulse_width * 0.017; // Speed of sound = 343 m/s (34.3 cm/ms), divide by 2 because sound goes to t

    return distance;
}
```

Figure 5-4: calculate distance

5.5.5.1 Function Declaration

```
float measure_distance(void) {
```

This line declares a function named `measure_distance` that returns a floating-point number (the distance).

5.5.5.2 Variable Declaration

```
uint32_t pulse_width;
float distance;
```

These lines declare two variables **`pulse_width`** to store the time the echo takes to return, and distance to store the calculated distance.

5.5.5.3 Trigger the Ultrasonic Sensor

```
HAL_GPIO_WritePin(GPIO_PORT, TRIG_PIN, GPIO_PIN_SET);
delay_us(10); // Wait for 10 microseconds
HAL_GPIO_WritePin(GPIO_PORT, TRIG_PIN,
GPIO_PIN_RESET);
```

- **Trigger ON**: The first line sets the TRIG_PIN high to send a trigger pulse.
- **Wait**: The delay_us(10) function waits for 10 microseconds to ensure the pulse is long enough.
- **Trigger OFF**: The last line sets the TRIG_PIN low to complete the trigger pulse.

5.5.5.4 Wait for Echo Signal to Start

```
while (HAL_GPIO_ReadPin(GPIO_PORT, ECHO_PIN) == GPIO_PIN_RESET);
```

This line waits until the ECHO_PIN goes high, indicating the sensor has received the echo back.

5.5.5.5 Measure the Pulse Width of the Echo Signal

```
__HAL_TIM_SET_COUNTER(&htim4, 0); // Reset the timer counter
while (HAL_GPIO_ReadPin(GPIO_PORT, ECHO_PIN) == GPIO_PIN_SET); // Wait until the echo signal goes low
pulse_width = __HAL_TIM_GET_COUNTER(&htim4); // Get the pulse width
```

- **Reset Timer**: The first line resets the timer to start counting from zero.
- **Wait for Echo End**: The second line waits until the ECHO_PIN goes low, marking the end of the echo.
- **Capture Pulse Width**: The third line reads the timer value, which represents the duration of the echo pulse.

5.5.5.6 Calculate Distance in Centimeters

```
distance = (float)pulse_width * 0.017; // Speed of sound = 343 m/s (34.3 cm/ms), divide by 2 because sound goes to the object and comes back
```

This line converts the pulse width to distance. The constant 0.017 is derived from the speed of sound (343 meters per second) converted to centimeters per microsecond and divided by 2 to account for the round-trip of the sound wave.

5.5.5.7 Return the Distance

```
return distance;
}
```

The function returns the calculated distance.

5.5.6 Then we will explain ACC main code

The code in (Figure 5-5) is used to read the calculated distance and adjust car speed (motor rotation speed) depending on the distance between the car and other things.

```
  while (1)
  {
    /* USER CODE END WHILE */
      distance = readDistance(); // Read distance from ultrasonic sensor
      adjustSpeed(distance); // Adjust car speed based on distance
            HAL_Delay(100);
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

float readDistance() {
    // Code to read distance from sensors
    float Distance = 0;
    Distance = measure_distance();
    // Return distance value
    return Distance;
}

// Function to adjust car speed based on distance
void adjustSpeed(float distance) {
    uint32_t pulse_width;

    if (distance > SAFE_DISTANCE_THRESHOLD) {
        // If the distance is greater than the safe distance threshold, accelerate
        pulse_width = (uint32_t)((MAX_SPEED / 100.0) * htim3.Init.Period);
    } else {
        // If the distance is less than or equal to the safe distance threshold, decelerate
        pulse_width = (uint32_t)((MIN_SPEED / 100.0) * htim3.Init.Period);
    }

    // Set the PWM pulse width to control motor speed
    motor_control_speed(&htim3, TIM_CHANNEL_1, pulse_width);
}
```

```
Problems  Tasks  Console  Properties
CDT Build Console [ACC]

21:14:26 Build Finished. 0 errors, 0 warnings. (took 2s.805ms)
```

Figure 5-5: read the calculated distance

- In this code we mainly need **three modules** or drivers:
  1) PWM module driver.
  2) Timer module driver.
  3) Ultrasonic sensor module driver.

- Other drivers:
  1) GPIO module driver.

5.5.6.1 Main Loop:

```
while (1) {
    /* USER CODE END WHILE */
    distance = readDistance(); // Read distance from ultrasonic sensor
    adjustSpeed(distance); // Adjust car speed based on distance
    HAL_Delay(100); // delay 100ms after perform this block
}
```

- **`While (1):`** This creates an infinite loop that keeps running as long as the program is on.

- **`distance = readDistance();:`** Calls the `readDistance` function to get the distance measured by the ultrasonic sensor.

- **`adjustSpeed(distance);`**: Calls the `adjustSpeed` function to adjust the car's speed based on the measured distance.

- **`HAL_Delay(100);:`** Pauses the loop for 100 milliseconds before repeating.

5.5.6.2 Read Distance Function

```
float readDistance() {
    // Code to read distance from sensors
    float Distance = 0;
    Distance = measure_distance();
    // Return distance value
    return Distance;
}
```

- **`float readDistance();`** Declares a function that returns a float value representing the distance.
- **`float Distance = 0;`** Initializes a variable `Distance` to store the measured distance.
- **`Distance = measure_distance();`** Calls the `measure_distance` function to get the distance and stores it in `Distance`.
- **`return Distance;`** Returns the measured distance.

5.5.6.3 Adjust Speed Function

```
void adjustSpeed(float distance) {
    uint32_t pulse_width;

    if (distance > SAFE_DISTANCE_THRESHOLD) {
        // If the distance is greater than the safe distance threshold, accelerate
        pulse_width = (uint32_t)((MAX_SPEED / 100.0) * htim3.Init.Period);
    }
 else {
        // If the distance is less than or equal to the safe distance threshold, decelerate
        pulse_width = (uint32_t)((MIN_SPEED / 100.0) * htim3.Init.Period);
    }
    // Set the PWM pulse width to control motor speed
    motor_control_speed(&htim3, TIM_CHANNEL_1, pulse_width);
}
```

- **`void adjustSpeed(float distance);`** Declares a function that takes a float parameter `distance` and returns nothing (void).
- **`uint32_t pulse_width;`:** Declares a variable `pulse_width` to store the PWM (Pulse Width Modulation) pulse width.
- **`if (distance > SAFE_DISTANCE_THRESHOLD);`:** Checks if the measured distance is greater than the safe distance threshold.
  - ✓ pulse_width = (uint32_t)((MAX_SPEED / 100.0) * htim3.Init.Period);: If true, calculates the pulse width for maximum speed.
- **`else`:** If the distance is less than or equal to the safe distance threshold:
  - ✓ pulse_width = (uint32_t)((MIN_SPEED / 100.0) * htim3.Init.Period);: Calculates the pulse width for minimum speed.
- **`motor_control_speed(&htim3, TIM_CHANNEL_1, pulse_width);`:** Sets the PWM pulse width to control the motor speed using the calculated `pulse_width`.

5.6 Second Test Case

second application is Temperature monitoring system in car.

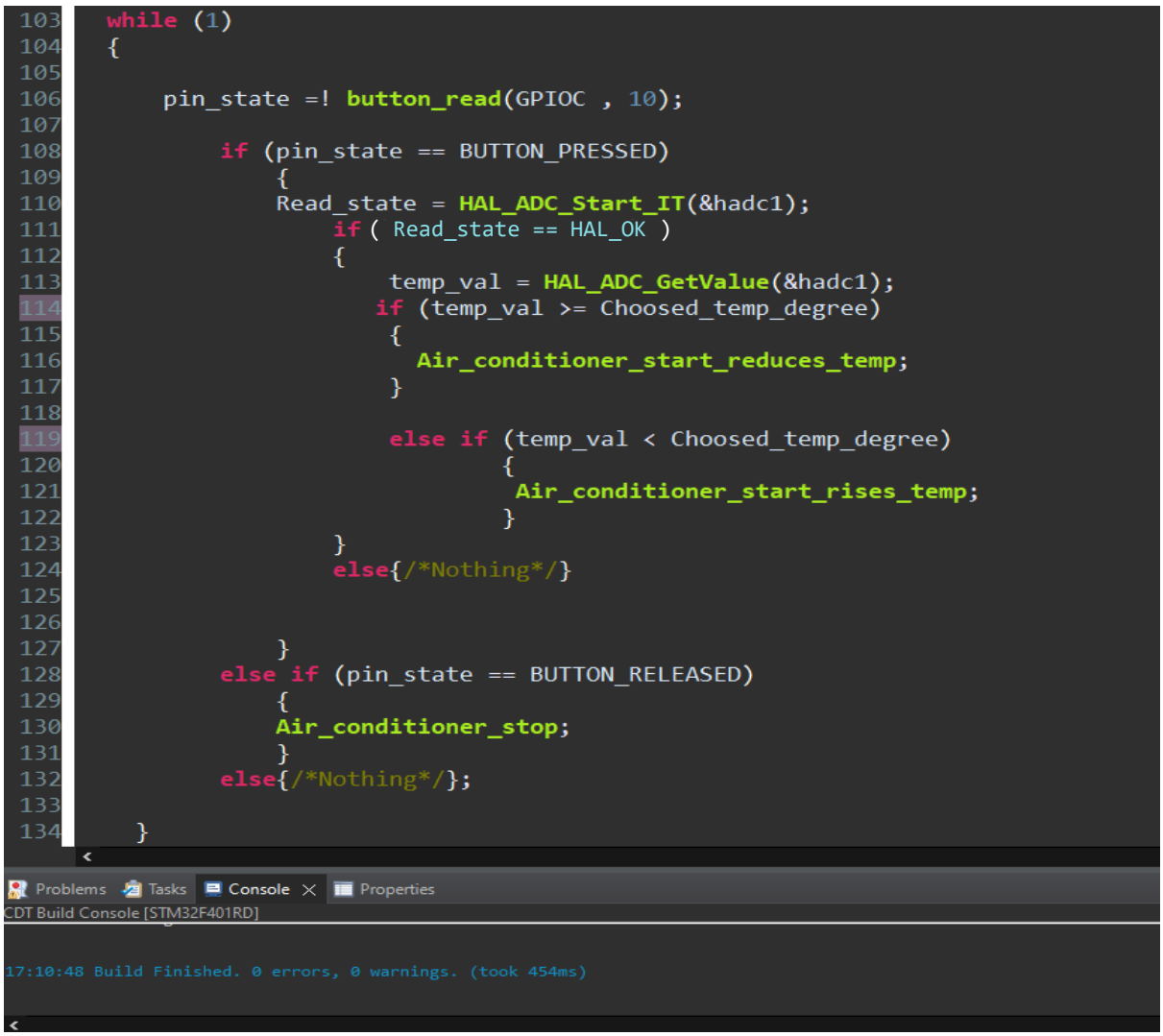


Figure 5-6: Temperature monitoring system in car

- The code in (Figure 5-6) we mainly need **foure modules** or drivers:
  1) Air conditioner based on GPIO module driver.
  2) Button based on GPIO module driver.
  3) ADC module driver.
  4) Motor based on GPIO module driver.

5.6.1 Main Loop

```
while (1) {
    pin_state = !button_read(GPIOC, 10);
```

- **`while (1):`** This creates an infinite loop that keeps running as long as the program is on.
- **`pin_state = !button_read(GPIOC, 10);:`** Reads the state of the button connected to pin 10 on GPIOC. The `!` inverts the result, so if the button is pressed (LOW state), `pin_state` becomes true, and if the button is released (HIGH state), `pin_state` becomes false.

5.6.2 Button Pressed Check

```
if (pin_state == BUTTON_PRESSED) {
    Read_state = HAL_ADC_Start_IT(&hadc1);
```

- **`if (pin_state == BUTTON_PRESSED)`**: Checks if the button is pressed.
- **`Read_state = HAL_ADC_Start_IT(&hadc1);`**: Starts the ADC (Analog-to-Digital Converter) in interrupt mode to read the temperature sensor.

5.6.3 ADC Read Success Check

```
if (Read_state == HAL_OK) {
temp_val = HAL_ADC_GetValue(&hadc1);
```

- **`if (Read_state == HAL_OK):`** Checks if the ADC read was successful (this condition always evaluates to true; might be a placeholder or error).

- **`temp_val = HAL_ADC_GetValue(&hadc1);:`** Gets the temperature value from the ADC.

5.6.4 Temperature-Based Air Conditioner Control

```
if (temp_val >= Choosed_temp_degree) {
        Air_conditioner_start_reduces_temp;
       } else if (temp_val < Choosed_temp_degree)
{
                Air_conditioner_start_rises_temp;
            }
```

- **`if (temp_val >= Choosed_temp_degree):`** If the temperature is equal to or higher than the chosen temperature, the air conditioner is set to reduce the temperature.
- **`Air_conditioner_start_reduces_temp;:`** Starts the air conditioner in cooling mode.
- **`else if (temp_val < Choosed_temp_degree):`** If the temperature is below the chosen temperature, the air conditioner is set to increase the temperature.
- **`Air_conditioner_start_rises_temp;:`** Starts the air conditioner in heating mode.

5.6.5 Handling Unsuccessful ADC Read

```
} else {/*Nothing*/}
```

- **`else {/*Nothing*/}:`** Placeholder for handling unsuccessful ADC read (currently does nothing).

5.6.6 Button Released Check

```
    } else if (pin_state == BUTTON_RELEASED) {
        Air_conditioner_stop;
    } else {/*Nothing*/}
}
```

- else if (pin_state == BUTTON_RELEASED): Checks if the button is released.
- **`Air_conditioner_stop;`**: Stops the air conditioner.
- **`else {/*Nothing*/};`**: Placeholder for any other conditions (currently does nothing).

CHAPTER 6

RTOS

GRADUATION PROJECT 2024

6 Chapter 6: RTOS In Embedded System:

6.1 Introduction: -

6.1.1 What is an RTOS?

An RTOS (Real-Time Operating System) is a type of operating system designed to manage hardware resources and execute tasks within strict timing constraints. Unlike general-purpose operating systems, an RTOS ensures that critical tasks are completed on time, which is essential for applications requiring real-time responses.

6.1.2 Why RTOS in Automotive Systems?

Modern cars are complex systems with various electronic components, collectively called embedded systems. These systems control everything from the engine and brakes to entertainment and navigation. An RTOS in a car's embedded system ensures that critical functions like braking and airbag deployment happen reliably and without delay.

6.2 Key Features of RTOS in Automotive Systems:

1. **Task Management**: An RTOS can manage multiple tasks, prioritizing critical tasks (like brake control) over less critical ones (like adjusting seat position).
2. **Interrupt Handling**: RTOS handles interrupts efficiently, ensuring immediate response to important events like collision detection.
3. **Deterministic Behavior**: An RTOS guarantees predictable behavior, meaning it knows how long a task will take to complete, which is crucial for safety-critical applications.
4. **Resource Allocation**: It efficiently allocates CPU, memory, and other resources to ensure smooth operation of all system components.

6.3 Benefits of RTOS in Cars:

1. **Safety**: Ensures timely execution of safety-critical tasks such as airbag deployment and anti-lock braking.
2. **Reliability**: Provides consistent performance, reducing the risk of system failures.
3. **Efficiency**: Manages resources effectively, improving the overall performance of the car's embedded systems.
4. **Scalability**: Supports the integration of new features and technologies without compromising existing functionalities.

6.4 Examples of RTOS in Automotive Applications:

- **Engine Control Units (ECU)**: Manages engine performance, fuel efficiency, and emissions.
- **Advanced Driver Assistance Systems (ADAS)**: Enhances safety by assisting with tasks like lane-keeping, adaptive cruise control, and emergency braking.
- **Infotainment Systems**: Manages multimedia, navigation, and connectivity features, ensuring seamless user experience.

6.5 Challenges and Considerations:

1. **Complexity**: Designing an RTOS-based system requires careful planning to handle multiple tasks and interrupts efficiently.
2. **Cost**: Implementing an RTOS can be expensive due to the need for specialized hardware and software.
3. **Maintenance**: Keeping the system updated and ensuring compatibility with new technologies can be challenging.

6.6 ACC functions as RTOS

We will use our ACC functions as RTOS Task to be used in critical situation shown in (Figure 6-1).

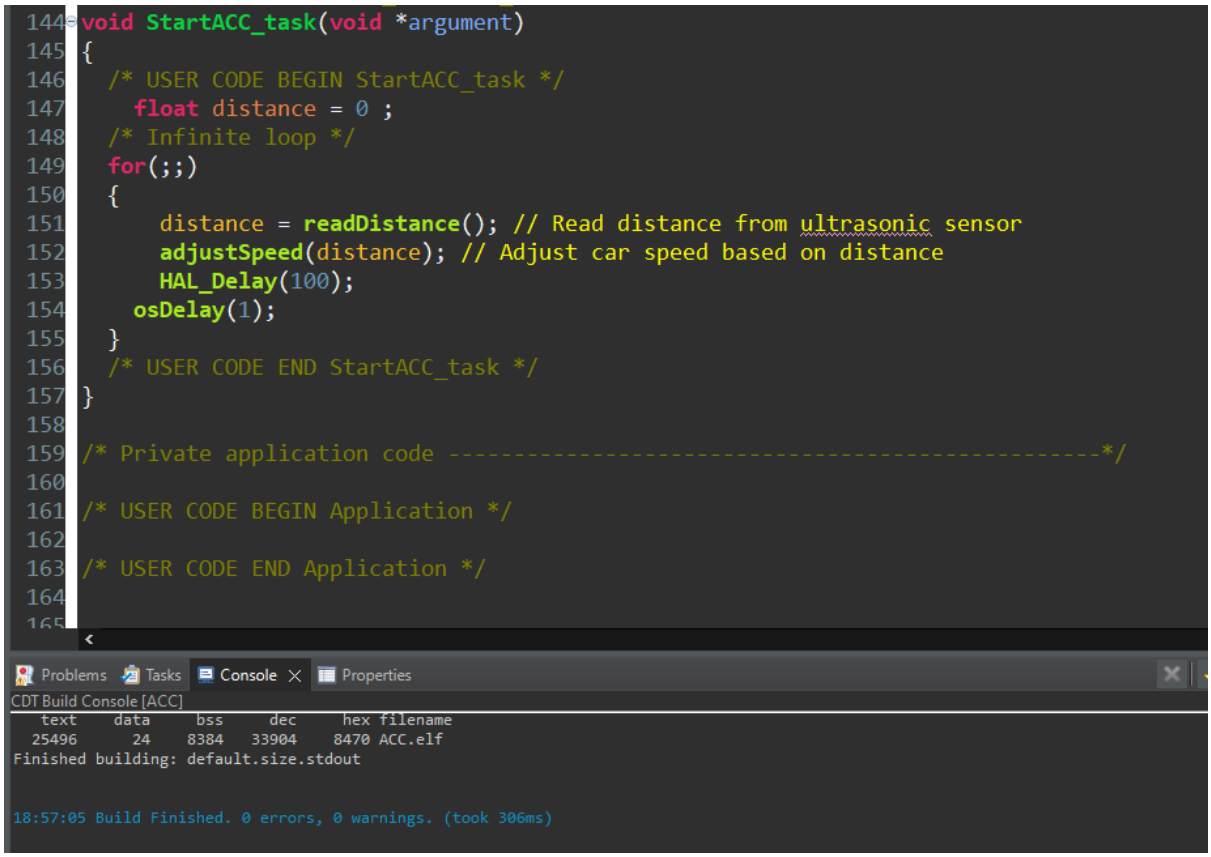


Figure 6-1: ACC functions as RTOS

```
void StartACC_task(void *argument)
{
  /* USER CODE BEGIN StartACC_task */
  float distance = 0;
  /* Infinite loop */
  for(;;)
  {
    distance = readDistance(); // Read distance from
ultrasonic sensor
    adjustSpeed(distance); // Adjust car speed based
on distance
    HAL_Delay(100); // Wait for 100 milliseconds
    osDelay(1); // Yield control to other tasks for 1
time unit
  }
  }
```

6.6.1 Line-by-Line Explanation

6.6.1.1 Function Definition

```
void StartACC_task(void *argument)
```

- This line defines the function **`StartACC_task`** that takes a single **`void*`** argument. It's intended to be used as a task in an RTOS environment.

6.6.1.2 Variable Declaration

```
float distance = 0;
```

- Declares a floating-point variable **`distance`** and initializes it to 0. This variable will store the distance measured by the ultrasonic sensor.

6.6.1.3 Infinite Loop

```
for(;;)
```

- Starts an infinite loop, which is typical in RTOS tasks to keep the task running continuously.

6.6.1.4 Read Distance

```
distance = readDistance();
```

- Calls the **`readDistance`** function to get the current distance from the ultrasonic sensor and stores it in the `distance` variable.

6.6.1.5 Adjust Speed

```
adjustSpeed(distance);
```

- Calls the `adjustSpeed` function, passing the `distance` value. This function adjusts the car's speed based on the measured distance, likely to maintain a safe following distance.

6.6.1.6 Delay

```
HAL_Delay(100);
```

- Delays execution for 100 milliseconds, allowing the car's speed adjustment to take effect and providing time for the sensor to take another reading.

6.6.1.7 Yield Control

```
osDelay(1);
```

- This RTOS-specific function yields control to other tasks, allowing them to run for 1 time unit. It helps in managing multitasking effectively.

6.7 Why RTOS Here?

1. **Task Management** RTOS efficiently manages multiple tasks, ensuring that critical operations like reading sensor data and adjusting speed are performed timely.

2. **Real-Time Performance** The RTOS provides real-time performance guarantees, crucial for automotive applications where delays could lead to safety risks.

3. **Multitasking** With `osDelay`, the function yields control to other tasks, allowing the system to perform multiple operations simultaneously without conflict or delay.

4. **Deterministic Behavior** RTOS ensures predictable execution times for tasks, which is essential for maintaining consistent and safe vehicle behavior.

6.8 Summary

The **`StartACC_task`** function continuously reads the distance from an ultrasonic sensor and adjusts the car's speed accordingly. The use of RTOS allows this task to run reliably and efficiently, providing real-time performance and effective multitasking capabilities, which are crucial for automotive safety and performance.

CHAPTER 7

TELEMATICS CONTROL UNIT

GRADUATION PROJECT 2024

7 Chapter 7: Telematics Control Unit

7.1 ESP File System

7.1.1 What is a file system and what are its uses?

The ESP allows us to partition the system flash in a way so we can use it to store both code and support a file system. This filing system can be used to store infrequently changing data such as web pages, configurations, sensor calibration data, etc. We can then read the files from the ESP file system in our program.

The file system is used to:

- Store Configuration Files: It stores files which usually include Wi-Fi credentials, server addresses, sensor thresholds or user preferences that we need to retain across the firmware updates.

- Store Firmware Updates: The file system is used to store the new firmware binary files that are downloaded over the air because usually updating involves downloading the new firmware to the file system and then performing a safe switch-over to the new firmware.

- **Web Server Data :** If your ESP device acts as a web server, the file system stores the HTML, CSS, and JavaScript files that make up the web pages users see when they connect. This allows for dynamic content or customization without needing the entire firmware to be updated.

7.1.2 File systems used in ESP based projects

7.1.2.1 SPIFFS (SPI Flash File System):

- SPIFFS is a lightweight file system crafted for microcontrollers with constrained resources.

- SPIFFS has some shortcomings when it comes to maximum file size and efficiency, however it is simple to use and integrate.

7.1.2.2 LittleFS:

- LittleFS which we have used in our project is an improved file system designed in place of SPIFFS, it is of better performance and solidity.
- It provides wear leveling and power-loss resilience, making it more robust for applications where the flash memory is frequently written.
- LittleFS is advocated for new projects due to its superiority over SPIFFS.

7.1.2.3 FatFS:

- FatFS is a more general-purpose file system that can be used with various storage media.
- It is a full-featured file system supporting larger files and directories.
- FatFS can be useful when working with more considerable storage needs that exceed the internal flash memory.

7.2 HEX Parser

A HEX parser is a program or function used when dealing with firmware updates distributed in HEX format. The HEX format is a text-based configuration used to depict binary data in hexadecimal form and we usually do this when programming microcontrollers' flash memory. It is vital to apply HEX parser properly as it maintains and ensures the reliability and efficiency of our project.

7.2.1 Uses of HEX parsers

- Firmware development: where HEX parsers are used to analyze firmware, images stored in hex format.

- Data analysis: HEX parsers are often used in debugging where it they analyze and identify possible issues with data stored in log files or communication protocols that use hex encoding
- File conversion: Some hex parsers can convert hex data into other formats like binary files or assembly code, useful for programmers working with low-level code.

7.2.1.1 How HEX parser is executed in FOTA

- Firstly, we receive the HEX file over the air typically through Bluetooth, WIFI or cellular
- The HEX parser will then read the file and **extract and validate** the data and will administer the addresses
- Validation of the data is done through the checksum of each record
- Next, the binary data we extricated is written in the appointed memory addresses in the flash memory of the microcontroller

7.2.1.2 Examples of HEX parsers:

- Embedded systems development tools: built-in hex parsers to work with firmware images
- Online tools: some websites offer the ability to convert a hex string to decimal or binary using a free online hex parser.
- Programming libraries: Programming languages have libraries that provide hex parsing service.

7.3 Linking ESP with Firebase

The idea of linking our ESP with firebase is so that we can store firmware versions, applications and FOTA update URLs. To link them we need a Firebase account and Firebase Realtime Database. So how did we go about linking them?

7.3.1 Setting up Firebase Project:

- Open the Firebase console
- Create and authenticate a project by adding a public-facing name and support email, this helps check access.
- Create a Realtime Database which is used to store and synchronize data Instantaneously.
- Set up your Cloud Storage and upload firmware “Application.bin” binary file to your storage.

7.3.2 Install Firebase Libraries:

- Install the Firebase ESP8266 client library. It is available in the Arduino IDE Library Manager as "Firebase Arduino Client Library for ESP8266 and ESP32 ".
- Define your API key, email, password and storage bucket ID in your Arduino IDE code.

7.3.3 Benefits of linking Firebase with ESP in FOTA:

- **Centralized Storage:** Oversee firmware versions from Firebase Storage.
- **Easy Updates:** You can automatically check and update to the latest version.
- **Revert from latest update:** You can go back to a previous firmware version stored in Firebase.
- **Scalability:** Firebase can oversee updates for a large number of ESP devices.

7.4 TCU Code Breakdown

```
6
7   /*****************************************   Included Libraries   ***********************************************************/
8   #include <LittleFS.h>
9   #include <Arduino.h>
10  #include <Firebase_ESP_Client.h>
11  #include <PubSubClient.h>
12  #include <addons/TokenHelper.h>
13
14  /*************************************************  MACROS - Variables  ***************************************************/
15  #if defined(ESP32) || defined(ARDUINO_RASPBERRY_PI_PICO_W)
16  #include <WiFi.h>
17  #elif defined(ESP8266)
18  #include <ESP8266WiFi.h>
19  #elif __has_include(<WiFiNINA.h>)
20  #include <WiFiNINA.h>
21  #elif __has_include(<WiFi101.h>)
22  #include <WiFi101.h>
23  #elif __has_include(<WiFiS3.h>)
24  #include <WiFiS3.h>
25  #endif
```

Figure 7-1: TCU Code Breakdown

7.4.1 Included Libraries

- **LittleFS.h**: Filesystem library for ESP8266 and ESP32.
- **Arduino.h**: Standard Arduino library.
- **Firebase_ESP_Client.h**: Library for Firebase interactions.
- **PubSubClient.h**: MQTT client library.
- **TokenHelper.h**: Helper library for Firebase token management.
- **WiFi.h**, **ESP8266WiFi.h**, **WiFiNINA.h**, **WiFi101.h**, **WiFiS3.h**: Libraries for Wi-Fi connectivity depending on the board used.

```
/* 1. Define the WiFi credentials */
#define WIFI_SSID             "Fadel"
#define WIFI_PASSWORD         "012345678"

/* 2. Define the API Key */
#define API_KEY               "AIzaSyAwL-MVSb_uf4fZ4gjM3PBiYBYanKi2Q8c"

/* 3. Define the user Email and password that alreadey registerd or added in your project */
#define USER_EMAIL            "fadelmohamed100s@gmail.com"
#define USER_PASSWORD         "123456"

/* 4. Define the Firebase storage bucket ID e.g bucket-name.appspot.com */
#define STORAGE_BUCKET_ID     "fota-test-ee1af.appspot.com"

#define NOT_ACKNOWLEDGE       0xAB
#define JUMP_SECCEDDED        0x01
#define ERASE_SUCCEDDED       0x03
#define WRITE_SUCCEDDED       0x01

const char* mqtt_server = "test.mosquitto.org";

/* Command packet in communication with STM32 */
uint8_t packet[100];

// Define Firebase Data object
FirebaseData fbdo;

FirebaseAuth auth;
FirebaseConfig config;

#if defined(ARDUINO_RASPBERRY_PI_PICO_W)
WiFiMulti multi;
#endif

WiFiClient espClient;
PubSubClient client(espClient);
```

Figure 7-2: Libraries

7.4.2 Macros – Global Variables

- **Wi-Fi credentials**: WIFI_SSID, WIFI_PASSWORD.
- **Firebase credentials**: API_KEY, USER_EMAIL, USER_PASSWORD, STORAGE_BUCKET_ID (which have been defined in the firebase section)
- **MQTT server**: mqtt_server (to link ESP with our mqtt broker which is mosquito)
- Command packet: packet.
- NOT_ACKNOWLEDGE 0xAB
- JUMP_SECCEDDED 0x01
- ERASE_SUCCEDDED 0x03

  } Constants used when communicating with STM32
- WRITE_SUCCEDDED 0x01
- **FirebaseData fbdo**: Object for Firebase operations.
- **FirebaseAuth auth**: Authentication object for Firebase.
- **FirebaseConfig config**: Configuration object for Firebase.

- WiFiClient espClient: Client for Wi-Fi.
- PubSubClient client: Client for MQTT.

```
/*************************************************    Setup    ************************************************/
void setup()
{
    Serial.begin(115200);
    Serial1.begin(115200, SERIAL_8N1);

#if defined(ARDUINO_RASPBERRY_PI_PICO_W)
    multi.addAP(WIFI_SSID, WIFI_PASSWORD);
    multi.run();
#else
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
#endif

    Serial.print("Connecting to Wi-Fi");
    unsigned long ms = millis();
    while (WiFi.status() != WL_CONNECTED)
    {
        Serial.print(".");
        delay(300);
#if defined(ARDUINO_RASPBERRY_PI_PICO_W)
        if (millis() - ms > 10000)
            break;
#endif
    }
    Serial.println();
    Serial.println("Connected");

    Serial.printf("Firebase Client v%s\n", FIREBASE_CLIENT_VERSION);

    /* Assign the api key (required) */
    config.api_key = API_KEY;

    /* Assign the user sign in credentials */
    auth.user.email = USER_EMAIL;
    auth.user.password = USER_PASSWORD;

#if defined(ARDUINO_RASPBERRY_PI_PICO_W)
    config.wifi.clearAP();
    config.wifi.addAP(WIFI_SSID, WIFI_PASSWORD);
#endif

    /* Assign the callback function for the long running token generation task */
    config.token_status_callback = tokenStatusCallback;

    // Comment or pass false value when WiFi reconnection will control by your code or third party library e.g. WiFiManager
    Firebase.reconnectNetwork(true);
    fbdo.setBSSLBufferSize(4096 /* Rx buffer size in bytes from 512 - 16384 */, 1024 /* Tx buffer size in bytes from 512 - 16384 */);
    /* Assign download buffer size in byte */
    config.fcs.download_buffer_size = 2048;
    Firebase.begin(&config, &auth);

    client.setServer(mqtt_server, 1883);
    client.setCallback(callback);

    pinMode(2, OUTPUT);
}
```

Figure 7-3: Macros – Global Variables

7.4.3 Setup

- **Serial.begin:** Makes for serial initialization where it sets up communication with STM32 and for debugging.
- **WiFi.begin:** connects to a certain Wi-Fi network and waits a certain amount of second to ensure that a network connection is established
- **Firebase configuration:** Configures Firebase client with API key and user credentials. Sets up token status callback and buffer sizes.
- **MQTT Setup**: Configures MQTT client with the server and sets the callback function.
- **Pin Setup**: Configures GPIO pin 2 as an output for LED or another indicator.

```
121 /*************************************************     Loop     ************************************************************/
122 void loop()
123 {
124     if(!client.connected()) { reconnect(); }
125     client.loop();
126 }
127
128
129 /********************************** The Firebase Storage download callback function *************************************/
130 void fcsDownloadCallback(FCS_DownloadStatusInfo info)
131 {
132     if (info.status == firebase_fcs_download_status_init)
133     {
134         Serial.printf("Downloading file %s (%d) to %s\n", info.remoteFileName.c_str(), info.fileSize, info.localFileName.c_str());
135     }
136     else if (info.status == firebase_fcs_download_status_download)
137     {
138         Serial.printf("Downloaded %d%s, Elapsed time %d ms\n", (int)info.progress, "%", info.elapsedTime);
139     }
140     else if (info.status == firebase_fcs_download_status_complete)
141     {
142         Serial.println("Download completed\n");
143     }
144     else if (info.status == firebase_fcs_download_status_error)
145     {
146         Serial.printf("Download failed, %s\n", info.errorMsg.c_str());
147     }
148 }
```

Figure 7-4: Setup

7.4.4 Loop

- **reconnect( ) :** confirms a connection with MQTT client.
- **client.loop( ) :** manages and processes incoming MQTT messages and maintains the connection. In the case of a disconnection, it will try to reconnect.

7.4.5 Firebase Callback

- **fcsDownloadCallback(FCS_DownloadStatusInfo info):** When downloading a file from Firebase Storage, this function deals with status updates that involve initialization, progress, completion, and errors.

```
151  /****************************************************** Reconnect to MQTT ******************************************************/
152  void reconnect()
153  {
154    while(!client.connected())
155    {
156      Serial.println("Attempting MQTT connection...");
157
158      if(client.connect("ESPClient"))
159      {
160        Serial.println("Connected");
161        client.subscribe("/FOTA/Boot");
162      }
163      else
164      {
165        Serial.print("Failed, rc=");
166        Serial.print(client.state());
167        Serial.println(" try again in 5 seconds");
168        delay(5000);
169      }
170    }
171  }
```

Figure 7-5: Firebase Callback

7.4.6 MQTT Reconnection

- **reconnect( ):** If the connection is lost with our MQTT broker, it will attempt reconnection.
- Subscribes to the topic "/FOTA/Boot" upon reconnection

```
174 /************************************************** Node-Red Callback **************************************************/
175 void callback(char* topic, byte* payload, unsigned int length)
176 {
177     byte recCommand = 0;
178     Serial.print("Message arrived [");
179     Serial.print(topic);
180     Serial.print("]: ");
181     recCommand = payload[0];
182     Serial.println(recCommand);
183 
184     switch(recCommand)
185     {
186       case 'G':  Send_CMD_Read_CID();               break;
187       case 'R':  Send_CMD_Read_Protection_Level();  break;
188       case 'J':  Send_CMD_Jump_To_Application();    break;
189       case 'E':  Send_CMD_Erase_Flash();            break;
190       case 'U':  Send_CMD_Upload_Application();     break;
191     }
192 }
193 
```

Figure 7-6: MQTT Reconnection

7.4.7 Node Red Callback

i. *callback (char* topic, byte* ***payload, unsigned int length):** Processes incoming MQTT messages. It has 5 different cases, and each received command will call upon a specific function to send to our STM32.

i. case 'G': Send_CMD_Read_CID( ) ⟶ Read chip ID

ii. case 'R': Send_CMD_Read_Protection_Level( ) ⟶ Read protection Level

iii. case 'J': Send_CMD_Jump_To_Application( ) ⟶ Jump to Application

iv. case 'E': Send_CMD_Erase_Flash( ) ⟶ Erase Flash Memory

v. case 'U': Send_CMD_Upload_Application( ) ⟶ Upload New Application

```
195 /************************************** Communication with bootloader **************************************************/
196 void SendCMDPacketToBootloader(uint8_t packetLength)
197 {
198     for(uint8_t dataIndex = 0 ; dataIndex < packetLength ; dataIndex++){
199         Serial1.write(packet[dataIndex]);
200         Serial.print(packet[dataIndex], HEX);
201         Serial.print(" ");
202     }
203     Serial.println("\n");
204 }
205 
206 void ReceiveReplayFromBootloader(uint8_t packetLength)
207 {
208     for(uint8_t dataIndex = 0 ; dataIndex < packetLength ; dataIndex++)
209     {
210         while(!Serial1.available());
211         packet[dataIndex] = Serial1.read();
212         if(NOT_ACKNOWLEDGE == packet[0]) break;
213     }
```

Figure 7-7: Node Red Callback

7.4.8 Communication with Bootloader

i. Send command packet

- **SendCMDPacketToBootloader(uint8_t packetLength):** Sends a command packet via Serail1 to the STM32 bootloader.
- Prints the packet data for debugging.

ii. Receive reply

- ReceiveReplayFromBootloader(uint8_t packetLength): Receives a reply packet from the STM32 bootloader.
- Waits until data is available on Serial1 before reading

7.4.9 Bootloader Commands

Each of these functions prepares a command packet, sends it to the STM32 bootloader, receives the response, and publishes the response to the MQTT topic "/FOTA/BootReply"

```
/************************************** Bootloader Supported Commands' Functions ***************************************/
void Send_CMD_Read_CID()
{
    char Replay[16] = {0};
    String sReplay = "Chip ID: 0x";

    packet[0] = 5;
    packet[1] = 0x10;
    *((uint32_t*)((uint8_t*)packet + 2)) = calculateCRC32((uint8_t*)packet, 2);
    SendCMDPacketToBootloader(6);
    ReceiveReplayFromBootloader(2);

    if(NOT_ACKNOWLEDGE == packet[0]){ sReplay = "NACK"; }
    else{ sReplay += String(packet[1], HEX);   sReplay += String(packet[0], HEX); }

    sReplay.toCharArray(Replay, sReplay.length()+1);
    client.publish("/FOTA/BootReply", Replay, false);
}
```

Figure 7-8: Bootloader Commands

i. Read Chip ID

- **Send_CMD_Read_CID( ):** Sends a command to read the chip ID from the STM32 bootloader.
- Then it receives and processes the response.

- If the chip ID is received, then it will be published to the MQTT topic "/FOTA/BootReply" but if not then NACK (not acknowledge) is published to the MQTT topic.

```
236  void Send_CMD_Read_Protection_Level()
237  {
238      char Replay[16] = {0};
239      String sReplay = "ProtecLvl: ";
240
241      packet[0] = 5;
242      packet[1] = 0x11;
243      *((uint32_t*)((uint8_t*)packet + 2)) = calculateCRC32((uint8_t*)packet, 2);
244      SendCMDPacketToBootloader(6);
245      ReceiveReplayFromBootloader(1);
246
247      if(NOT_ACKNOWLEDGE == packet[0]){ sReplay = "NACK"; }
248      else{ sReplay += String(packet[0], HEX); }
249
250      sReplay.toCharArray(Replay, sReplay.length()+1);
251      client.publish("/FOTA/BootReply", Replay, false);
252  }
253
```

Figure 7-9: Read Chip ID

ii. Read Protection Level

- **Send_CMD_Read_Protection_Level( ):** Sends a command to read the protection level from the STM32 bootloader.
- Then it receives and processes the response.
- If the protection level is received, then it will be published to the MQTT topic "/FOTA/BootReply" but if not then NACK (not acknowledge) is published to the MQTT topic.

```
254  void Send_CMD_Jump_To_Application()
255  {
256      char Replay[16] = {0};
257      String sReplay = "";
258      packet[0] = 5;
259      packet[1] = 0x12;
260      *((uint32_t*)((uint8_t*)packet + 2)) = calculateCRC32((uint8_t*)packet, 2);
261      SendCMDPacketToBootloader(6);
262      ReceiveReplayFromBootloader(1);
263
264      if(JUMP_SECCEDDED == packet[0]){ sReplay += "Jump Success"; }
265      else{ sReplay += "Jump Fail"; }
266
267      sReplay.toCharArray(Replay, sReplay.length()+1);
268      client.publish("/FOTA/BootReply", Replay, false);
269  }
```

Figure 7-10: Read Protection Level

iii. Jump to Application

- **Send_CMD_Erase_Flash( ):**Prepares and sends a command to jump to the application code on the STM32.
- Then it receives and processes the response.
- Next the jump status is published to the MQTT topic.

```
270
271  void Send_CMD_Erase_Flash()
272  {
273      char Replay[16] = {0};
274      String sReplay = "";
275      packet[0] = 5;
276      packet[1] = 0x13;
277      *((uint32_t*)((uint8_t*)packet + 2)) = calculateCRC32((uint8_t*)packet, 2);
278      SendCMDPacketToBootloader(6);
279      ReceiveReplayFromBootloader(1);
280
281      if(ERASE_SUCCEDDED == packet[0]){ sReplay += "Erase Success";  }
282      else{ sReplay += "Erase Unsuccess"; }
283
284      sReplay.toCharArray(Replay, sReplay.length()+1);
285      client.publish("/FOTA/BootReply", Replay, false);
286  }
```

Figure 7-11: Jump to Application

iv. Erase Flash Memory

- **Send_CMD_Erase_Flash( ):** Prepares and sends a command to erase the flash memory on the STM32.
- Then it receives and processes the response.
- Next the erase status is published to the MQTT topic.

```
void Send_CMD_Upload_Application()
{
    uint8_t isFailed = 0;
    if (Firebase.ready())
    {
        Serial.println("\nDownload file...\n");

        // The file systems for flash and SD/SDMMC can be changed in FirebaseFS.h.
        if (!Firebase.Storage.download(&fbdo, STORAGE_BUCKET_ID /* Firebase Storage bucket id */, "Application.bin" /* path of remote file stored in the bucket */
            Serial.println(fbdo.errorReason());

        File file = LittleFS.open("/update.bin", "r");
        short addressCounter = 0;

        while(file.available())
        {
            char Replay[16] = {0};
            String sReplay = "";
            digitalWrite(2, !digitalRead(2));
            uint8_t ByteCounter = 0;

            packet[0] = 74;
            packet[1] = 0x14;
            *(int*)(&(packet[2])) = 0x8008000 + (0x40 * addressCounter);
            packet[6] = 64;

            Serial.printf("Start Address: 0x%X\n", *(int*)(&(packet[2])));

            while(ByteCounter < 64)
            {
                packet[7 + ByteCounter] = file.read();
                ByteCounter++;
            }
            *((uint32_t*)((uint8_t*)packet + 71)) = calculateCRC32((uint8_t*)packet, 71);

            SendCMDPacketToBootloader(75);

            ReceiveReplayFromBootloader(1);

            if(WRITE_SUCCEDDED == packet[0]){ sReplay += "Write Success"; }
            else{ isFailed = 1; break; }

            sReplay.toCharArray(Replay, sReplay.length()+1);
            client.publish("/FOTA/BootReply", Replay, false);

            addressCounter++;
        }

        if(isFailed){ client.publish("/FOTA/BootReply", "Upload Fail", false); }
        else{ client.publish("/FOTA/BootReply", "Upload Success", false);  }
```

Figure 7-12: Erase Flash Memory

Upload Application

- **Send_CMD_Upload_Application( ):** Prepares and sends a command to start the application upload process to the STM32.
- Downloads the firmware binary file from Firebase Storage.
- The CRC32 checksum is calculated to ensure integrity of data.

- Reads the firmware binary file and sends it to the STM32 in parts and response is received.
- If the response indicates a successful write (WRITE_SUCCEDDED), the current counter value is published to the MQTT topic.

v. CRC Calculation

- **CalculateCRC32(const uint8_t *pData, uint8_t pLength)**: Calculates the CRC32 checksum for data integrity verification.
- **CRC32_MPEG2(uint8_t *p_data, uint32_t data_length)**: Helper function for CRC32 calculation using an MPEG-2 lookup table.
- Uses a bitwise method with a specified polynomial
- The function processes each byte and bit (starting from the most significant bit) of data.
- The function returns the final calculated CRC value.

```
/********************************************** CRC Caculation ****************************************************/
uint32_t calculateCRC32(const uint8_t *pData, uint8_t pLength)
{
    uint8_t crcPacket[290];
    uint16_t crcPacketCounter = 0;
    for(uint8_t i = 0 ; i < pLength ; i++)
    {
        crcPacket[crcPacketCounter++] = 0x00;
        crcPacket[crcPacketCounter++] = 0x00;
        crcPacket[crcPacketCounter++] = 0x00;
        crcPacket[crcPacketCounter++] = pData[i];
    }
    return CRC32_MPEG2(crcPacket, crcPacketCounter);
}

static const uint32_t crc_table[0x100] = {
        0x00000000, 0x04C11DB7, 0x09823B6E, 0x0D4326D9, 0x130476DC, 0x17C56B6B, 0x1A864DB2, 0x1E475005, 0x2608EDB8, 0x22C9F00F, 0x2F8AD6D6, 0x2B4BCB61, 0x350C9B
        0x4C11DB70, 0x48D0C6C7, 0x4593E01E, 0x4152FDA9, 0x5F15ADAC, 0x5BD4B01B, 0x569796C2, 0x52568B75, 0x6A1936C8, 0x6ED82B7F, 0x639B0DA6, 0x675A1011, 0x791D40
        0x9823B6E0, 0x9CE2AB57, 0x91A18D8E, 0x95609039, 0x8B27C03C, 0x8FE6DD8B, 0x82A5FB52, 0x8664E6E5, 0xBE2B5B58, 0xBAEA46EF, 0xB7A96036, 0xB3687D81, 0xAD2F2D
        0xD4326D90, 0xD0F37027, 0xDDB056FE, 0xD9714B49, 0xC7361B4C, 0xC3F706FB, 0xCEB42022, 0xCA753D95, 0xF23A8028, 0xF6FB9D9F, 0xFBB8BB46, 0xFF79A6F1, 0xE13EF6
        0x34867077, 0x30476DC0, 0x3D044B19, 0x39C556AE, 0x278206AB, 0x23431B1C, 0x2E003DC5, 0x2AC12072, 0x128E9DCF, 0x164F8078, 0x1B0CA6A1, 0x1FCDBB16, 0x018AEB
        0x7897AB07, 0x7C56B6B0, 0x71159069, 0x75D48DDE, 0x6B93DDDB, 0x6F52C06C, 0x6211E6B5, 0x66D0FB02, 0x5E9F46BF, 0x5A5E5B08, 0x571D7DD1, 0x53DC6066, 0x4D9B30
        0xACA5C697, 0xA864DB20, 0xA527FDF9, 0xA1E6E04E, 0xBFA1B04B, 0xBB60ADFC, 0xB6238B25, 0xB2E29692, 0x8AAD2B2F, 0x8E6C3698, 0x832F1041, 0x87EE0DF6, 0x99A95D
        0xE0B41DE7, 0xE4750050, 0xE9362689, 0xEDF73B3E, 0xF3B06B3B, 0xF771768C, 0xFA325055, 0xFEF34DE2, 0xC6BCF05F, 0xC27DEDE8, 0xCF3ECB31, 0xCBFFD686, 0xD5B886
        0x690CE0EE, 0x6DCDFD59, 0x608EDB80, 0x644FC637, 0x7A089632, 0x7EC98B85, 0x738AAD5C, 0x774BB0EB, 0x4F040D56, 0x4BC510E1, 0x46863638, 0x42472B8F, 0x5C007B
        0x251D3B9E, 0x21DC2629, 0x2C9F00F0, 0x285E1D47, 0x36194D42, 0x32D850F5, 0x3F9B762C, 0x3B5A6B9B, 0x0315D626, 0x07D4CB91, 0x0A97ED48, 0x0E56F0FF, 0x1011A0
        0xF12F560E, 0xF5EE4BB9, 0xF8AD6D60, 0xFC6C70D7, 0xE22B20D2, 0xE6EA3D65, 0xEBA91BBC, 0xEF68060B, 0xD727BBB6, 0xD3E6A601, 0xDEA580D8, 0xDA649D6F, 0xC423CD
        0xBD3E8D7E, 0xB9FF90C9, 0xB4BCB610, 0xB07DABA7, 0xAE3AFBA2, 0xAAFBE615, 0xA7B8C0CC, 0xA379DD7B, 0x9B3660C6, 0x9FF77D71, 0x92B45BA8, 0x9675461F, 0x883216
        0x5D8A9099, 0x594B8D2E, 0x5408ABF7, 0x50C9B640, 0x4E8EE645, 0x4A4FFBF2, 0x470CDD2B, 0x43CDC09C, 0x7B827D21, 0x7F436096, 0x7200464F, 0x76C15BF8, 0x68860B
        0x119B4BE9, 0x155A565E, 0x18197087, 0x1CD86D30, 0x029F3D35, 0x065E2082, 0x0B1D065B, 0x0FDC1BEC, 0x3793A651, 0x3352BBE6, 0x3E119D3F, 0x3AD08088, 0x2497D0
        0xC5A92679, 0xC1683BCE, 0xCC2B1D17, 0xC8EA00A0, 0xD6AD50A5, 0xD26C4D12, 0xDF2F6BCB, 0xDBEE767C, 0xE3A1CBC1, 0xE760D676, 0xEA23F0AF, 0xEEE2ED18, 0xF0A5BD
        0x89B8FD09, 0x8D79E0BE, 0x803AC667, 0x84FBDBD0, 0x9ABC8BD5, 0x9E7D9662, 0x933EB0BB, 0x97FFAD0C, 0xAFB010B1, 0xAB710D06, 0xA6322BDF, 0xA2F33668, 0xBCB466
};                      /* Global 32-bit CRC (MPEG-2) Lookup Table */

uint32_t CRC32_MPEG2(uint8_t *p_data, uint32_t data_length)
{
    uint32_t checksum = 0xFFFFFFFF;

    if (data_length == 0xFFFFFFFF)
        return checksum;

    for (unsigned int i = 0 ; i < data_length ; i++)
    {
        uint8_t top = (uint8_t) (checksum >> 24);
        top ^= p_data[i];
        checksum = (checksum << 8) ^ crc_table[top];
    }

    return checksum;
```

Figure 7-13: CRC Calculation

CHAPTER 8

GUI

GRADUATION PROJECT 2024

8 Chapter 8: GUI

The Role of GUI in FOTA Systems The Graphical User Interface (GUI) serves as the primary interface for users, typically automotive technicians or administrators, to interact with the FOTA system. It provides an intuitive platform for managing firmware updates, diagnosing issues, and monitoring system statuses across multiple ECUs within the vehicle.

GUI, or Graphical User Interface, is a visual way for users to interact with electronic devices, software, or systems through graphical icons and visual indicators rather than text-based interfaces or command-line interfaces (CLI). It enhances user experience by making complex operations more intuitive and accessible. Here are the key details about GUI:

8.1 Components of GUI

8.1.1 Visual Elements:

- **Icons and Images:** Represent functions, files, or applications visually.
- **Buttons:** Trigger actions when clicked or tapped.
- **Text Fields and Labels:** Display information and allow user input.
- **Menus and Toolbars:** Provide access to different functions and options.
- **Scrollbars and Sliders:** Navigate content or adjust settings.

8.1.2 Layout and Structure:

- **Windows and Frames:** Contain and organize GUI elements.
- **Panels and Tabs:** Segment and organize content within windows.
- **Grids and Alignment:** Arrange elements in a structured format.

8.1.3 Interactive Elements:

- **Forms:** Collect user input through fields and checkboxes.
- **Dialog Boxes:** Prompt for user decisions or provide notifications.
- **Drag-and-Drop:** Enable users to move files or data between elements.

8.2 Functions and Features

8.2.1 User Interaction:

- **Point-and-Click:** Navigate and interact with elements using a mouse or touch input.
- **Keyboard Input:** Enter data or navigate through keyboard shortcuts.
- **Gestures:** Perform actions on touch-enabled devices through gestures like tapping, swiping, or pinching.

8.2.2 Feedback and Display:

- **Status Indicators:** Show system status, progress, or alerts.
- **Feedback Messages:** Provide confirmation, error messages, or warnings.
- **Visual Styles:** Customize appearance for themes or accessibility needs.

8.2.3 Integration and Connectivity:

- **API Integration:** Connect with external services or data sources.
- **Network Communication:** Facilitate data exchange with remote systems or servers.
- **Device Compatibility:** Adapt to various screen sizes, resolutions, and input methods.

8.3 Design Principles

8.3.1 User-Centered Design:

- **Usability:** Ensure intuitive navigation and operation for users.
- **Accessibility:** Support diverse user needs, including visual or motor impairments.
- **Consistency:** Maintain uniformity in design elements and interactions.

8.3.2 Visual Design:

- **Layout Design:** Organize elements logically for efficient use.
- **Color and Contrast:** Use colors effectively for readability and emphasis.
- **Typography:** Choose fonts and text sizes that are clear and readable.

8.4 Performance and Responsiveness:

- **Speed:** Respond quickly to user input and system events.
- **Optimization:** Efficiently manage resources and minimize latency.
- **Adaptability:** Adjust to changing conditions or user preferences dynamically.

8.5 Advantages of GUI

- **Ease of Use:** Simplifies complex tasks by visualizing options and actions.
- **Productivity:** Speeds up workflow with intuitive navigation and direct interaction.
- **Learnability:** Reduces learning curve compared to text-based or command-line interfaces.
- **Engagement:** Enhances user satisfaction through appealing visual design and interactive features.

8.6 Applications of GUI

- **Software Applications:** Operating systems, productivity tools, and multimedia applications.
- **Embedded Systems:** Consumer electronics, industrial control panels, and automotive interfaces.
- **Web Interfaces:** Websites, web applications, and content management systems.

GUIs have revolutionized how users interact with technology, offering a versatile and user- friendly interface that continues to evolve with advancements in design, technology, and user expectations.

8.7 GUI Components and Functionalities

Main Interface The GUI is designed with a user-centric approach, featuring a clear layout that includes navigation bars, status indicators, and distinct functional areas. It ensures ease of navigation and efficient access to critical functionalities.

8.8 Buttons and Controls

- **First Page: User Login:** Enter username and password to login.

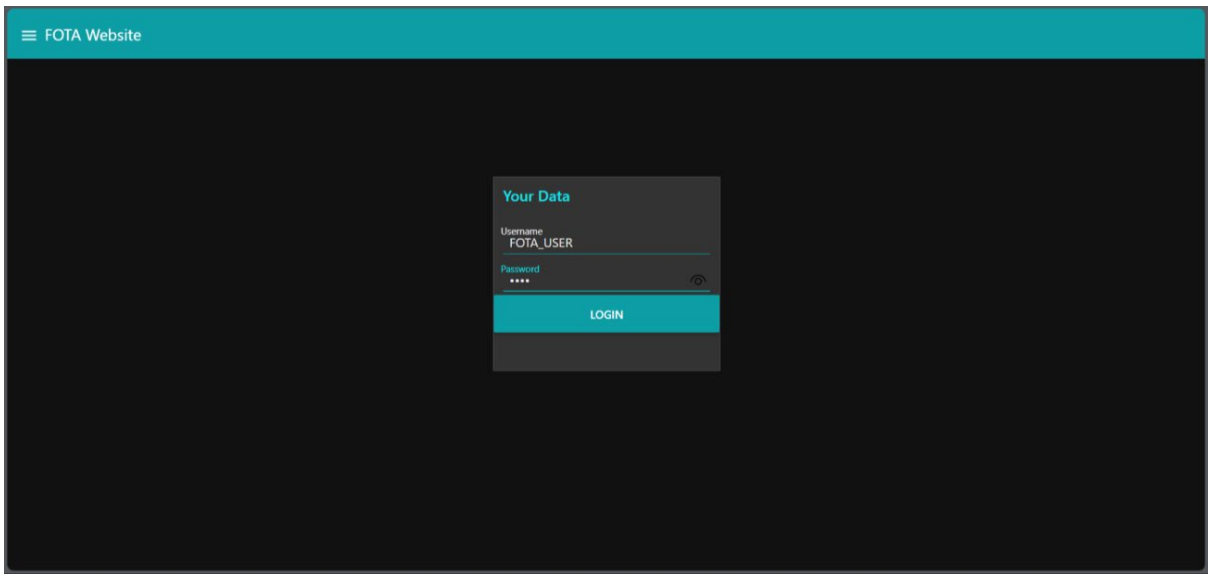


Figure 8-1: User Login

- **Upload Application**: Allows users to select and upload new firmware versions to ECUs via MQTT and a Mosquitto server, ensuring seamless integration and communication.

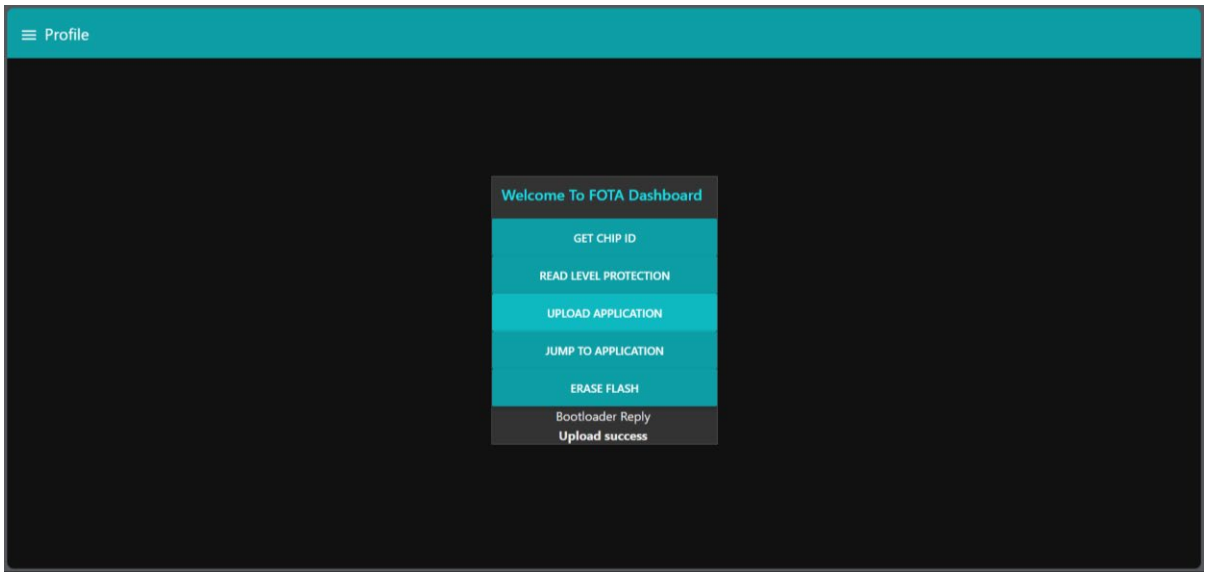


Figure 8-2: Upload Application

Erase Flash: Enables the erasure of flash memory on ECUs, preparing them for new firmware installations without the need for physical intervention.

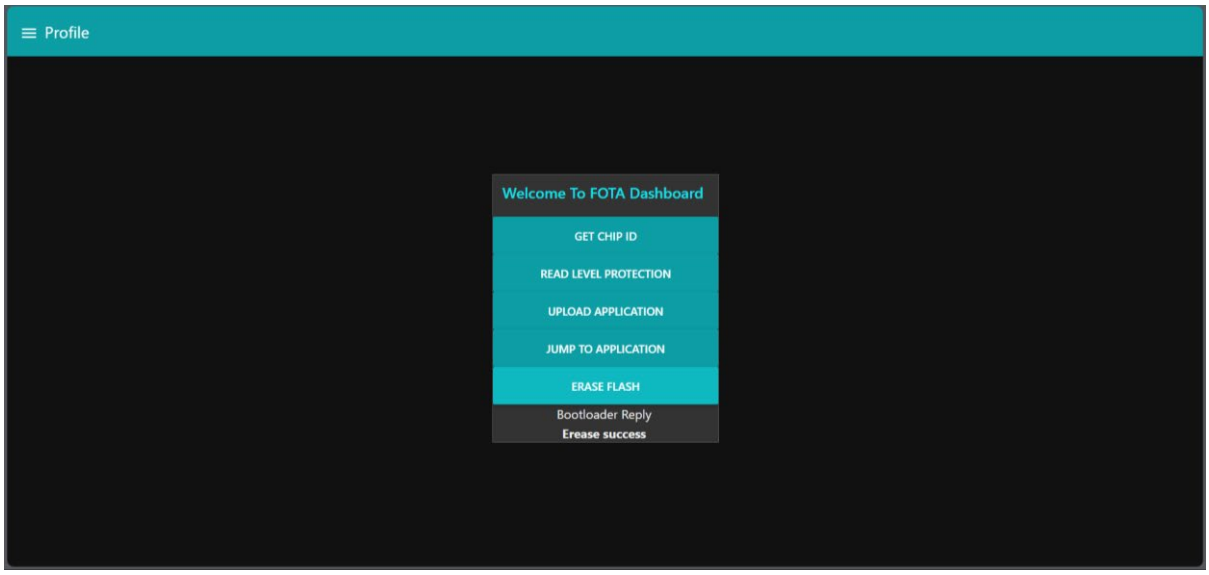


Figure 8-3: Erase Flash

- **Get Chip ID**: Provides a mechanism to retrieve unique chip identifiers from ECUs, facilitating identification and tracking.

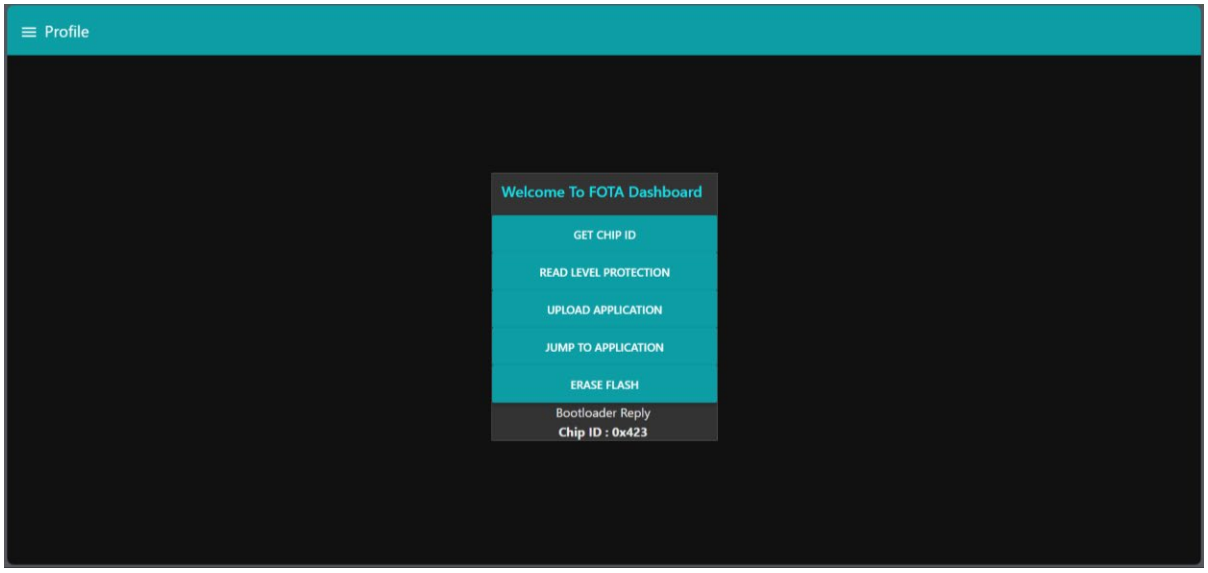


Figure 8-4: Get Chip ID

- **Read Protection Level**: Displays current security levels and access restrictions applied to ECUs, ensuring compliance with system integrity and security protocols

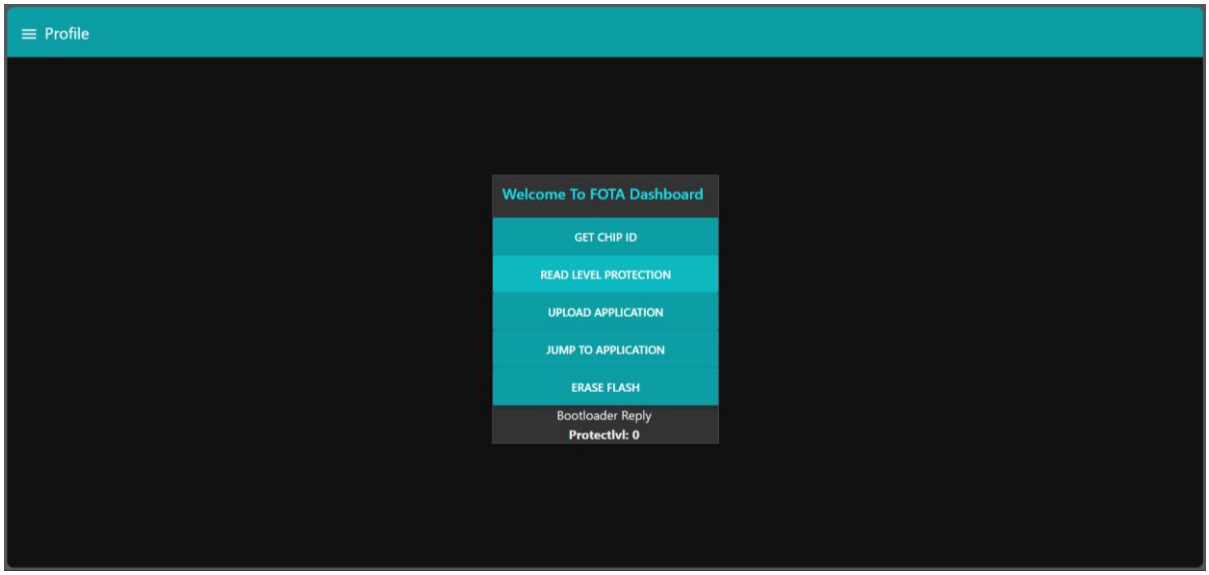


Figure 8-5: Read Protection Level

8.9 GUI Functionalities

8.9.1 User Input Handling

- **Event Handling:** Processes user interactions such as clicks, keystrokes, and mouse movements. Event handlers trigger appropriate actions or responses based on user input.
- **Input Validation:** Ensures that user-entered data meets specified criteria or formats before proceeding with operations, preventing errors or inconsistencies.

8.9.2 Navigation and Interaction

- **Navigation Bars:** Provide navigation controls and shortcuts to different sections or features within an application. They enhance user efficiency by facilitating quick access to commonly used functionalities.
- **Contextual Menus:** Display relevant options or commands based on the current context or selected items, offering context-sensitive actions that streamline user workflows.

8.9.3 Data Presentation and Visualization

- **Tables and Lists:** Display tabular or list-based data in a structured format, allowing users to view, sort, and manipulate data entries efficiently.
- **Charts and Graphs:** Visualize numerical data trends, relationships, or comparisons using graphical representations such as bar charts, line graphs, or pie charts.

8.9.4 Accessibility and Usability

- **Accessibility Features:** Include functionalities like keyboard navigation, screen reader compatibility, and high-contrast modes to accommodate users with diverse needs and disabilities.
- **Localization:** Adapt GUI elements, text, and formats to suit different languages, regions, enhancing usability for international users.

8.10 Visual Design

- **Color Scheme and Icons**: Utilizes a cohesive color palette that enhances readability and distinguishes between different system states (e.g., idle, updating, error).
- **Iconography**: Standardized icons for common actions (e.g., upload, erase, read) improve usability and navigation efficiency across the interface.
- **Responsive Design** Ensures compatibility with diverse automotive display screens, optimizing the GUI's performance and usability under different operating conditions and environments.

8.11 Communication Protocols: MQTT and Mosquitto Server

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for efficient communication between devices, particularly in constrained environments such as IoT (Internet of Things) and embedded systems. In the context of a FOTA (Firmware Over-The-Air) system for automotive vehicles, MQTT serves as a reliable means to transmit data and commands between the GUI running on Raspberry Pi devices and the ECUs (Electronic Control Units) embedded within the vehicle.

- **Publish-Subscribe Architecture:** MQTT operates on a publish-subscribe model where clients (in this case, the GUI on Raspberry Pi and ECUs) connect to a broker (such as Mosquitto server) which acts as a message mediator. This architecture enables multiple clients to exchange messages without direct communication, enhancing scalability and efficiency.
- **Topics:** Messages in MQTT are organized into topics, which act as channels for communication. The GUI and ECUs subscribe to relevant topics to receive updates or commands related to FOTA operations, such as firmware updates, status notifications, or diagnostic data.
- **Quality of Service (QoS):** MQTT supports different levels of QoS to ensure message delivery reliability:
  - **QoS 0 (At most once):** Messages are delivered at most once, without acknowledgment or retries.
  - **QoS 1 (At least once):** Messages are delivered at least once, ensuring delivery but allowing duplicates.
  - **QoS 2 (Exactly once):** Messages are delivered exactly once by using a four- step handshake process, ensuring no duplicates or lost messages.
- **Persistence and Retained Messages:** MQTT brokers like Mosquitto support retained messages, which store the last message published on a topic. This feature ensures that newly subscribed clients receive the most recent state immediately upon connecting.

8.12 Error Handling

Effective error handling is critical in ensuring the reliability and stability of the FOTA system, especially during communication between the GUI and ECUs. The GUI incorporates robust mechanisms to detect, manage, and respond to various types of errors that may occur:

- **Connection Errors:** Monitoring and handling connection failures between the GUI and MQTT broker or between the broker and ECUs. This includes retry mechanisms to automatically reconnect and resume communication.

- **Message Delivery Failures:** Implementing strategies to handle scenarios where messages fail to be delivered or acknowledged. Depending on the QoS level used, the GUI may resend messages or take corrective actions based on feedback from the MQTT broker.

- **Timeouts and Response Handling:** Setting timeouts for communication requests to prevent prolonged waiting periods. If no response is received within the specified time, the GUI triggers appropriate error handling procedures and notifies users of the issue.

- **Diagnostic Alerts:** Providing real-time feedback to users through error messages, status indicators, or logs displayed on the GUI interface. These alerts inform users about communication disruptions, system errors, or potential issues with FOTA operations.

- **Logging and Monitoring:** Logging detailed information about communication events, errors encountered, and actions taken by the GUI. Monitoring these logs allows administrators or technicians to analyze patterns, troubleshoot issues, and improve system performance over time.

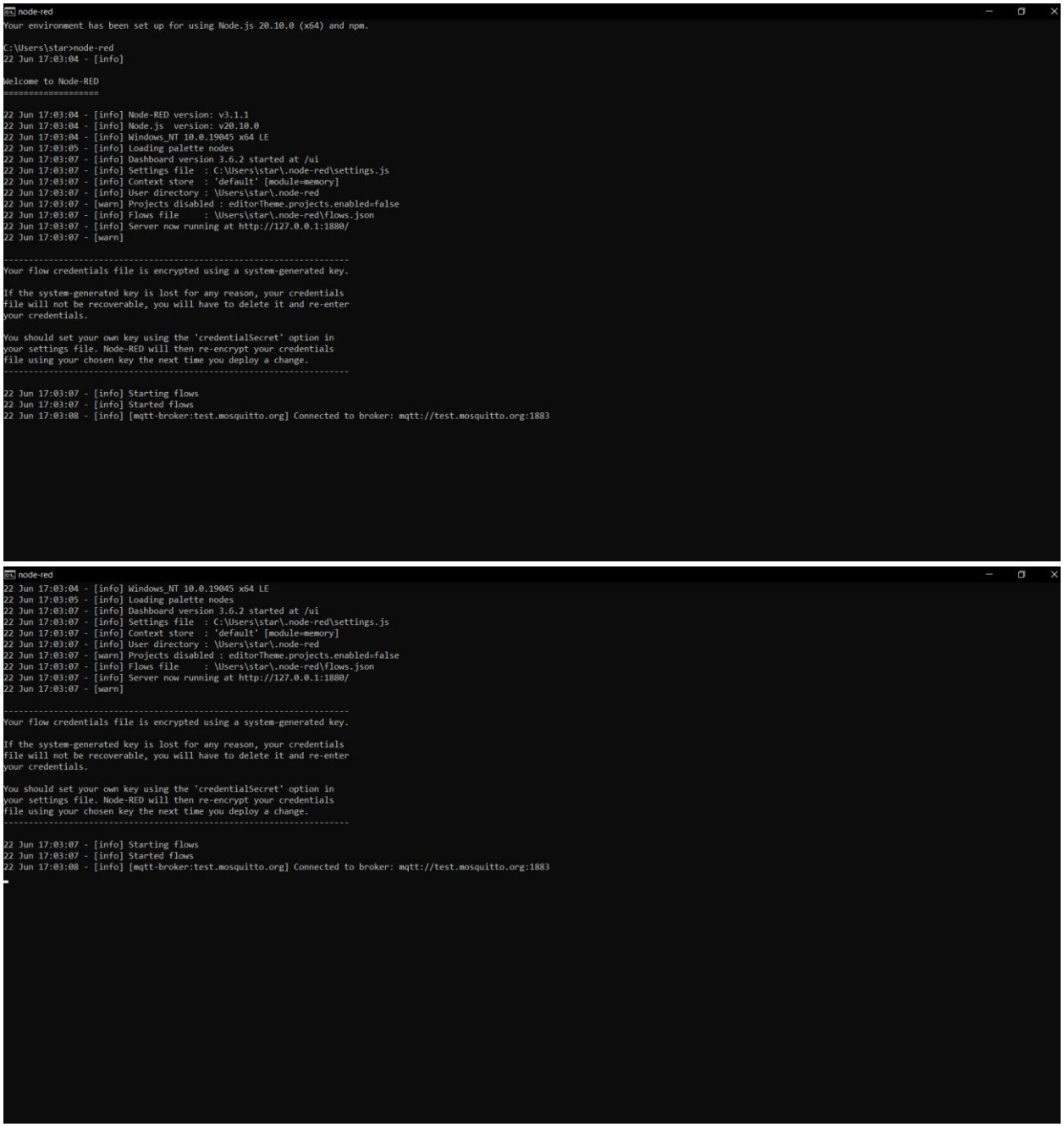


Figure 8-6: Error Handling

8.13 Summary

Integrating MQTT with a Mosquitto server in the FOTA system enables efficient and scalable communication between the GUI on Raspberry Pi devices and ECUs within automotive vehicles. Robust error handling mechanisms ensure the system's reliability by managing communication disruptions, handling errors promptly, and providing diagnostic alerts to users. This combination of reliable communication protocols and effective error

management enhances the overall performance and usability of the FOTA system, ensuring seamless firmware updates and operational efficiency in automotive environments.

8.14 Security Considerations

- **Authentication**: Implements secure login mechanisms to restrict access to authorized personnel, preventing unauthorized modifications to vehicle firmware.
- **Data Encryption**: Ensures end-to-end encryption of data exchanged between the GUI and ECUs, safeguarding sensitive information from potential cyber threats and unauthorized access.

CHAPTER 9

COMMUNICATION PROTOCOLS

GRADUATION PROJECT 2024

9 Chapter 9: Communication Protocols

9.1 UART:

UART stands for Universal Asynchronous Receiver Transmitter, it is a hardware communication interface associated with serial communication. UART is widely used in computer systems, microcontrollers, and embedded systems to transmit and receive data.

UART is very simple and only uses two wires between transmitter and receiver to transmit and receive in both directions. Both ends also have a ground connection. Communication in UART can be simplex (data is sent in one direction only), half-duplex (each side speaks but only one at a time), or full-duplex (both sides can transmit simultaneously). Data in UART is transmitted in the form of frames. The format and content of these frames will be briefly described and explained.

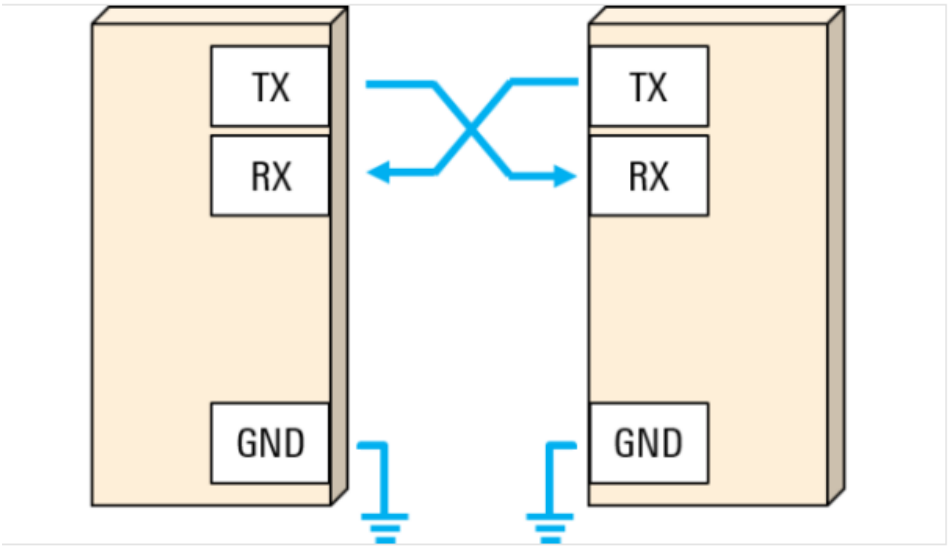


Figure 9-1: UART

9.2 Advantages of UART:

9.2.1 Serial protocol

Serial protocols are much better than parallel protocols as parallel protocols have a lot of problems such as high cost, high complexity, and interference between wires due to magnetic field.

9.2.2 Simplicity

This protocol requires only two wires for full duplex data transmission (apart from the power lines).

9.2.3 Asynchronous Communication

UART supports asynchronous communication, which means that the transmitter and receiver do not need to be synchronized by a common clock signal. This flexibility allows devices with different clock speeds or timing variations to communicate effectively.

9.2.4 Error Detection

The parity bit in UART is an optional bit used for error detection during data transmission. It is an additional bit transmitted after the data bits and before the stop bit(s). The parity bit allows the receiver to check for errors in the received data.

9.2.5 Flexibility in Baud Rates

UART supports a wide range of baud rates, allowing users to choose the appropriate data transfer speed according to the requirements of their application. This flexibility makes UART suitable for both low-speed and high-speed communication needs.

(Baud rate is used to determine the speed at which data is sent or received.)

9.2.6 UART Frame:

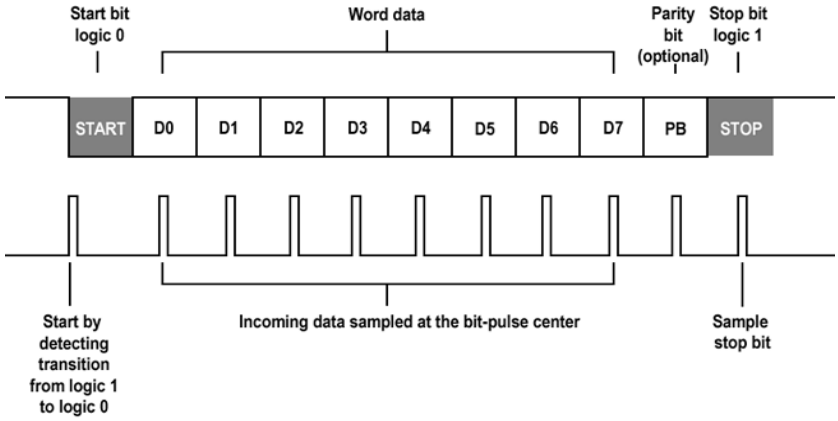


Figure 9-2: UART Frame

1) **start bit:** When UART is not transmitting data, the idle state of UART transmission line is high ‘’Logic 1’’. To start transmitting data, UART pulls the transmission line from high to low for only one clock cycle, When the receiver detects the high to low voltage transition, it begins to read the bits in the data frame at the frequency of the baud rate.

2) **Data bits:** The data bits contain the actual data that is being transferred. Its length can vary from 5 to 8 bits long if a parity bit is used. If there is no parity bit, the data length can be 9 bits long. The data is sent with the least significant bit first.

3) **Parity bit:** the simplest method for error detection

- A bit (0 or 1) is added to the data to help check if the data is received correctly. For even parity, all the '1' bits are counted, and if the count is odd, Then the parity bit is set to 1 to make an even number of ones.
- The same idea for the odd parity, the parity bit is set to make sure an odd number of ones in the data (including the parity bit).
- During reception, the receiver checks the received data and parity bit using the configured parity mode (even or odd). If the number of ones (including the parity bit) does not match the expected parity, an error is indicated. The receiver can then take appropriate action, such as discarding the data or requesting retransmission.
- parity checking can detect errors, it cannot correct them.

4) Stop bits:

- The stop bits in UART are used to indicate the end of a data frame during serial communication. After transmitting the data bits and the parity bit, one or more stop bits are sent to make the transmission line IDLE again.
- The stop bits are always logical high (1) and are used to signal the receiver that the transmitted data frame has ended. The receiver uses the stop bits to identify the boundary between consecutive data frames and prepare for the reception of the next frame.

9.2.7 UART in our project

- We used UART to connect between node MCU and main MCU
- Its function is to transfer the HEX files from node MCU to our STM.

9.3 MQTT:

- Message Queuing Telemetry Transport is a lightweight messaging protocol designed for efficient communication between devices in constrained networks, such as those with limited bandwidth, high latency, or low power availability. It follows a publish-subscribe messaging pattern, where publishers send messages to a broker, and subscribers receive those messages from the broker.
- The MQTT protocol is widely used in the Internet of Things (IoT) domain, where it provides a scalable and efficient solution for connecting and exchanging data between IoT devices, sensors, cloud platforms, and other components of IoT infrastructures.

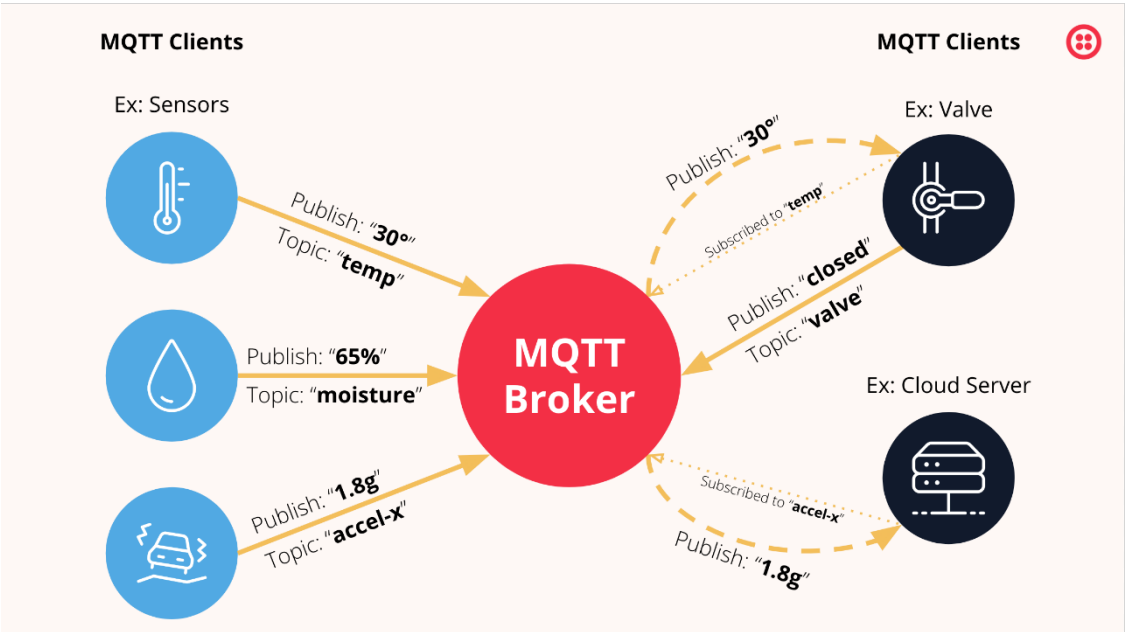


Figure 9-3: MQTT

9.4 Advantages of MQTT Protocol:

9.4.1 Publish-Subscribe Model:

- MQTT uses a publish-subscribe messaging model. Publishers (devices or applications) send messages, called "topics," to a broker. Subscribers can then subscribe to specific topics of interest and receive messages published on those topics. This decoupled approach allows for flexible and scalable communication between multiple publishers and subscribers.

9.4.2 Quality of Service Levels:

- MQTT provides different levels of Quality of Service to ensure reliable message delivery. The three QoS levels are:
- **QoS 0:** At most once delivery: Messages are sent without acknowledgment or guaranteed delivery. They may be lost or delivered multiple times.
- **QoS 1:** At least once delivery: Messages are guaranteed to be delivered at least once, but duplicates may occur.
- **QoS 2:** Exactly once delivery: Messages are guaranteed to be delivered exactly once, eliminating duplicates. This level involves a more complex handshake process.

9.4.3 Asynchronous Communication:

- MQTT supports asynchronous communication, allowing devices to send and receive messages without waiting for a response. This asynchronous nature enables efficient and non-blocking communication.

9.4.4 Wide Platform Support:

- MQTT is supported by a wide range of platforms, programming languages, and operating systems, making it easy to integrate into existing systems and frameworks.

9.4.5 MQTT in our Project

- We used secured mosquito broker to make asynchronous encryption to the messages
- Asynchronous encryption allows for end-to-end encryption of the firmware data, ensuring that the firmware is encrypted during the entire transport process.
- This protects the firmware data from potential eavesdropping or tampering during the transmission over the MQTT broker.
- The Function of MQTT is to send our commands from the website to the TCU (Our node MCU) to execute a certain function.

9.5 WI-FI

We also used the WI-FI module with the MQTT protocol and the TCU to serve these functions:

9.5.1 Connectivity and Data Transfer:

- The Wi-Fi module provides wireless connectivity between the TCU and the MQTT broker, enabling the transfer of FOTA-related data, such as firmware updates, configuration changes, and status updates.

9.5.2 Remote Access and Control:

- The Wi-Fi module and the MQTT protocol allow for remote access and control of the TCU from the FOTA management system.
- This enables the FOTA management system to initiate firmware updates, monitor the update status, and perform other FOTA-related operations on the telematics unit.

9.6 CAN Bus

9.6.1 What is CAN?

CAN stands for Controller Area Network

The CAN Bus is a serial two-wire half-duplex communication that allows multiple nodes in a system to communicate efficiently with each other. Each node is capable of sending and receiving messages, not all nodes can be communicating at once. All nodes receive all broadcast data from the bus then each node decide whether or not that data is relevant according to the filter masks of each node as shown in the following figure.

Figure 9-4: CAN Network

Any node wants to send a message it puts it on the bus with specific ID , then each node from the other nodes will check weather this ID is included in its filters or no , if yes it reads the message from the bus ,otherwise it ignores it.

CAN protocol is already widely used in vehicle and vessel internal components. In recent years, it has seen adoption in data communications and control in industry field.

9.6.2 Why did we choose CAN?

- Speed variety
  - Low Speed:

    CAN network offers low signal transfer rates that ranges between 40 kbps to 125 kbps.

  - High Speed:

    CAN offers high signal transfer rates between 40 kbps and 1 Mbps. High-speed CAN networks are terminated at both ends of the bus line with a 120-ohm resistor between CAN high and CAN low lines.

    The lower signaling rates allow communication to continue on the bus, even when a wiring failure takes place, while high-speed doesn't.

- Low cost

  When the CAN protocol was created, its aim and primary goal was to make faster communication between modules and

electronic devices in vehicles while decreasing the amount of wiring (and the amount of copper) necessary.

- Built-in Error Detection

  Error handling is built in the CAN protocol, each node can check for its errors during transmission and maintain its own error counter. When errors are detected nodes transmit a special Error Flag message and will destroy the offending bus traffic to prevent the error from spreading through the whole network. Even the node which is generating the fault will detect its own error in transmission, and raise its error counter, this eventually led the device to "bus off" and cease participating in the CAN network traffic. In this way, CAN nodes can both detect errors and prevent faulty devices from creating any useless bus traffic.

- Flexibility

  To understand the flexibility of the CAN bus protocol in communications, we need to know the difference between message-based protocols and address-based and. In address-based communication protocol, nodes communicate directly with each other by configuring themselves onto the same address.

  The CAN bus protocol is a message-based communication protocol. In this type of protocol, nodes on the bus have no address to identify them. As a result, nodes can easily be added or removed without performing any software or hardware updates on the system.

This feature makes it easy to integrate new electronic devices into the CAN bus network without significant programming overhead and supports a modular system that is easily modified to suit your specs or requirements.

9.6.3 CAN specifications

- Wired
- Serial
- Asynchronous
- Multi-Master No Slave (MMNS)
- Half duplex

9.6.4 CAN Transmission Handling

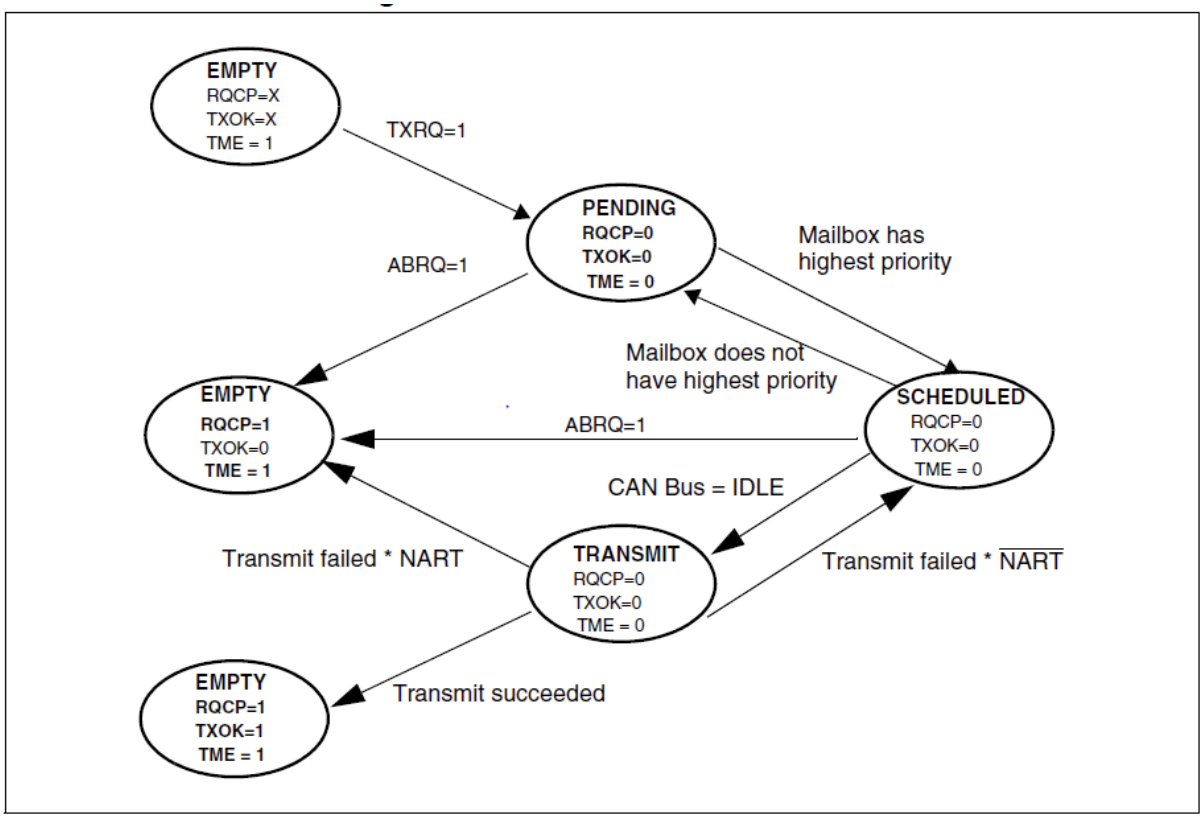


Figure 9-5: Transmit Mailbox States

1. In order to transmit a message, the application must select one empty transmit mailbox, setup the identifier, the data length code (DLC) and the data before requesting the transmission by setting the corresponding TXRQ bit (Transmit mailbox request) in the CAN_TIxR register.
2. When the mailbox left empty state and after the TXRQ bit has been set, the mailbox enters pending state and waits to become the highest priority mailbox (in our case the highest priority is defined by transmit request order) As soon as the mailbox has the highest priority it will be scheduled for transmission.
3. The transmission of the message will start (enter transmit state) when the CAN bus becomes idle, then once the mailbox message has been successfully transmitted by setting the RQCP(Request Complete) and TXOK(Transmission Ok) bits in the CAN_TSR register the mailbox becomes empty and we can use it again.

9.6.5 CAN Reception Handling

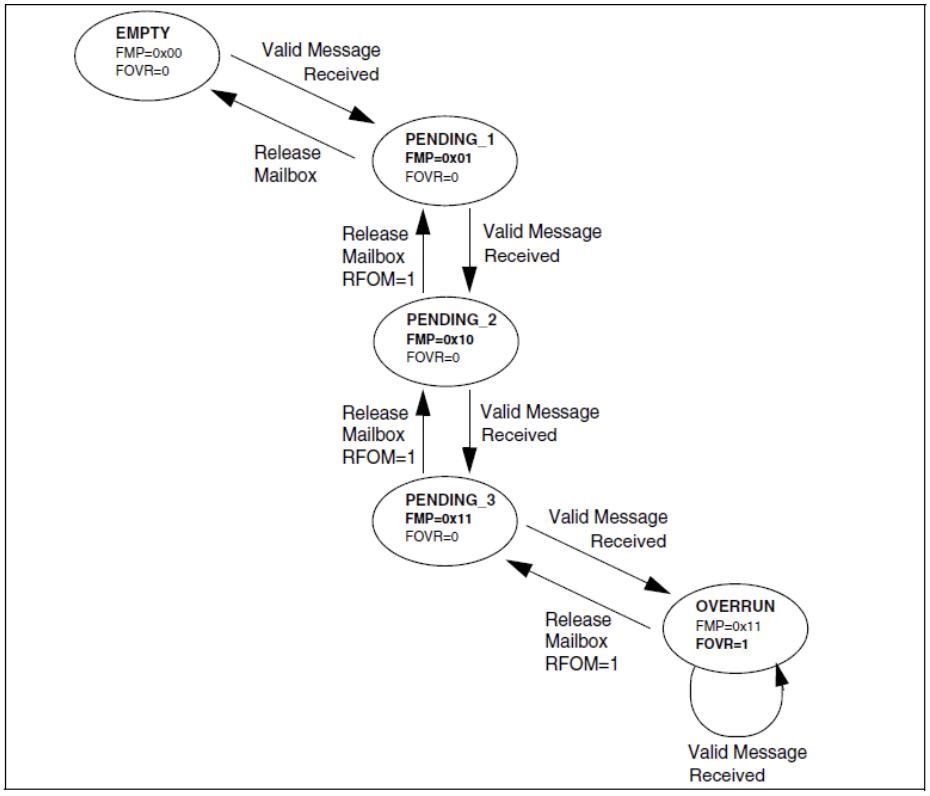


Figure 9-6: Receive FIFO States

We can receive up to three messages in the FIFO mailbox then if we start from the empty state, the first valid message received is stored in the FIFO which becomes pending_1 and if we receive the next valid message, it will be stored in the FIFO and enters pending_2 until we received three messages then we must release the FIFO mailbox in order to receive another messages

9.6.6 CAN Filters

We have 14 configurable and scalable filter banks (13-0) to the application in order to receive only the messages the software needs and each filter bank consists of two 32-bit registers.

- Mask Mode:

Filter masks are used to know which bits in the message identifier (ID) should be compared with the filter bits, as following:

The mask bit may be set to one or zero, if it is set to one the corresponding identifier bit will be compared with the filter bit, if they are matched the messages will be accepted otherwise it will be rejected. While, if a mask bit is set to zero, the corresponding identifier bit will automatically be accepted, regardless of the value of the filter bit.

Example 1:

We wish to accept only frames with ID of 00001657 (hexadecimal values)

Set filter to 00001657

Set mask to 1FFFFFFF

When a message is received its ID should be compared with the filter and all bits must match; if only one bit does not match the ID 00001567, this frame will be rejected.

Example 2:

We wish to accept the frames that have IDs from 00001650 to 0000165F

Set filter to 00001650

Set mask to 1FFFFFF0

When a message is received its ID should be compared with the filter and all bits except bits 0,1,2, and 3 must match; if any other bit does not match, this frame will be rejected.

- Identifier list mode:

We have a list of specific IDs, if the transmitted message carries ID that matches with anyone in the list, it will be accepted otherwise the frame is rejected.

9.6.7 CAN frame

Frame types:

- Error Frame – Reports error condition

Any node in the CAN network sender or receiver, may signal an error condition at any time during a data or remote frame transmission.

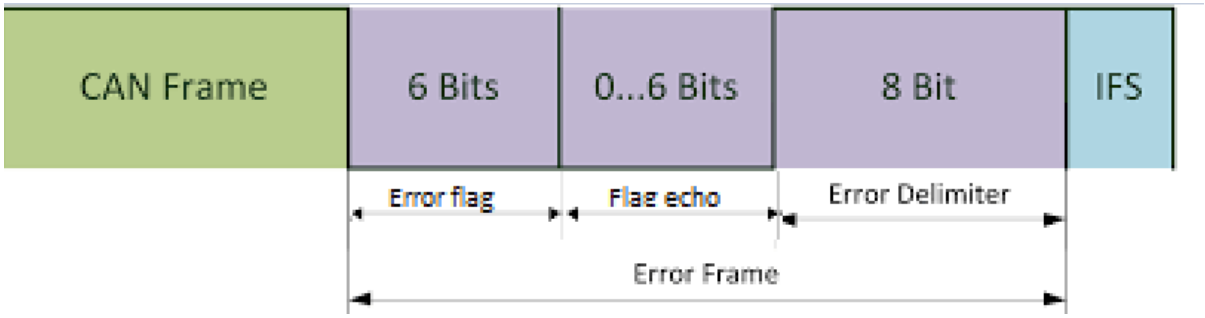


Figure 9-7: Error Frame

- Overload Frame – Reports node overload

A node sends overload frame to request a delay between two remote or data frames, so the overload frame can only occur between data or remote frame transmissions.

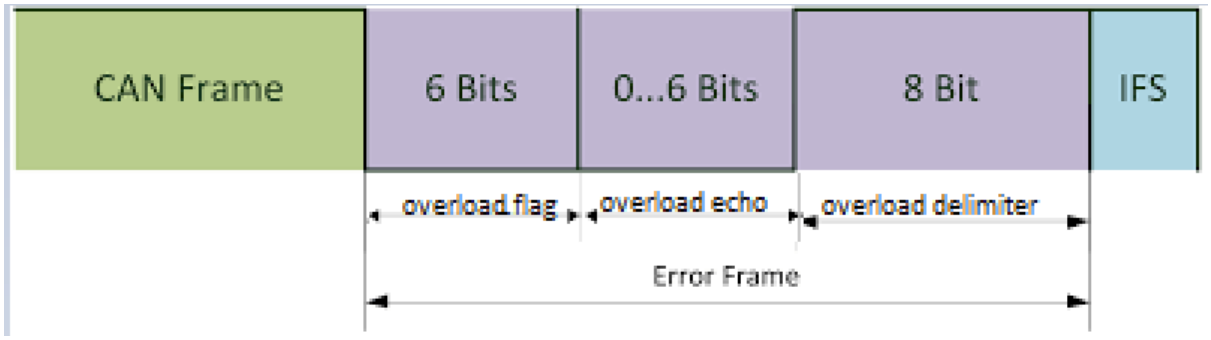


Figure 9-8: Error Frame

- Data Frame – Sends data

Data transfer from node to the CAN bus and the nodes that are interested in the message can read it (This is done according to the filters of the node).

- Remote Frame – Requests data

Any node may request data from other nodes. The remote frame represents the request so its consequently followed by a data frame containing the requested data.

Both data and remote frame have the same frame but in remote frame data = 0

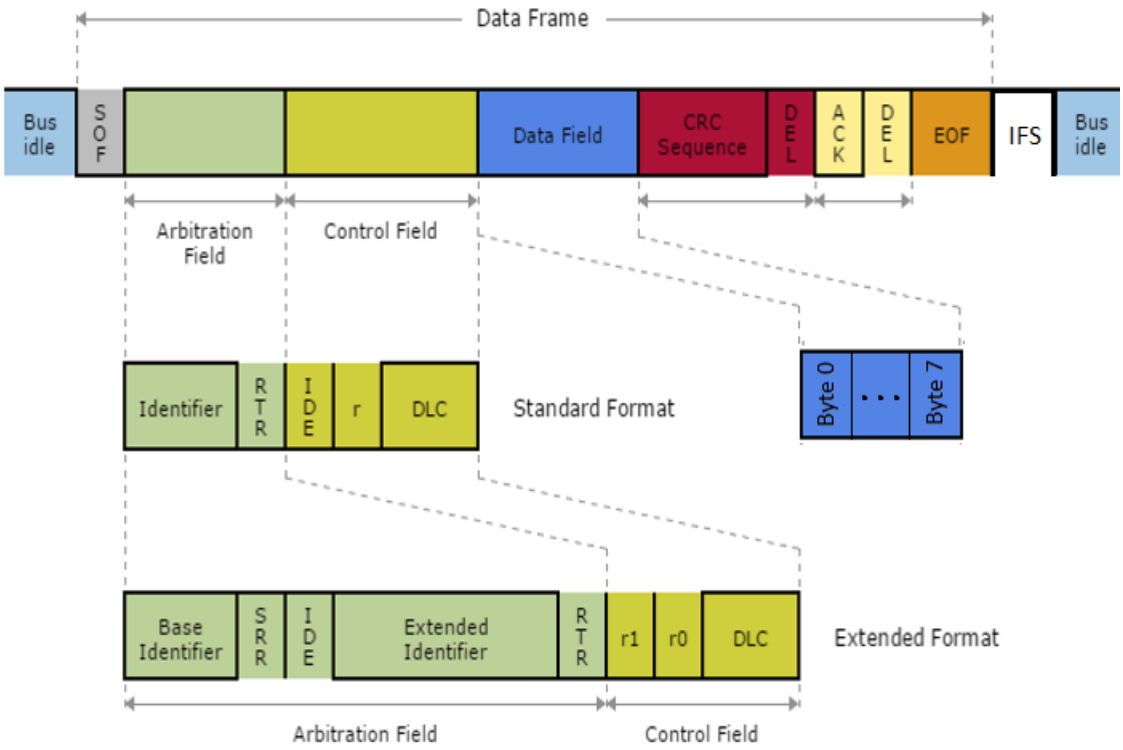


Figure 9-9: Data and Remote frames

- **SOF** (Start of Frame) (1 bit) – Marks the beginning of data and remote Frames
- **Arbitration Field** – Includes the message ID (11bits in standard ID and 29 bits in extended ID) and RTR (Remote Transmission Request) bit, which distinguishes data and remote frames.
- **Control Field** – DLC to determine data size (4 bits) and IDE (1 bit) to determine message ID length.
- **Data Field** – The actual data (which is zero in remote frame and contains the actual data frame)
- **CRC Field** (16 bits) – Checksum, CRC stands for Cyclic Redundancy Check, it contains Checksum and delimiter.
- **ACK Field** (2 bits) – Acknowledge and delimiter.
- **EOF** (End of Frame) (7 bits)– Marks the end of data and remote frames
- **IFS Field** (3 bits) – Inter-frame space

ACK delimiter: The acknowledgement is sent from the receiving node to the transmitting node and it need some time so ACK delimiter is used.

CRC delimiter: The ECU needs some time to calculate the CRC so a delimiter bit is introduced to make some delay for the ECU.

9.6.8 CAN in our project

The specifications that are suitable with our needs are:

- Speed : 500 Kbps.
- Priority : By transmit request order.
- Using 3 mailboxes each can have 8 bytes of data

We used CAN remote frames to control our system , here is the usage of remote frames in our system :

- Ask and get response from the user weather he wants to accept the update or not.
- Give ACK that the data is transmitted from main to app ECUs.
- Get the version on the app ECU
- Wait for any update request from main to app ECU

While data frames are used to transmit hex file from main ECU to app between ECUs and also transmit update progress from main ECU to GUI.

9.6.9 Problems that faced us with CAN

1. CAN Network should be supplied with 5v.
2. The connections using wires is not stable, try to make a PCB that contains the CAN network.

9.6.10 Procedures to follow to check if your network is working or not

1. First put a list of the ID's that each node can accept in the filter registrs
2. Try with only two nodes and only one mailbox and one FIFO.
3. Work with loopback mode on the first node (this mode transmits only) and with silent mode on the second one (this mode recives only).
4. If step 2 worked try to change the first node to silent mode and the other to loopback mode
5. If step 3 worked well , that's great , now try normal mode.
6. If step 5 worked now you should only scale the driver to use the 3 mailboxes and 2 FIFOs .

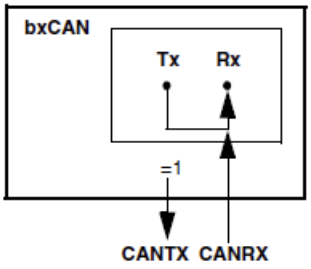


Figure 9-10: Node in silent mode

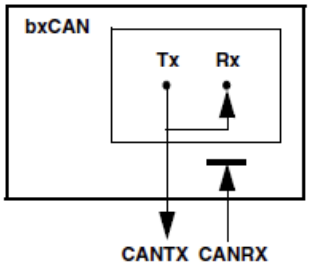


Figure 9-11: Node in loopback mode

This chapter was about our communication protocols that we used in our project and how each of them could work. The next chapter will talk about the Main ECU in our project which can connect between our server and all ECUs in our project.

CHAPTER 10

AUTOMOTIVE CYBERSECURITY

GRADUATION PROJECT 2024

10 Chapter 10: Automotive Cybersecurity

10.1 Introduction to Automotive Cybersecurity and its History

The automotive industry is on a fast track towards a future filled with connected and autonomous vehicles. This technological leap offers significant benefits for safety, convenience, and efficiency. However, these advancements rely heavily on Electronic Control Units (ECUs) and software, creating a complex and interconnected network within a car. This interconnectedness, while a driver of progress, introduces a new and critical vulnerability – cyberattacks.

10.1.1 The Rise of Connected Cars and Cybersecurity Concerns

The concept of a "connected car" refers to a vehicle equipped with internet connectivity, allowing it to communicate with external systems and devices. This connectivity enables features like:

Remote diagnostics and software updates
Real-time traffic information and navigation
Advanced driver-assistance systems (ADAS)
Vehicle-to-Everything (V2X) communication

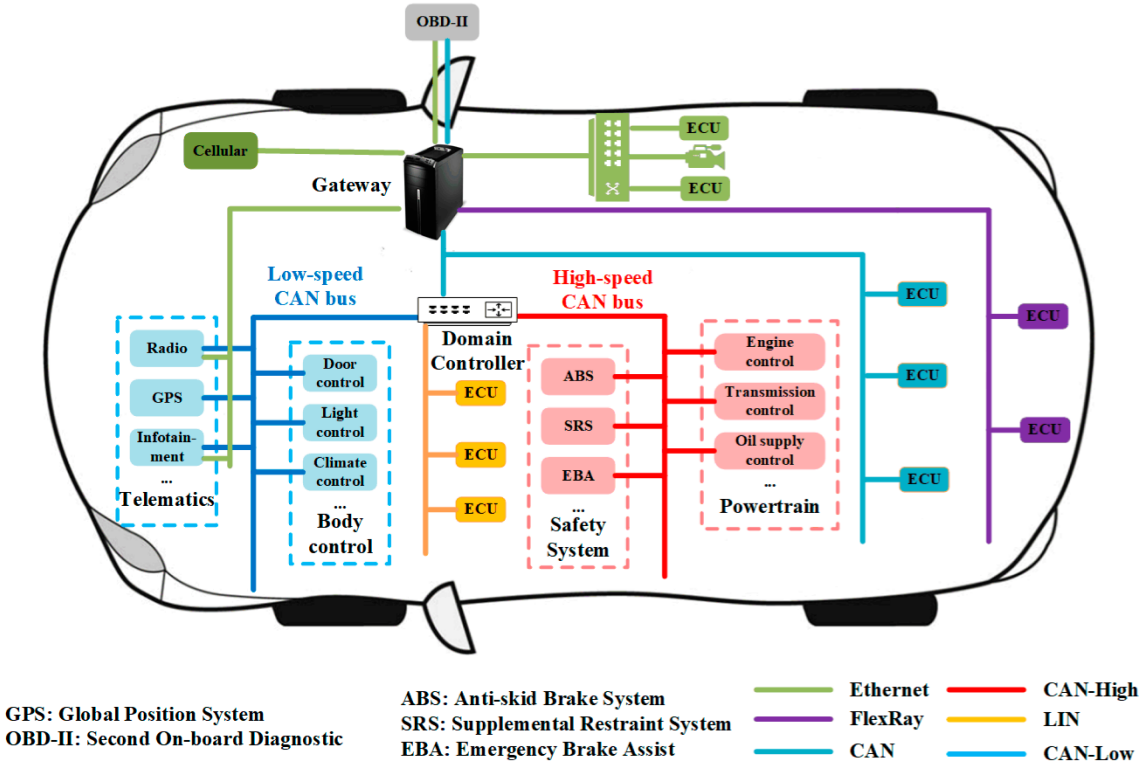


Figure 10-1: The Rise of Connected Cars and Cybersecurity Concerns

While these features enhance the driving experience, they also create a vast attack surface for malicious actors. Hackers can exploit vulnerabilities in software, communication protocols, or physical access points to compromise a vehicle's operation.

10.1.2 A Brief History of Automotive Cybersecurity

The history of automotive cybersecurity is relatively young. Traditionally, security concerns focused on physical tampering with vehicles, like hotwiring or disabling safety features. However, the landscape changed drastically with the rise of connected cars.

A pivotal moment occurred in 2015 when a team of hackers remotely compromised a Jeep Cherokee, demonstrating how vulnerabilities in software could allow unauthorized control of critical systems like steering and braking.

This incident sent shockwaves through the industry, highlighting the potential dangers of unsecured connected cars.

Since then, automotive cybersecurity has become a top priority for manufacturers, regulators, and researchers. The industry has witnessed the development of new security standards, best practices, and technologies to protect connected vehicles from cyberattacks.

10.2 The Threat Landscape in Automotive Cybersecurity

The potential consequences of cyberattacks on connected vehicles are severe. Here's a breakdown of the key threats:

Vehicle Control Systems: Hackers gaining access to critical systems like steering, braking, and engine control units could cause accidents, take control of the vehicle, or disable safety features.

Data Privacy: Modern vehicles collect a vast amount of data on driver behavior, location, and surrounding environment. Security breaches compromising this data could be a privacy nightmare, with potential misuse for identity theft or targeted advertising.

Diagnostics and Telematics: Telematics systems allow remote diagnostics and software updates. Unsecured channels could be exploited for unauthorized access, manipulating diagnostics data, or delaying critical software updates.

Supply Chain Attacks The complex automotive supply chain creates vulnerabilities. Malicious actors could target software development tools, embedded components, or third-party service providers to gain a foothold in connected vehicles.

10.3 Cryptography

Cryptography is the science of encrypting and decrypting messages and text, or the study of hidden writing and it is a method of protecting communications and information through the use of codes, so that we can prevent public from reading private messages, only those for whom the information is intended can read and process it. The cryptography word consists of two prefixes, crypto stands for hidden and graphy stands for writing.

10.3.1 Cryptography techniques

Cryptography is related to the disciplines of cryptanalysis and cryptology and. It includes

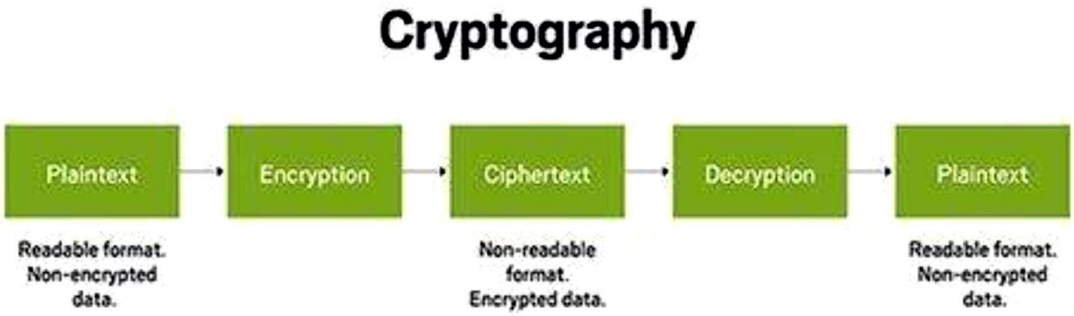


Figure 10-2 Cryptography system

techniques to hide information in storage or transit such as merging words with images, microdots and other ways. But, Today in computer-centric world, cryptography is about scrambling plaintext (Nonencrypted data) into ciphertext(Encrypted data) and this process is called **encryption**, then taking the ciphertext and backs it to plaintext and this process is called **decryption**. Cryptographers are the people who practice this field.

Modern cryptography should achieve the following four objectives:

1. **Confidentiality**: the information cannot be understood by anyone for whom it was unintended
2. **Integrity**: the information cannot be altered in storage or transit between sender and intended receiver without the alteration being detected
3. **Non-repudiation**: the creator/sender of the information cannot deny at a later stage his or her intentions in the creation or transmission of the information
4. **Authentication**: the sender and receiver can confirm each other's identity and the origin/destination of the information.

10.3.2 Cryptographic algorithm

Cryptosystems use ciphers or cryptographic algorithms, to encrypt and decrypt messages so that it can secure communications among devices such as smartphones, computer systems and applications. A cipher suite uses one algorithm for encryption, another algorithm for key exchange and another algorithm for message authentication. This process involves digital signing and verification for message authentication, and key exchange, public and private key generation for data encryption/decryption.

10.3.3 Types of cryptography

- **Single-key or symmetric-key encryption** (AES, DES): uses single key for encryption and decryption
- **Public-key or asymmetric-key encryption** (RSA): uses key for encryption and another key for decryption.
- **Hash functions** (SHA-1, SHA-2, SHA-3): uses mathematical transformation to encrypt data.

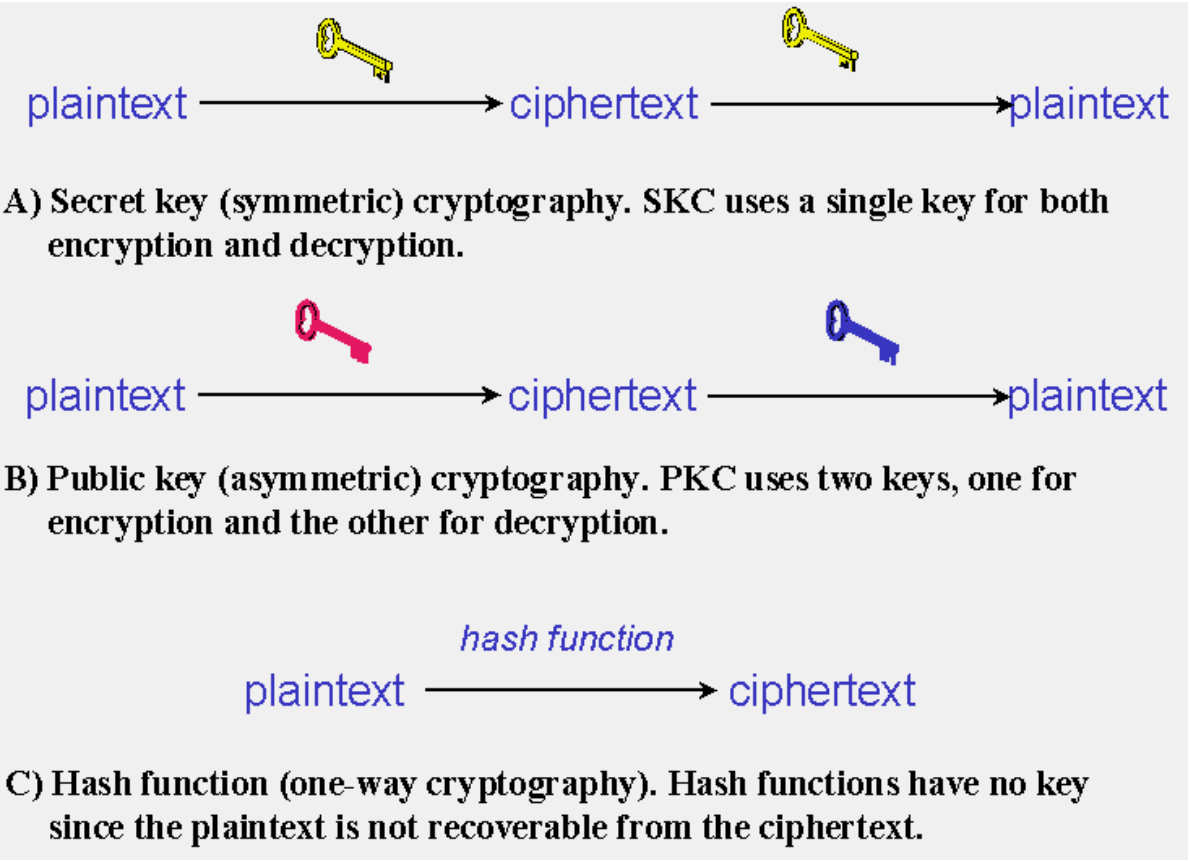


Figure 10-3 Three types of cryptography

10.4 Secure Boot on STM32 Microcontrollers

Secure boot is a fundamental security mechanism that ensures only authorized firmware is loaded onto an ECU during startup. This prevents unauthorized code execution and protects the integrity of the system.

10.4.1 Secure Boot Features of STM32

STM32 microcontrollers, widely used in automotive applications, offer various features for secure boot implementation:

Unique Device Identifier (UID): Each STM32 microcontroller has a unique identifier that can be used to verify the authenticity

10.4.2 Secure Key Storage

Option M on STM32 microcontrollers provides a dedicated secure key storage area. This secure enclave protects cryptographic keys used for secure boot and other security operations from unauthorized access.

10.4.3 Boot Process with Secure Boot

The secure boot process on STM32 typically involves the following steps:

1. Power-on Self-Test (POST): The microcontroller performs self-tests upon power-up.
2. Bootloader Verification: The bootloader code integrity is verified using cryptographic signatures.
3. Main Application Verification: The bootloader verifies the authenticity and integrity of the main application firmware using the UID and cryptographic keys stored in Option M.
4. Firmware Loading: If both verifications pass, the main application firmware is loaded onto the microcontroller's main memory for execution.

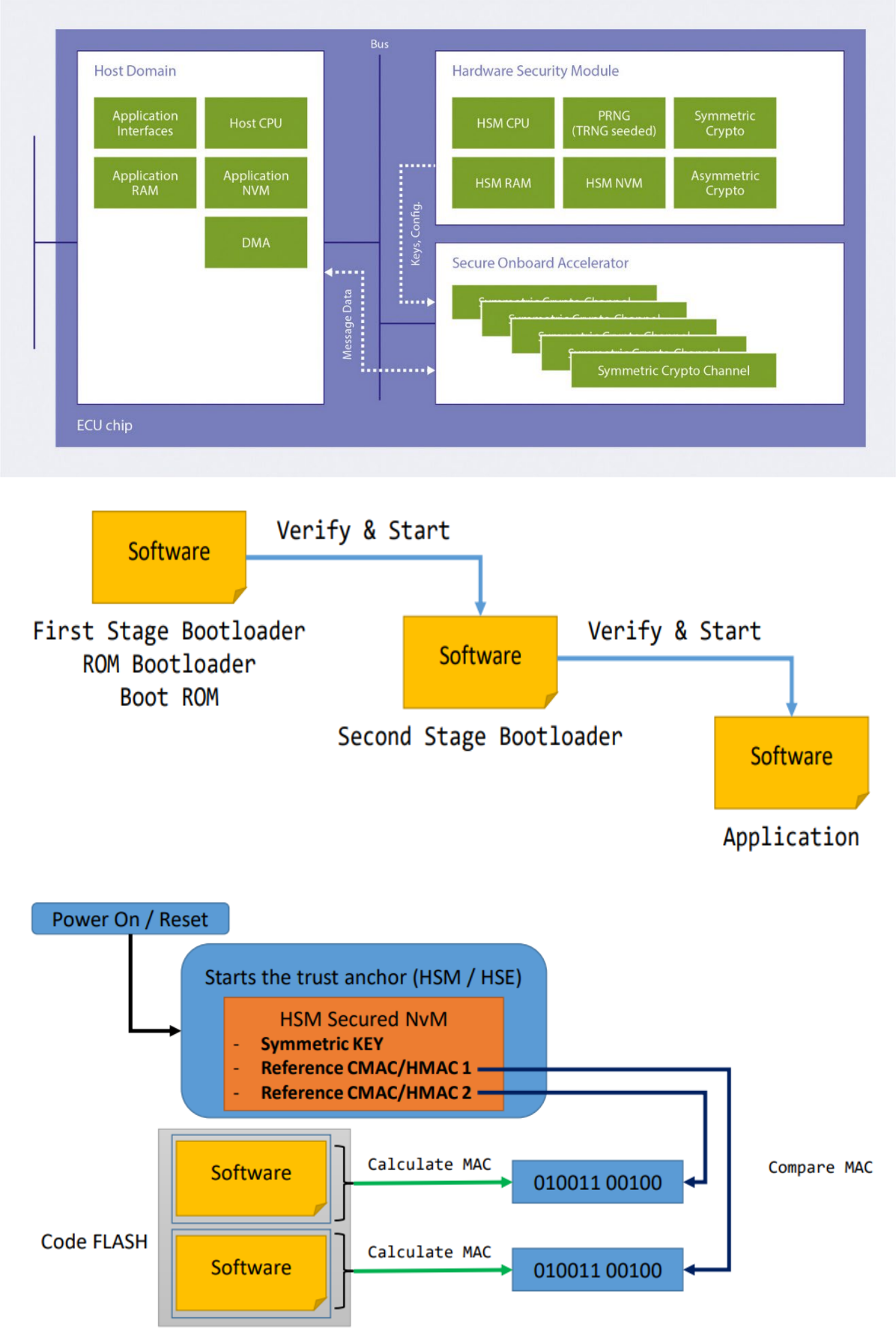


Figure 10-4: Secure boot operation

10.5 Secure Flash Programming on STM32

Flash memory is commonly used to store firmware in ECUs. Secure flash programming techniques are crucial for protecting the firmware from unauthorized modification.

10.5.1 Flash Protection Mechanisms on STM32

STM32 offers various flash protection mechanisms to enhance security:

Write Protection: Specific memory regions can be configured as read-only, preventing accidental or unauthorized modifications.

Option Bytes: Option bytes allow configuration of security features like write protection and enabling secure boot functionality.

Password Protection: Flash memory can be password-protected, requiring a valid password before any programming operations can be performed.

10.5.2 Secure Flash Programming Workflow

A secure flash programming workflow might involve the following steps:

1. Authentication: The programming tool authenticates itself with the microcontroller using a secure communication protocol.
2. Firmware Encryption: The firmware image is encrypted before being transferred to the microcontroller.
3. Flash Programming The encrypted firmware image is programmed into the flash memory using a secure communication channel.
4. Verification After programming, the integrity of the programmed firmware is verified using cryptographic hashes.

10.6 6. Secure Onboard Communication on STM32

Modern vehicles employ various communication protocols for data exchange between ECUs. These protocols can be vulnerable to eavesdropping, data tampering, and unauthorized message injection. Securing onboard communication is essential for protecting sensitive data and ensuring reliable operation.

10.6.1 Secure Communication Features of STM32

STM32 microcontrollers offer several features to facilitate secure communication:

Hardware Cryptographic Accelerators: These accelerators improve performance of cryptographic operations like encryption and decryption, essential for secure communication protocols.

Secure Sockets Layer (SSL/TLS): STM32 libraries support SSL/TLS, a widely used protocol for secure communication over networks. This ensures data confidentiality and integrity during communication between ECUs.

Secure Hash Algorithms (SHA): SHA algorithms can be used to generate message digests, allowing verification of data integrity and tamper detection.

10.6.2 Implementing Secure Communication on STM32*

Here's a simplified approach to implementing secure communication on STM32:

1. Configure cryptographic accelerators: Enable and configure the hardware accelerators for the chosen cryptographic algorithms (e.g., AES for encryption).

2. Establish secure connections: Utilize libraries like SSL/TLS to establish secure connections between ECUs. This involves key exchange, certificate verification, and secure data transmission channels.
3. Data encryption and signing: Implement encryption algorithms to protect the confidentiality of transmitted data. Additionally, digital signatures can be used to ensure data integrity and prevent tampering.

10.7 Conclusion

The increasing reliance on software and connectivity in modern vehicles necessitates a robust cybersecurity approach. This chapter provided an overview of key concepts in automotive cybersecurity, with a focus on secure boot, flash programming, and onboard communication on STM32 microcontrollers. By adopting a Secure Development Lifecycle (SDL) and utilizing security features offered by microcontrollers like STM32, engineers can contribute to building trustworthy and secure automotive systems.

This section can delve deeper into specific topics or emerging trends in automotive cybersecurity, such as:

- Security considerations for specific automotive applications like ADAS or V2X communication.
- Intrusion Detection and Prevention Systems (IDS/IPS) for automotive applications.
- The role of standardization and regulations in promoting automotive cybersecurity.

By incorporating these additional sections, you can create a more comprehensive chapter on automotive cybersecurity.

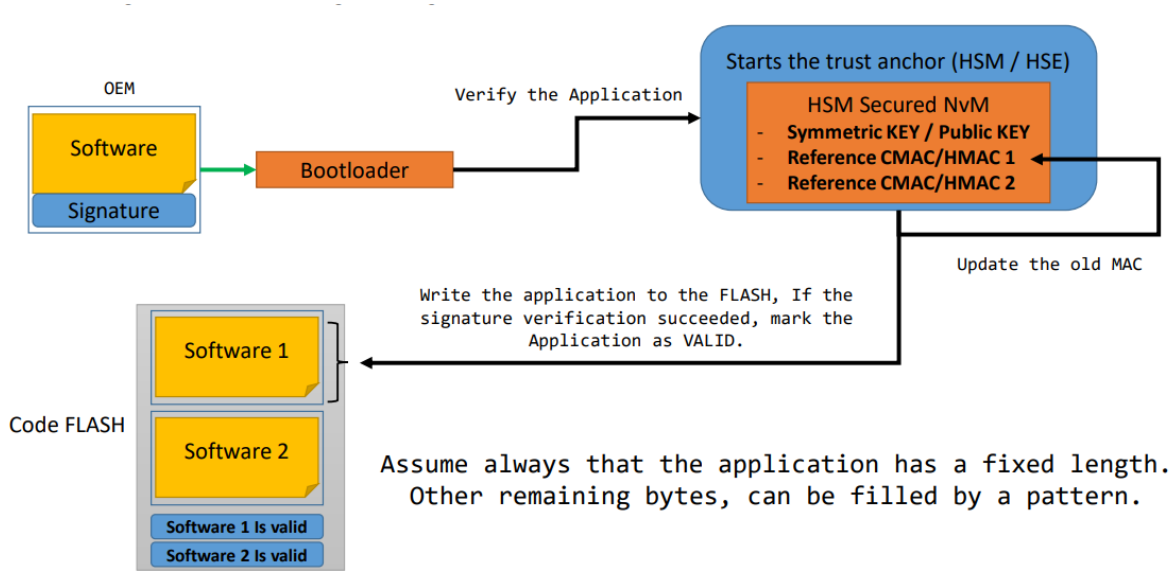
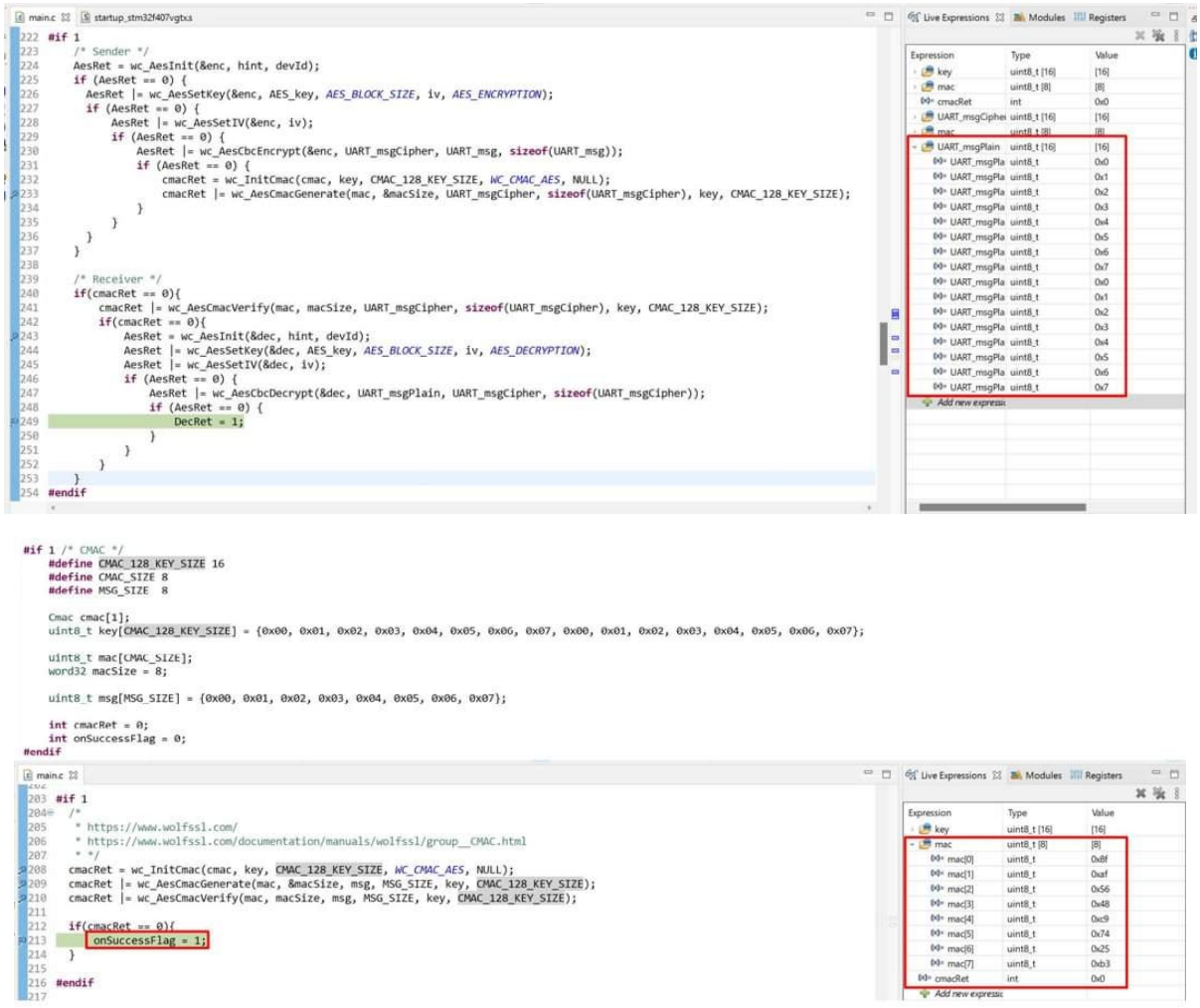


Figure 10-5: Secure flash programming



```
#if 1
    /* Sender */
    AesRet = wc_AesInit(&enc, hint, devId);
    if (AesRet == 0) {
      AesRet |= wc_AesSetKey(&enc, AES_key, AES_BLOCK_SIZE, iv, AES_ENCRYPTION);
      if (AesRet == 0) {
          AesRet |= wc_AesSetIV(&enc, iv);
          if (AesRet == 0) {
              AesRet |= wc_AesCbcEncrypt(&enc, UART_msgCipher, UART_msg, sizeof(UART_msg));
              if (AesRet == 0) {
                  cmacRet = wc_InitCmac(cmac, key, CMAC_128_KEY_SIZE, WC_CMAC_AES, NULL);
                  cmacRet |= wc_AesCmacGenerate(mac, &macSize, UART_msgCipher, sizeof(UART_msgCipher), key, CMAC_128_KEY_SIZE);
              }
          }
      }
    }

    /* Receiver */
    if(cmacRet == 0){
        cmacRet |= wc_AesCmacVerify(mac, macSize, UART_msgCipher, sizeof(UART_msgCipher), key, CMAC_128_KEY_SIZE);
        if(cmacRet == 0){
            AesRet = wc_AesInit(&dec, hint, devId);
            AesRet |= wc_AesSetKey(&dec, AES_key, AES_BLOCK_SIZE, iv, AES_DECRYPTION);
            AesRet |= wc_AesSetIV(&dec, iv);
            if (AesRet == 0) {
                AesRet |= wc_AesCbcDecrypt(&dec, UART_msgPlain, UART_msgCipher, sizeof(UART_msgCipher));
                if (AesRet == 0) {
                    DecRet = 1;
                }
            }
        }
    }
#endif
```

```
#if 1 /* CMAC */
    #define CMAC_128_KEY_SIZE 16
    #define CMAC_SIZE 8
    #define MSG_SIZE  8

    Cmac cmac[1];
    uint8_t key[CMAC_128_KEY_SIZE] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07};

    uint8_t mac[CMAC_SIZE];
    word32 macSize = 8;

    uint8_t msg[MSG_SIZE] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07};

    int cmacRet = 0;
    int onSuccessFlag = 0;
#endif
```

```
#if 1
  /*
   * https://www.wolfssl.com/
   * https://www.wolfssl.com/documentation/manuals/wolfssl/group__CMAC.html
   * */
  cmacRet = wc_InitCmac(cmac, key, CMAC_128_KEY_SIZE, WC_CMAC_AES, NULL);
  cmacRet |= wc_AesCmacGenerate(mac, &macSize, msg, MSG_SIZE, key, CMAC_128_KEY_SIZE);
  cmacRet |= wc_AesCmacVerify(mac, macSize, msg, MSG_SIZE, key, CMAC_128_KEY_SIZE);

  if(cmacRet == 0){
      onSuccessFlag = 1;
  }

#endif
```

Figure 10-6: Secure onboard communication

CHAPTER 11

FUTURE WORK

GRADUATION PROJECT 2024

11 Chapter 11: Future Work

11.1 Seamless FOTA

The main idea of the seamless FOTA is, to have two blocks of FLASH memory for code execution in each microcontroller of the single ECUs. The first block (Block A) is used to execute the actual code and the second block (Block B) is a spare FLASH block. So, the new software can be programmed into block B in the background while driving the car. Then, the code execution will be swapped from block A to block B and this will happen after all ECUs finish the pre-storing process. The SWAP will be completed with a final restart of the ECUs.

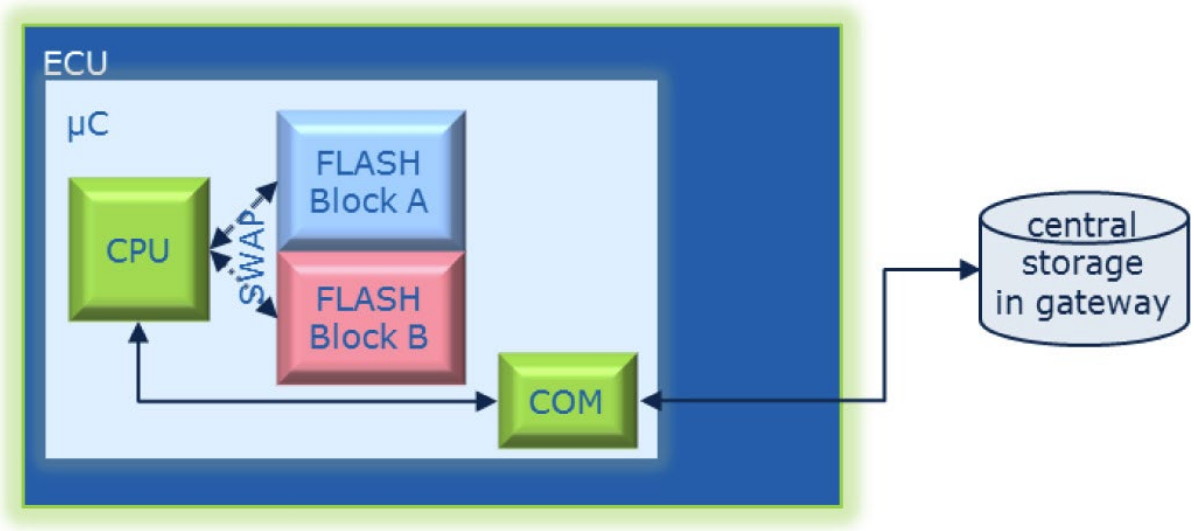


Figure 11-1: A/B swap process

11.1.1 Advantages and disadvantages of seamless FOTA

The advantages of seamless FOTA are:

1. Almost there is zero downtime of the vehicle.
2. A restart takes no more than a few milliseconds and then can be added to the power-up or -down cycle as mentioned before.

3. There is a fallback solution available within block A if something went wrong in the vehicles network.
4. There is always the option to switch back to block A of all ECUs of the actual service update within a few milliseconds, If the new software image of any ECU turns out to be corrupted.

The disadvantages of seamless FOTA are:

1. The size of embedded program FLASH is doubling and also today, the mechanisms of swapping process are not available for most microcontrollers.
2. The demand for swapping mechanism of a seamless memory is a challenge for any microcontroller supplier. The potential impact on the entire system architecture should be carefully considered in order to come to a robust implementation of the swap mechanism apart of the necessary availability of components with the required memory sizes.
3. When the FLASH of a microcontroller is doubled from 4MB to 8MB, this obviously will increase the cost.
4. The A/B Swap challenge approach is even graver for components at the upper memory limit of the actual technology note.

11.2 Delta file

11.2.1 What is delta file

Delta file is a very important feature that would add a very great value to the project, as it has a great effect in using large files, it saves time and storage .Delta file is mainly used to reduce the size of the sent file, it is a way of storing or transmitting data in the form of differences (deltas) between sequential data instead of complete files; this is known as data differencing. Delta encoding is also sometimes called **delta compression**.

11.2.2 Advantages of delta compression

- Reduce the potential for security flaws to be exploited before the affected software can be updated.
- Allow the Software to be updated more quickly.
- Reduce error due to wireless network.
- Reduce Transmission Time.

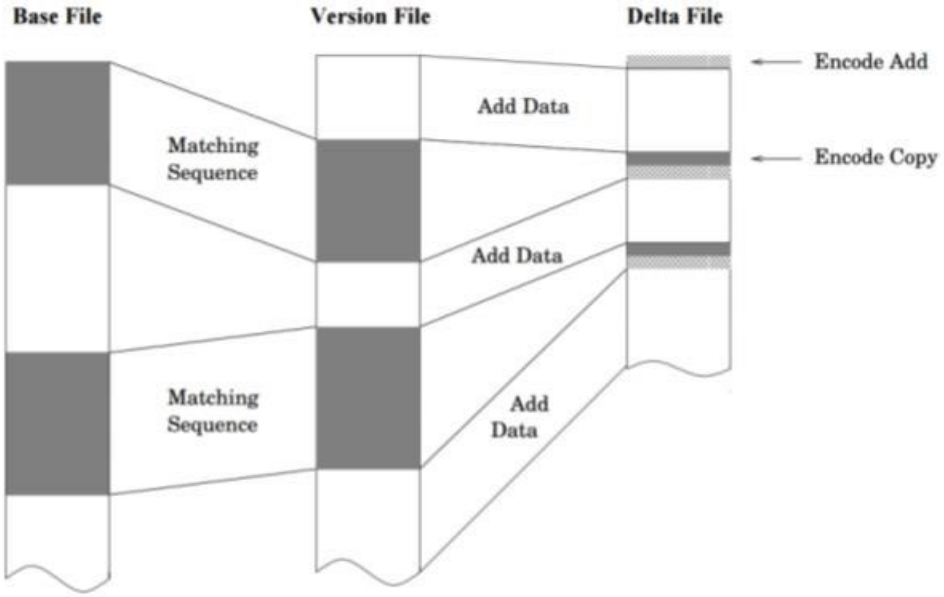


Figure 11-2: Delta File Algorithm

11.2.3 Generating the delta (Patch) file

- Patch File Consists of 3 files:

  1- Control File containing ADD and INSERT Instructions.

  - Each ADD instruction specifies an offset in the old file and a length; after that the correct number of bytes are read from the old file and added to the same number of bytes from the difference file.
  - INSERT instruction specify only a length; the specified number of bytes is read from the extra file.

  2- Difference File.

  3- Extra file.

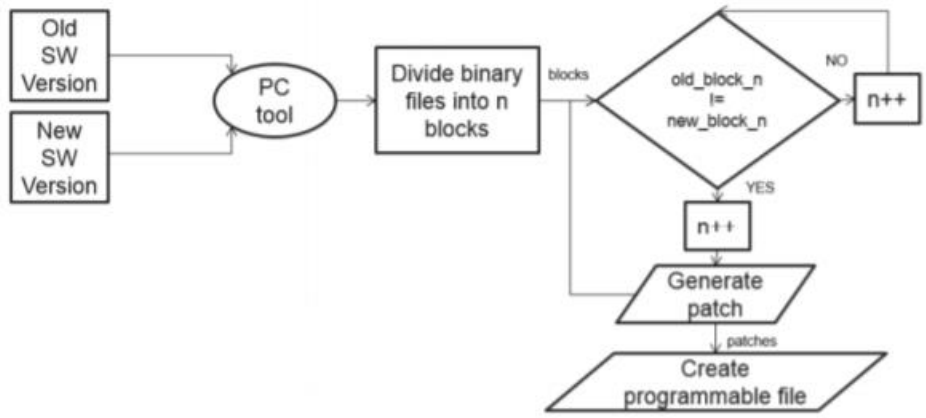


Figure 11-3: generating delta file

In FOTA delta file should be generated at the server using specific technique and only this file will be sent to the main ECU then after the ECU downloads the delta file it will mix both the old file and the delta file with each other and by using a specific algorithm, a new file will be generated a new file which should be burn on the microcontroller by the bootloader.

REFERENCES

GRADUATION PROJECT 2024

References

1) https://www.futurebridge.com/blog/over-the-air-software-updates-reaping-benefits-for-the-automotive-industry/
2) https://www.marketresearchfuture.com/reports/automotive-over-the-air-updates-market-7606
3) https://www.icpdas-usa.com/cancheck.html
4) https://www.totalphase.com/blog/2019/08/5-advantages-of-can-bus-protocol/
5) RM0008 Reference Manual
6) http://www.cse.dmu.ac.uk/~eg/tele/CanbusIDandMask.html
7) https://copperhilltech.com/blog/controller-area-network-can-bus-message-frame-architecture/
8) https://en.wikipedia.org/wiki/GSM
9) https://en.wikipedia.org/wiki/General_Packet_Radio_Service
10) https://en.wikipedia.org/wiki/4G
11) https://www.tutorialspoint.com/gsm/gsm_user_services.htm
12) https://en.wikipedia.org/wiki/GSM_services#:~:text=In%20order%20to%20gain%20access,has%20been%20consu%20med%20(postpaid)
13) https://www.researchgate.net/figure/Conventional-access-to-Internet-through-GSM-networks_fig1_2274364
14) SIM800L Datasheet
15) https://searchstorage.techtarget.com/definition/flash-memory
16) https://www.hyperstone.com/en/Solid-State-bit-density-and-the-Flash-Memory-Controller-1235,12728.html
17) PM0075 Programming manual, STM32F10xxx Flash memory microcontrollers
18) https://en.wikipedia.org/wiki/Intel_HEX
19) https://www.slant.co/versus/16724/22768/~tkinter_vs_pyqt
20) https://doc.qt.io/qt-5/classes.html

21) https://www.thalesgroup.com/en/markets/digital-identity-and-security/iot/industries/automotive/cybersecurity
22) https://www.vector.com/us/en/products/solutions/safety-security/automotive-cybersecurity/
23) https://elearning.vector.com/mod/page/view.php?id=333
24) https://www.embitel.com/blog/embedded-blog/what-is-flash-bootloader-and-nuances-of-an-automotive-ecu-re-programming

Our Project

Scan this QR code

To get our GitHub Project